STAT 534

Lecture 1

Python Basics

4 Apr 2019

mmp@stat.washington.edu
Scribes: Xiang Meng, Rong Fan

1 Some Data Structures in Python

All examples in `pythondemo.py`.

- List

```
l=[1,2,3]
l4 = l.append(4) # add item
print(l4)
l4 = [1,2,3,4]
```

Let l be a Python list.

  - l.append(x) - Modifies list by adding an element x.
  - l.copy() - Returns a new copy of l.
  - l.reverse() - Modifies l by reversing it.
  - l.insert(i, x) - Inserts element x at index i.
  - l.pop() - Removes the last element of l and returns it.
  - l.pop(i) - Removes the element at index i in l and returns it.
  - l.remove(x) - Modifies list by removing the first occurence of the element x.

Note: The item to be removed must be in the list.

Remark: the definition of list from the perspective of abstract data structures (coming next lecture) is different from the list *implementation* in Python. It is the place to say that one must distinguish between **(linked) list**, an abstract data structure described e.g. in CRLS, and `list` a type of **object** in python. Note that in python and most modern languages, everything but "simple" numbers, characters, and truth values `int, float, bool, char` is an object. In particular, `list, dict, numpy.ndarray` are objects; these objects **implement** data structures, that is they manage data "cptainers" (or collections) and provide functions to handle these data.

- **Deque** is another data structure in python, that implements the abstract concept of linked list.

```
from collections import deque
q = deque([]) # initialization of a deque
q.append('first') # add 'first' to q
q.append('second')
q.append('third')
print(q)
q = deque(['first', 'second', 'third'])
q.pop() # remove the last item from q
q.popleft() # remove item from the beginning of q
```

  Deque, like list and many other data structures, does not distinguish between the contents its items, so when you append 'second' twice, there will be two 'second' elements in the deque.

- Numpy array

```
import numpy as np
X = np.array() # initialization of a numpy array
X = np.array([[0,1,2,3], [4,5,6, 7], [8,9,10, 11] ])
X.shape # tuple containg the shape
X.ndim # number of dimensions
X.size # number of elements
```

  where *np* is a shortcut/nickname for the full name of the library. When you write your code, the data structure *numpy* is recommended and it is better if you use one type of data structure consistently.

```
X1 = X.flat[:]  # write numpy array X into a 1 dimentional vector
X1 = array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
X[1,1:3] += 2 # add 2 to the '1' row, '1 ~ 2' column
X = array([[ 0, 1, 2, 3], [ 4, 7, 8, 7], [ 8, 9, 10, 11]])
X>5 # generate a numpy array of boolean type of value
array([[False, False, False, False], [False, True, True, True],
[ True, True, True, True]])
X[X>5]=0 # zero out the items where X>5 is true
array([[0, 1, 2, 3], [4, 0, 0, 0], [0, 0, 0, 0]])
```

2 Data Structure and Memory Management

2.1 The Random Access Memory (RAM)

RAM is the "main memory" of a computer; in the context of software and programming, memory refers to the RAM. The RAM can be seen (in a very simplified way) as a very long vector of **words**, numbered sequentially from 0 to $2^M - 1$ where M is an integer, e.g 64 or larger. A word is the smallest **adressable**

unit. A **memory address** is an integer between 0 and $2^M - 1$ which identifies a word; it is also called a **pointer** (C), **reference** (Java, python).

In the assignment `variable = expression`, the left hand side (l.h.s) represents one or more memory locations (e.g the variable `x`, or the range `X[:10:2]`). The r.h.s is an expression, something to be computed, and whose result, once obtained, is written in the locations described by l.h.s.

Every variable in program is **allocated** memory, enough to store this variable. Note that an integer may take less space than a string, and a list of integers more than a single one. Tables stored by the python **interpreter** keep track of the memory allocation, so that the location of a variable is not overwritten by another (unless we specifically want it so), or by code, etc.

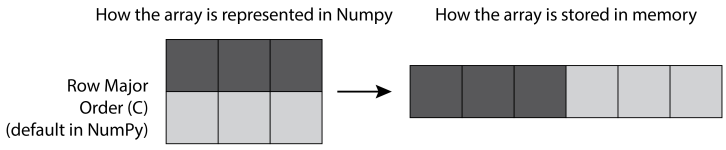
The RAM

- every word can be accessed in equal time
- every word can store any type of "thing": variables, pointers, executable instructions, garbage; they are all stored as 0,1 numbers.
- allocating memory is much more expensive than reading/writing memory
- memory must be allocated/deallocated during the run of a program (e.g. when we read data from a file, space to store it in the RAM is allocated). Sometimes, variables must be moved around to make (contiguous) space
- therefore, to save time, various strategies are employed (transparently to the user): typically allocate more space than required to have a reserve is the e.g list grows; allocate block of size 2^k that fit easier together than blocks of arbitrary size. It is easier for the binary-based computer to deal with powers of 2, and it gives room for memory growth. If the variable grows beyond the pre-assigned memory, double allocation will happen. Thus, you should be careful when allocation is executed a lot in your program.
- in python, the program that manages the RAM is called **garbage collector**

2.2 Data Structures in python

Basic data types (eg. integer, boolean) have their values stored directly in Random Access Memory, or RAM. Objects, such as array-like types (eg. tuple, list) have a block of data that describes the object/data structure (e.g the length of the string, type of elements of the array); this block contains the address where the actual data values are being stored.

For example, a numpy array is stored using row-major order A NumPy array is basically described by metadata (notably the number of dimensions, the shape, and the data type) and the actual data. The data is stored in a homogeneous and contiguous block of memory, at a particular address in system memory (RAM). This block of memory is called the data buffer. This is the



main difference between an array and a pure Python structure, such as a deque, where the items are scattered across the RAM. This aspect is the critical feature that makes NumPy arrays so efficient. [1]

Thus when you set `A=X`, the metadata of `A` and `X` would be the same, pointing to the same data memory at the same time. When you change the items in `A`, `X` will also be changed. If you want to copy the metadata as well as the data of `X`, you can use `X3 = X.copy()`, which assign a new metadata to `X3`. In this case, when you change the values in `X3`, `X` will not be affected.

Some python/numpy operations on arrays, lists etc, are executed **in place**, that is they modify the data structure. Others first make a **defensive copy**, i.e. duplicate the data, and modify the copy, leaving the original untouched. In python it is not immediate to see which type of operation is performed by a given function (often, the option `copy` lets you be "defensive"); make sure you check this!

[1] https://ipython-books.github.io/45-understanding-the-internals-of-numpy-to-avoid-unnecessary-array-copying/