Hadoop

Python Overview

Objectives

- What is Python?
- Variables
- Decisions and Truth
- Loops
- Lists
- Tuples
- Dictionaries
- Functions
- Command Line Arguments

What is Python?

- Python is a scripting / programming language that supports data types, exceptions, and objects.
- Python is well suited for processing large sets of data.
- Python interacts well in multiple operating systems. It gets along exceptionally well in UNIX / Linux environments.

Variables

- Python variables and identifiers should begin with an alpha character (or underscore).
- Python variables and identifiers are case sensitive.
- Python variables must have a value assigned to them before accessing (similar to Java)
- All Python variables have a type. That type may change throughout the running of a program. That type may be “NoneType” (which is similar to a Java null)
- Examples:
  - my_value = 10 # my_value is an int
  - my_value = “ten” # my_value is now a str
  - my_value = 10.10 # my_value is now a float
  - my_value = True # my_value is now a bool
  - my_value = None # my_value is now a NoneType

Decisions and Truth

- Truth:
  - True (one of two bool literals)
  - ‘X’ (a non-empty str)
  - ‘ ‘ (a str with a single space)
  - 5 (a non-zero int)
  - 3.3 (a non-zero float)
  - 0.000000000000001 (a non-zero float)

Decisions and Truth

- Falseness:
  - False (the other bool literal)
  - ‘’ (an empty str – quote-quote or doublequote-doublequote)
  - 0 (an int with a zero value)
  - 0.0 (a float with a zero value)
  - None (a variable of type NoneType, AKA a variable with a value of None)

Decisions and Truth

- operators (for all types of data):

– ==	Equal
– <	Less than
– >	Greater than
– <=	Less than or equal to
– >=	Greater than or equal to
– !=	Not equal to
– is	Same object
– is not	Different object

Decisions and Truth

- The “if” statement
  - The statement must be terminated with a colon (:)
  - The block to execute as a result of a true condition must be indented a consistent amount (spaces or tabs, technically, but preferred to use spaces)
  - There must be a block to be executed. If an empty statement needs to be provided, use the keyword “pass”
  - examples:

```
if  my_value == “yes”:
        pass
if my_value:
        print(“something is there”)
```

Decisions and Truth

- and / or - Logical “and” and logical “or”

  – examples:

```
if  my_value == “yes” and your_value:
        print(“We both have a value”)

if my_value or your_value:
        print(“something is there”)
        print(my_value or your_value) #print the first True value
```

Decisions and Truth

- else and elif
  - examples:

```
if  my_value:
    print(“I have a value”)
else:
    print(“I do not have a value”)

if  my_value:
    print(“I have a value”)
elif your_value:
    print(“You have a value”)
else:
    print(“No one has a value”)
```

Loops

- The for loop # Process things until you run out

- # Process literal words in a list
- for word in ["Do", "The", "Chickens", "Have", "Large", "Talons"]:
- 	print word

- # Process letters
- words = "Do the chickens have large talons"
- for word in words:
- 	print word

- # Process a list of words
- words = words.split()
- for word in words:
- 	print word

Loops

- More for loop

- # Print numbers 0 thru 19
- for value in range(20):
- print (value)

- # Print numbers 1 thru 9
- for value in range(1, 10):
- print (value)

- # Print even numbers from 2 thru 20
- for value in range(2, 21, 2):
- print value

Loops

- More for loop

```
#  Print every third word in words
words = "Do the chickens have large talons"
words = words.split()
for value in range(0, len(words),3):
    print words[value]

#  Print every third word in words but backwards
words = "Do the chickens have large talons"
words = words.split()
for value in range(len(words) – 1, 0, -3):
    print words[value]
```

Loops

- The while loop # Process things until no longer True

- words = "Do the chickens have large talons"
- words = words.split()

- list_length = len(words)
- counter = 0
- while counter < list_length:
- 	print(words[counter])
- 	counter += 1

Strings

- Immutable (like Java strings)
- Data type “str”
- Many methods. Use dir(str) for list of methods. Useful methods include:
  - capitalize() # Capitalize first letter in string
    - “texas”.capitalize() # “Texas”
    - “houston texas” # “Houston texas”
  - count() # Count substring occurrences within string
    - “san antonio, texas”.count(“a”) # 3
  - endswith() # True or False, string ends with substring?
    - “austin”.endswith(“tin”) # True
    - “bandera”.endswith(“band”) # False
  - find() # Find index of where substring is found in string.
    - "san antonio, texas".find("a") # 1
    - "san antonio, texas".find("a",3) # 4
    - "san antonio, texas".find("a",5) # 16
    - "san antonio, texas".find”(z”) # -1

Strings

- format() # Return formatted string (printf'ish)
  - "{:0.2f}".format(123.4567) # "123.46"
  - "Salary for {:s} is {:0.2f}".format("Theodore", 123.4567)
    - "Salary for Theodore is 123.46"

- index() # Similar to find but ValueError on not found
  - "san antonio, texas".find"(z") # !! ValueError

- isalpha() # True or False, all alpha characters?
  - "aaa1bbb".isalpha() # False
  - "aaaLccc".isalpha() # True

- isdigit() # True or False, all digits within string?
- "123a456".isdigit() # False
- "123456".isdigit() # True
- "123.456".isdigit() # False

- join() # Join many things, producing one string
  - "".join(["Theodore", "Cleaver"] # "TheodoreCleaver"
  - ",".join(["Theodore", "Cleaver"] # "Theodore,Cleaver"

Strings

- lower() # Return the lower cased version of a string
  - "Your Mom Goes To College".lower()
    - # your mom goes to college
- replace() # Replaces substrings with other string values
  - elvis presley".replace("elvis", "costello") # "elvis costello"
- split() # Break a string into multiple substrings
  - "10 20 30 40".split() # [10, 20, 30, 40]
  - "10,20,30,40".split(",") # [10, 20, 30, 40]
- startswith() # True or False, string starts with substring?
  - "Denver, CO".startswith("Dallas") # False
  - "San Antonio, TX".startswith("San") # True
- upper() # Return the upper cased version of a string
  - "five Ten fifTeen".upper() # "FIVE TEN FIFTEEN"

Strings

- Substrings

  - quote = “I triple dog dare you”
  - quote[2: 8] # “triple”
  - quote[2: 6] # “trip”
  - quote[9 :] # “dog dare you”
  - quote [0 : 6] # “I trip”
  - quote [: 6] # “I trip”

- Operators

  - toy = “Yoyo”
  - toy + toy # “YoYoYoYo”
  - toy * 2 # “YoYoYoYo”
  - (toy * 2).upper() # “YOYOYOYO”
  - toy.upper * 2 # “YOYOYOYO”
  - “Yoy” in toy # True
  - “yoy” in toy # False
  - “YOY” in toy.upper() # True

Lists

- Mutable collections
- Indicated by surrounding with square brackets
- May be constructed with list()
- May be constructed with assignment:
  - my_list = [“The”, “Rain”, “In”, “Spain”]
  - my_list = [] # Empty List
  - my_list = [“Hello”] # One element list
  - my_not_a_list = “Hello” # String
- Elements may be of mixed / complex types.
- Initial index position is 0
- [10, 20, 30, 40] # 4 elements, indexed by 0 thru 3

Lists

- Accessing individual list elements

- name_list = [“Bob”, “Carol”, “Ted”, “Alice”]
- print(name_list[2]) # “Ted”
- print(name_list) # [Everybody]
- print(name_list[0 : 2] # [“Bob”, “Carol”]
- print(name_list[1 : 3] # [“Carol”, “Ted”]
- print(name_list[2 :] # [“Ted”, “Alice”]
- print(name_list[: 2] # [“Bob”, “Carol”]
- print(name_list[:] # [Everybody]
- name_list[1] = “Willie” # No more “Carol”
- name_list[2 : 4] = (“Willie”, “Waylon”) # “Ted” & “Alice” gone

Lists

- There are many list methods, including:
  - append() # Add an element at the end
  - pop() # Remove an element
  - insert() # Add an element somewhere
  - remove() # Remove an element by value
  - extend() # Add another list to the end
  - index() # Find a value's position in the list
  - sort() # Resequence the list in sorted sequence
  - reverse() # Reverse the order of the elements

- + operator for combining lists. * operator replicating lists.
  - new_list = ["Begining", "Stuff"] + ["Stuff", "At", "The", "End"]
  - Length of new_list is 6
  - new_list = new_list + new_list
  - Length of new_list is now 12
  - new_list = new_list * 4
  - Length of new_list is now 48

Tuples

- Immutable collections
- May be constructed with tuple()
- Indicated by surrounding with parenthesis (Most of the time)
- May be constructed with assignment:
  - my_tuple = (10,20,30,40) # Four element tuple
  - my_tuple = () # Empty tuple
  - my_not_a_tuple = (“Batman”) # String
  - my_tuple = (“Batman”,) # One element tuple
  - my_tuple = (“Falls”, “Mainly”, “In”, “The”, “Plain”)
- Elements may be of mixed / complex types.
- Initial index position is 0
- ([10,20], “Rico”, 7, “Napoleon”) # 4 element tuple

Tuples

- Tuples support just about everything that lists support as long as the operator or method does not attempt to modify the tuple.

- Examples of valid tuple expressions:
  - print(my_tuple[1])
  - print(my_tuple * 3)
  - new_tuple = my_tuple + my_tuple + your_tuple

- Examples of invalid tuple expressions:
  - my_tuple[1] = “Jerry Jeff”
  - my_tuple.append(“Garth”)
  - my_tuple.pop()

- A few tuple methods
  - count() # How many matching items are in the tuple?
  - index() # Index of first matching item in the tuple.

Dictionaries

- Mutable key-value pair collection (like a java Map or a Perl hash)
- Indicated by surrounding with curly brackets
- May be constructed with dict()
  - my_dict = dict(TX="Texas", OK="Oklahoma", AR="Arkansas")
- May be constructed with assignment:
  - my_dict = {"TX" : "Austin", "LA" : "Baton Rouge" }
  - my_dict = {} # Empty dictionary
  - my_dict["OK"] = "Tulsa" # Add a key-value pair
  - my_dict["OK"] = "Oklahoma City" # Update a key-value pair
- Keys and values may be of any type but it's usual for them to be consistent within the dictionary.
- Dictionaries are unordered. Keys are unique. You can't control a dictionary's internal structure.

Dictionaries

- There are many dictionary methods, including:
  - clear() # Delete all dictionary pairs
  - get() # Fetch a value by key
  - has_key() # True or False. Key exists in dictionary
  - items() # Returns a list of all key-value pairs
  - keys() # Returns a list of all keys
  - pop() # Return and remove a key-value pair
  - update() # Update dict with pairs from another dict
  - values() # Return a list of all values

- Dictionaries have more methods than those listed above but this is a good start.

Functions

- def (keyword to start a function definition)
- return (keyword to end a function and send an optional result)
- indentation defines body of function

- def properCase(myString):
- 	myResult = ""
- 	myList = myString.split()
- 	for word in myList:
- 		myResult += (word[0].upper() + word[1:].lower() + " ")
- 	return myResult.strip()

Functions

- lambda functions

  – Small, anonymous functions that contain a single expression and return that expression's result.
  – lambda functions are often passed as arguments to transformation functions.
  – Example:
    - def namedFunction(val):
  – valueLength = len(val.strip())
     convertedValue = val.upper().strip()
     return (convertedValue, valueLength)
  – As a lambda function
    - lambda val : (val.upper().strip(), len(val.strip()))

Functions

- lambda functions continued

- Example:
  - x = lambda val : (val.upper().strip(), len(val.strip()))
  - print(x(“roll over beethoven”))
    - ('ROLL OVER BEETHOVEN', 19)
- or:
  - print((lambda x : x * 2)(50))
    - 100
- or:
  - print((lambda x, y : x * y ** 2)(3, 4))
    - 48
- or:
  - print((lambda val: "{:.2f}".format(val * 3.14125 * 2))(7))
    - 43.98

Command Line Arguments

- Anytime you reference command line arguments in Python, you'll need to import the sys module
- Example:
  - import sys

- Your command line arguments may then be referenced via the sys.argv list. sys.argv will always have a minimum length of 1. Element 0 in the list is your script file name.

- Example:
  - myPythonScript.py aaa bbb ccc
  - Inside the myPythonScript.py:
    - print(sys.argv) would yield:
      - ['./myPythonScript.py', 'aaa', 'bbb', 'ccc']

      - if len(sys.argv) > 1:
      - print(sys.argv[1] # Will print aaa