CS2180: Artificial Intelligence Lab (Jan – Apr 2019)

Lab 1 (Introduction)

Note: The majority of this document is based on UC Berkeley CS188 Intro to AI -- Course Materials (available online at http://ai.berkeley.edu/tutorial.html)

Operators

The Python interpreter can be used to evaluate expressions, for example simple arithmetic expressions. If you enter such expressions at the prompt (`>>>`) they will be evaluated and the result will be returned on the next line.

```
>>> 1 + 1
2
>>> 2 * 3
6
```

Boolean operators also exist in Python to manipulate the primitive `True` and `False` values.

```
>>> 1==0
False
>>> not (1==0)
True
>>> (2==2) and (2==3)
False
>>> (2==2) or (2==3)
True
```

Strings

Like Java, Python has a built in string type. The `+` operator is overloaded to do string concatenation on string values.

```
>>> 'artificial' + "intelligence"
'artificialintelligence'
```

There are many built-in methods which allow you to manipulate strings.

```
>>> 'artificial'.upper()
'ARTIFICIAL'
>>> 'HELP'.lower()
'help'
>>> len('Help')
4
```

Notice that we can use either single quotes `' '` or double quotes `" "` to surround string. This allows for easy nesting of strings.

We can also store expressions into variables.

```
>>> s = 'hello world'
>>> print s
hello world
>>> s.upper()
'HELLO WORLD'
>>> len(s.upper())
11
>>> num = 8.0
>>> num += 2.5
>>> print num
10.5
```

In Python, you do not have declare variables before you assign to them.

Exercise: Dir and Help

Learn about the methods Python provides for strings. To see what methods Python provides for a datatype, use the `dir` and `help` commands:

```
>>> s = 'abc'

>>> dir(s)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__',
'__eq__', '__ge__', '__getattribute__', '__getitem__',
'__getnewargs__', '__getslice__', '__gt__', '__hash__',
'__init__','__le__', '__len__', '__lt__', '__mod__', '__mul__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__','__repr__',
'__rmod__', '__rmul__', '__setattr__', '__str__', 'capitalize',
'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip',
'replace', 'rfind','rindex', 'rjust', 'rsplit', 'rstrip', 'split',
'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']

>>> help(s.find)

Help on built-in function find:

find(...)
    S.find(sub [,start [,end]]) -> int
```

```
    Return the lowest index in S where substring sub is found,
    such that sub is contained within s[start,end].  Optional
    arguments start and end are interpreted as in slice notation.

    Return -1 on failure.


>> s.find('b')
1
```

Try out some of the string functions listed in `dir` (ignore those with underscores '_' around the method name).

Built-in Data Structures

Python comes equipped with some useful built-in data structures, broadly similar to Java's collections package.

Lists

Lists store a sequence of mutable items:

```
>>> fruits = ['apple','orange','pear','banana']
>>> fruits[0]
'apple'
```

We can use the + operator to do list concatenation:

```
>>> otherFruits = ['kiwi','strawberry']
>>> fruits + otherFruits
>>> ['apple', 'orange', 'pear', 'banana', 'kiwi', 'strawberry']
```

Python also allows negative-indexing from the back of the list. For instance, `fruits[-1]` will access the last element `'banana'`:

```
>>> fruits[-2]
'pear'
>>> fruits.pop()
'banana'
>>> fruits
['apple', 'orange', 'pear']
>>> fruits.append('grapefruit')
>>> fruits
['apple', 'orange', 'pear', 'grapefruit']
>>> fruits[-1] = 'pineapple'
>>> fruits
['apple', 'orange', 'pear', 'pineapple']
```

We can also index multiple adjacent elements using the slice operator. For instance, `fruits[1:3]`, returns a list containing the elements at position 1 and 2. In general `fruits[start:stop]` will get the elements in `start, start+1, ..., stop-1`. We can also do `fruits[start:]` which returns all elements starting from the `start` index. Also `fruits[:end]` will return all elements before the element at position `end`:

```
>>> fruits[0:2]
['apple', 'orange']
>>> fruits[:3]
['apple', 'orange', 'pear']
>>> fruits[2:]
['pear', 'pineapple']
>>> len(fruits)
4
```

The items stored in lists can be any Python data type. So for instance we can have lists of lists:

```
>>> lstOfLsts = [['a','b','c'],[1,2,3],['one','two','three']]
>>> lstOfLsts[1][2]
3
>>> lstOfLsts[0].pop()
'c'
>>> lstOfLsts
[['a', 'b'],[1, 2, 3],['one', 'two', 'three']]
```

Exercise: Lists

Play with some of the list functions. You can find the methods you can call on an object via the `dir` and get information about them via the `help` command:

```
>>> dir(list)
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__',
'__delslice__', '__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__',
'__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__',

'__rmul__', '__setattr__', '__setitem__', '__setslice__', '__str__',
'append', 'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse',
'sort']
```

```
>>> help(list.reverse)
```

```
Help on built-in function reverse:

reverse(...)
    L.reverse() -- reverse *IN PLACE*

>>> lst = ['a','b','c']
>>> lst.reverse()
>>> ['c','b','a']
```

Note: Ignore functions with underscores "_" around the names; these are private helper methods. Press 'q' to back out of a help screen.

Tuples

A data structure similar to the list is the *tuple*, which is like a list except that it is immutable once it is created (i.e. you cannot change its content once created). Note that tuples are surrounded with parentheses while lists have square brackets.

```
>>> pair = (3,5)
>>> pair[0]
3
>>> x,y = pair
>>> x
3
>>> y
5
>>> pair[1] = 6
TypeError: object does not support item assignment
```

The attempt to modify an immutable structure raised an exception. Exceptions indicate errors: index out of bounds errors, type errors, and so on will all report exceptions in this way.

Sets

A *set* is another data structure that serves as an unordered list with no duplicate items. Below, we show how to create a set, add things to the set, test if an item is in the set, and perform common set operations (difference, intersection, union):

```
>>> shapes = ['circle','square','triangle','circle']
>>> setOfShapes = set(shapes)
>>> setOfShapes
set(['circle','square','triangle'])
>>> setOfShapes.add('polygon')
>>> setOfShapes
set(['circle','square','triangle','polygon'])
>>> 'circle' in setOfShapes
True
>>> 'rhombus' in setOfShapes
```

```
False
>>> favoriteShapes = ['circle','triangle','hexagon']
>>> setOfFavoriteShapes = set(favoriteShapes)
>>> setOfShapes - setOfFavoriteShapes
set(['square','polyon'])
>>> setOfShapes & setOfFavoriteShapes
set(['circle','triangle'])
>>> setOfShapes | setOfFavoriteShapes
set(['circle','square','triangle','polygon','hexagon'])
```

Note that the objects in the set are unordered; you cannot assume that their traversal or print order will be the same across machines!

Dictionaries

The last built-in data structure is the *dictionary* which stores a map from one type of object (the key) to another (the value). The key must be an immutable type (string, number, or tuple). The value can be any Python data type.

Note: In the example below, the printed order of the keys returned by Python could be different than shown below. The reason is that unlike lists which have a fixed ordering, a dictionary is simply a hash table for which there is no fixed ordering of the keys (like HashMaps in Java). The order of the keys depends on how exactly the hashing algorithm maps keys to buckets, and will usually seem arbitrary. Your code should not rely on key ordering, and you should not be surprised if even a small modification to how your code uses a dictionary results in a new key ordering.

```
>>> studentIds = {'knuth': 42.0, 'turing': 56.0, 'nash': 92.0 }
>>> studentIds['turing']
56.0
>>> studentIds['nash'] = 'ninety-two'
>>> studentIds
{'knuth': 42.0, 'turing': 56.0, 'nash': 'ninety-two'}
>>> del studentIds['knuth']
>>> studentIds
{'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds['knuth'] = [42.0,'forty-two']
>>> studentIds
{'knuth': [42.0, 'forty-two'], 'turing': 56.0, 'nash': 'ninety-two'}
>>> studentIds.keys()
['knuth', 'turing', 'nash']
>>> studentIds.values()
[[42.0, 'forty-two'], 56.0, 'ninety-two']
>>> studentIds.items()
[('knuth',[42.0, 'forty-two']), ('turing',56.0), ('nash','ninety-
two')]
```

```
>>> len(studentIds)
3
```

As with nested lists, you can also create dictionaries of dictionaries.

Exercise: Dictionaries

Use `dir` and `help` to learn about the functions you can call on dictionaries.

Loops

Now that you've got a handle on using Python interactively, let's write a simple Python script that demonstrates Python's `for` loop.

```
# This is what a comment looks like
fruits = ['apples','oranges','pears','bananas']
for fruit in fruits:
    print fruit + ' for sale'

fruitPrices = {'apples': 200, 'oranges': 150, 'pears': 175}
for fruit, price in fruitPrices.items():
    if price <= 175:
        print '%s cost %f Rs per kg' % (fruit, price)
    else:
        print fruit + ' are too expensive!'
```

Remember that the print statements listing the costs may be in a different order on your screen than in this tutorial; that's due to the fact that we're looping over dictionary keys, which are unordered.

List Comprehensions

The next snippet of code demonstrates Python's *list comprehension* construction:

```
nums = [1,2,3,4,5,6]
plusOneNums = [x+1 for x in nums]
oddNums = [x for x in nums if x % 2 == 1]
print oddNums
oddNumsPlusOne = [x+1 for x in nums if x % 2 ==1]
print oddNumsPlusOne
```

Exercise: List Comprehensions

Write a list comprehension which, from a list, generates a lowercased version of each string that has length greater than five.

Writing Functions

In Python you can define your own functions:

```
fruitPrices = {'apples':200, 'oranges': 150, 'pears': 175}
```

```
def buyFruit(fruit, numKg):
    if fruit not in fruitPrices:
        print "Sorry we don't have %s" % (fruit)
    else:
        cost = fruitPrices[fruit] * numKg
        print "That'll be %f please" % (cost)
```

You can call the above function as follows

```
    buyFruit('apples',24)
    buyFruit('coconuts',2)
```

Exercise: Writing Functions

Write a `quickSort` function in Python using list comprehensions. Use the first element as the pivot.

Defining Classes

Here's an example of defining a class named `FruitShop`:

```
class FruitShop:

    def __init__(self, name, fruitPrices):
        """
            name: Name of the fruit shop
            fruitPrices: Dictionary with keys as fruit
            strings and prices for values
        """
        self.fruitPrices = fruitPrices
        self.name = name
        print 'Welcome to the %s fruit shop' % (name)

    def getCostPerKg(self, fruit):
        """
            fruit: Fruit string
        Returns cost of 'fruit', assuming 'fruit'
        is in our inventory or None otherwise
        """
        if fruit not in self.fruitPrices:
            print "Sorry we don't have %s" % (fruit)
            return None
        return self.fruitPrices[fruit]

    def getPriceOfOrder(self, orderList):
        """
            orderList: List of (fruit, numKg) tuples
            Returns cost of orderList. If any of the fruit are
        """
```

```
        totalCost = 0.0
        for fruit, numKg in orderList:
            costPerKg = self.getCostPerKg(fruit)
            if costPerPound != None:
                totalCost += numKg * costPerKg
        return totalCost

    def getName(self):
        return self.name
```

The `FruitShop` class has some data, the name of the shop and the prices per pound of some fruit, and it provides functions, or methods, on this data. What advantage is there to wrapping this data in a class?

1. Encapsulating the data prevents it from being altered or used inappropriately,
2. The abstraction that objects provide make it easier to write general-purpose code.

Using Objects

We can create and use `FruitShop` objects as follows:

```
fruitPrices = {'apples': 100, 'oranges': 150, 'pears': 175}
fruitShop = FruitShop('Fruit Bowl', fruitPrices)
applePrice = fruitShop.getCostPerKg('apples')
print applePrice
```

Exercise: Write a class Queue that supports FIFO operations.

Exercise: Write a class Stack that supports LIFO operations.

Note: Use the data type collections.deque which supports thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Static vs Instance Variables

The following example illustrates how to use static and instance variables in Python.

Create the `person_class.py` containing the following code:

```
class Person:
    population = 0
    def __init__(self, myAge):
        self.age = myAge
        Person.population += 1
    def get_population(self):
        return Person.population
    def get_age(self):
        return self.age
```

```
>>> p1 = person_class.Person(12)
>>> p1.get_population()
1
>>> p2 = person_class.Person(63)
>>> p1.get_population()
2
>>> p2.get_population()
2
>>> p1.get_age()
12
>>> p2.get_age()
63
```

In the code above, `age` is an instance variable and `population` is a static variable. `population` is shared by all instances of the `Person` class whereas each instance has its own `age` variable.

Priority queues

In Python, one can use the "*heapq*" module which has the following functions

1. heapify(iterable) :- This function is used to convert the iterable into a heap data structure. i.e. in heap order.

2. heappush(heap, ele) :- This function is used to insert the element mentioned in its arguments into heap. The order is adjusted, so as heap structure is maintained.

3. heappop(heap) :- This function is used to remove and return the smallest element from heap. The order is adjusted, so as heap structure is maintained.

```
import heapq

li = [5, 7, 9, 1, 3]

heapq.heapify(li)

print "The created heap is : ", li

heapq.heappush(li,4)
print "The modified heap after push is : ", li

print "The popped and smallest element is : ", heapq.heappop(li)
```

4. heappushpop(heap, ele) :- This function combines the functioning of both push and pop operations in one statement, increasing efficiency. Heap order is maintained after this operation.

5. heapreplace(heap, ele) :- This function also inserts and pops element in one statement, but it is different from above function. In this, element is first popped, then element is pushed.i.e, the value larger than the pushed value can be returned.

```
import heapq

li1 = [5, 7, 9, 4, 3]
li2 = [5, 7, 9, 4, 3]

heapq.heapify(li1)
heapq.heapify(li2)

print "The popped item using heappushpop() is : ",
heappq.heappushpop(li1, 2)

print "The popped item using heapreplace() is : ",
heapq.heapreplace(li2, 2)
```

6. nlargest(k, iterable, key = fun) :- This function is used to return the k largest elements from the iterable specified and satisfying the key if mentioned.

7. nsmallest(k, iterable, key = fun) :- This function is used to return the k smallest elements from the iterable specified and satisfying the key if mentioned.

```
import heapq

li1 = [6, 7, 9, 4, 3, 5, 8, 10, 1]

heapq.heapify(li1)

print "The 3 largest numbers in list are : ", heapq.nlargest(3, li1)
print("The 3 smallest numbers in list are : ", heapq.nsmallest(3,li1)
```

Exercise: Experiment with insert, pop and print operations a priority queue that contains tuples.

Exercise: Explore the package numpy.random. Refer Sec. 4.24 in the NumPy reference manual.

Exercise: Read the practice the PyPlot and Image tutorial available on Moodle.

Reading and writing files in Python

Creating a file ***file_object = open("filename", "mode")*** where ***file_object*** is the variable to add the file object. The modes are:

- '***r***' – Read mode which is used when the file is only being read
- '***w***' – Write mode which is used to edit and write new information to the file (any existing files with the same name will be erased when this mode is activated)

- '***a***' – Appending mode, which is used to add new data to the end of the file; that is new information is automatically amended to the end
- '***r+***' – Special read and write mode, which is used to handle both actions when working with a file

```
file = open("testfile.txt", "w")
file.write("Hello World")
file.write("This is our new text file")
file.write("and this is another line.")
file.write("Why? Because we can.")
file.close()
```

Reading a file

```
file = open("testfile.txt", "r")
# Will read the full file
print file.read()

file = open("testfile.txt", "r")
# Will read the first 5 characters
print file.read(5)

file = open("testfile.txt", "r")
# Will read file and return a list where each element in the list is
a line # of th file
data = file.readlines()

for line in data:
     words = line.split()
     print words
```

Reading and writing in binary to file

```
#pickle module implements a fundamental, but powerful algorithm for
serializing and de-serializing a Python object structure
import pickle


class MyClass:
    def __init__(self,name):
        self.name = name
    def display(self):
        print(self.name)
```

```
my = MyClass("someone")
pickle.dump(my, open("myobject", "wb"))
me = pickle.load(open("myobject", "rb"))
me.display()
```

Parsing binary files to read labels

Before running this code, download the file "train-labels-idx1-ubyte" from Moodle. The format of this file can be found at http://yann.lecun.com/exdb/mnist/

```
import numpy as np
import struct

numLabels = 5

fLabel = open('train-labels-idx1-ubyte', 'rb')
fLabel.read(8)

bufLabel = fLabel.read(numImages)
labelData = np.frombuffer(bufLabel, dtype=np.uint8)
print(labelData)
```

Parsing binary files to display images

Before running this code, download the file "train-images-idx3-ubyte" from Moodle. The format of this file can be found at http://yann.lecun.com/exdb/mnist/

```
%matplotlib inline
import numpy as np
import struct
import matplotlib.pyplot as plt

fImg = open('train-images-idx3-ubyte','rb')

fImg.read(8)
numImages = 5
numRows = struct.unpack(">i",fImg.read(4))[0]
numCols = struct.unpack(">i",fImg.read(4))[0]

bufImg = fImg.read(numCols * numRows * numImages)
imgData = np.frombuffer(bufImg, dtype=np.uint8)
data = data.reshape(numImages, numRows, numCols)
plt.imshow(data[4], cmap="gray")
plt.show()
```