Programming Lab II
Handout 3

Antonio de la Vega de León, Bijun Zhang, Thomas Blaschke, Dr. Martin Vogt
`martin.vogt@bit.uni-bonn.de`

May 3, 2016

Assignments for classes on May 3 and May 10

Python has three (four) popular builtin data structures for holding collections of arbitrary elements/items. These are:

1. lists
2. dictionaries (and sets)
3. tuples

(Strictly speaking, `string`s are also collections of elements similar to lists. However, here the elements have to be characters and strings are so ubiquitous and prominent that they deserve special treatment.)

Lists are the most popular and most basic of these. Conceptually, they most closely resemble Java's `ArrayList<Object>`. However, in contrast to Java there is a special syntax for lists. E.g.

- literal lists: `[1,2,3]` or `[['a','list'],'containing',['other','lists'],'as','elements']`
- indexing lists: The first, fifth, and last element of a list named `adam` can be accessed with `adam[0]`, `adam[4]`, and `adam[-1]`, respectively.
- list manipulation: operator `+` concatenates list, i.e. `[1,2,3]+['four','five','six']` yields `[1,2,3,'four','five','six']`.

Dictionaries are a very flexible data structure represented by key–value pairs and have proven to be a very powerful data structure for scripting. Interestingly, different programming languages use different names for the dictionary data structure. In Java they are called maps, in Perl they are called hashes, and in Awk (a popular language focussing on text processing and a precursor of Perl) they are called associative arrays. In dictionaries, each key item is associated unambiguously with a single value item. For instance, letters representing amino acids might be the keys and the frequency of their occurrence in a specific protein might be the corresponding values of a dictionary named `aminoAcidFrequency`:

- literals: `aminoAcidFrequency={'A':13,'R':27,'N': 22}` or `aminoAcidName={'A':'alanine','R':'arginine','N': 'asparagine'}`
- indexing: the letter `R` represents the amino acid `aminoAcidName['R']` and it occurs `aminoAcidFrequency['R']` times in a given amino acid sequence.
- manipulation: `aminoAcidName['D']='aspartic acid'` adds a new key–value pair to the dictionary and `aminoAcidFrequency['N']=aminoAcidFrequency['N']+1` increments the frequency of asparagine by 1.

In dictionaries the keys have to be unique. That is it is not possible to have two key–value pairs with identical keys. I.e., if you type `{'A':1,'B':2,'C':2,'A':4}` they resulting dictionary will still be valid but it will only contain the pair `'A':4` and the two pairs `'B':2` and `'C':2`, which have the same value but different keys. *Sets* are like dictionaries, only sets contain only keys and have no associated values. E.g. `aromatic = {'Phe','Trp','Tyr'}` is the set of aromatic amino acids.

Finally, *tuples* are a kind of 'fixed list'. They can be seen as a list of elements that cannot change. While lists can change by adding/removing/modifying elements, a tuple will never change its value once it has been given initialized. In this regard, tuples are similar to strings. Frequently, tuples are of short length of 5 or less and are often used literally in code. For instance, a function that determines the most frequent nucleotide of a DNA sequence can return not only the nucleotide but also the frequency using a tuple. I.e.

```
def mostFrequent(sequence):
    # ... some code magic
    return (nucleotide,frequency)

seq= ...
n,f = mostFrequent(seq)
print("The most frequent nucleotide is",n,"and occurs",f,"times.")
```

Exercises

Exercises marked (*) are slightly more challenging and you can consider them optional if you find them too hard.

Ex.0 *Reading*

The material in this handout is covered in chapters 10 to 13 of the book *Think Python*.

Ex.1 *(ex. 10.3 & 10.4 of Think Python)*

(a) Write a function called `middle` that takes a list and returns a new list that contains all but the first and last elements. So `middle([1,2,3,4])` should return `[2,3]`.

(b) Write a function called `chop` that takes a list, modifies it by removing the first and last elements, and returns `None`.

(c) Try the following:

```
>>> listA = [1,2,3,4]
>>> listB = middle(listA)
>>> print(listA, listB) # What is the value of listA and listB
>>> listC = [1,2,3,4]
>>> listD = listC
>>> chop(listD)
>>> print(listC, listD) # What is the value of listC and listD?
>>> if listC == [2,3]:
        print("I am surprised.")
```

If you are surprised study sections 10.10 – 10.12 (p. 95–98) again of Think Python and draw state diagrams for the example above. Understanding the concepts of *objects*, *values*, and *references* (or *aliasing* as it is termed in *Think Python*) is very important. Java in this regard is very similar to Python and understanding the distinction between object and value and the *reference semantics* of Java and Python is vital.

In contrast to Java and Python, R uses *value semantics*. This means that the values of objects will be *copied* to new objects during assignments and while passing arguments to function parameters. To illustrate the difference, consider the following example in Python and R.

Python:

```
>>> a = [1,2,3]
>>> b = a
>>> a[1] = -42
>>> print(a)
[1, -42, 3]
>>> print(b)
[1, -42, 3]
```

R:

```
> a <- c(1,2,3)
> b <- a
> a[2] <- -42
> print(a)
[1]   1 -42   3
> print(b)
[1] 1 2 3
```

Although the R example is a line by line translation of the Python code the result for `b` is different. This is because of the different semantic of `b = a` in Python and `b <- a` in R. In Python only a reference to the object is copied while in R the value is copied to a new object.

In Python and Java, strings are so-called *immutable* objects. This means that once an object received a value it will never change. In this special case an object can indeed be identified with its value and it does not matter whether value or reference semantics is used.

Ex.2 *Sum of all depths (Ex 10.1 & 10.2 of Think Python)*

(a) Write a function that takes a list of numbers and returns the cumulative sum; that is, a new list where the ith element is the sum of the first $i + 1$ elements from the original list. For example, the cumulative sum of `[1, 2, 3]` is `[1, 3, 6]`.

(b) Write a function called `nested_sum` that takes a nested list of integers and adds up the elements from all of the nested lists.

(c) (*) What about a list like `[1,[2,3,[4,5]],[[6,7],8,[9,[10]]]]` that contains numbers nested in lists of arbitrary depth. Write a function `depth_sum` that takes a list containing integers nested in lists of arbitrary depth and returns the sum of all the numbers. *Hint:* Use recursion.

Ex.3 *Reverse pairs (Ex. 10.9 & 10.11 & 11.1 of Think Python)*

(a) Write a function that reads the file `words.txt` and builds a list with one element per word. Write two versions of this function, one using the `append` method and the other using the idiom `t = t + [x]`. Which one takes longer to run? Why?

(b) Two words are a “reverse pair” if each is the reverse of the other. Write a program that finds all the reverse pairs in the word list `words.txt`. To this end, first read in all the words of `words.txt` in a (sorted) list. You can test whether a word is already in the list using the `in` operator. However, you can also use the `bisect` method of ex. 10.10 of *Think Python* or use the `bisect` module, which is part of the Python standard library.

(c) Modify the code of the previous exercise and read in all the words as keys of a dictionary (or a set) with arbitrary value (e.g. set the value to `None`) in each case. In this case use the `in` operator because `bisect` will only work on lists, and it will only work correctly, if the lists are sorted.

As an alternative Python provides the data structure `set`. You can think of a set simply as a dictionary with only the keys and without the values. If you have a set of items checking whether an element is contained in the set is particularly efficient.

Another thing to keep in mind is that a set cannot contain multiple identical elements. I.e. while a list might contain multiple identical elements in a set there can only be one copy at most. Finally, elements of sets and keys of dictionaries have to be *immutable*. That is, lists cannot be elements of sets or keys of a dictionary but tuples can.
In simple terms, the reason is that if you were to put a list in a set, and then modify it, it messes up the way elements are internally stored resulting in wonderful, hard to track bugs in your code. Python offers some protection against this happening. s For maps/sets in Java you do not have this kind of protection and you need to take the proper precaution yourself.

Ex.4 (*) *Anagrams (Ex.12.2 of Think Python)*

(a) Write a program that reads a word list from the file `words.txt` and prints all the sets of words that are anagrams.
Here is an example of what the output might look like:

```
['deltas', 'desalt', 'lasted', 'salted', 'slated', 'staled']
['retainers', 'ternaries']
['generating', 'greatening']
['resmelts', 'smelters', 'termless']
```

Hint: you might want to build a dictionary that maps from a set of letters to a list of words that can be spelled with those letters. The question is, how can you represent the set of letters in a way that can be used as a key?

(b) Modify the previous program so that it prints the largest set of anagrams first, followed by the second largest set, and so on.

(c) In Scrabble a "bingo" is when you play all seven tiles in your rack, along with a letter on the board, to form an eight-letter word. What set of 8 letters forms the most possible bingos? Hint: there are seven.

Ex.5 *Histograms (Section 11.2 of Think Python)*

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] += 1
    return d
```

Dictionaries have a method called `get` that takes a key and a default value. If the key appears in the dictionary, `get` returns the corresponding value; otherwise it returns the default value. For example:

```
>>> h = histogram('a')
>>> print(h)
{'a': 1}
>>> h.get('a', 0)
1
>>> h.get('b', 0)
0
```

Use `get` to write `histogram` more concisely. You should be able to eliminate the `if` statement. (If you are more adventurous checkout the `defaultdict` class in the Python module `collections`.)

(a) Take a look at the file `ecoli-genome.fna` found in `\\bitsmb\groups\workshops\proglab2\` containing the DNA of a strain of escherichia coli.
Write a function `fasta_frequency(filename)` that takes a filename as a parameter and returns a histogram containing the frequency of each nucleotide (or amino acid).

(b) Write a function `print_hist(h)` that takes a histogram as parameter and prints a table of keys and values in alphabetical order of the keys.

(c) Combine the functions into a script where the user is asked to input a filename of a file in fasta format that calculates the frequencies and prints a table of nucleotides/amino acid symbols and their respective frequency.

Ex.6 *Sorting a histogram & a slightly useful first script*
File `ecoli-proteome.faa` in `\\bitsmb\groups\workshops\proglab2\` lists the amino-acid sequences of escherichia coli in fasta format.

(a) Make sure that the function `fasta_frequency(filename)` from the previous exercise also works with multiple fasta sequences in one file. A single histogram should be returned combining the frequencies of all sequences. Calculate the histogram for the amino acids for all proteins combined.

(b) Now write a function that sorts the amino-acids by their *relative* frequency and print a table of amino acids and their *relative* frequency in decreasing order of frequency. *Hint:* The entries of a dictionary can be converted to a list of key–value tuples using the method `items()`. See section 12.6 of *Think Python*.

(c) In order to make this more useful combine the functions in a single script `fasta_frequency.py` that asks the user for a filename and as a results prints the occurring amino-acids/nucleotides in decreasing order of frequency.

(d) Frequently, scripts work in a non-interactive way. That is, they do not expect the user to enter anything while the program is running. Instead, all the information the program needs is passed to the program before the start using command line parameters. The program can then called from the command line:

```
W:\> python fasta_frequency.py sequence.fasta
```

or from the notebook

```
In [ ]: %run fasta_frequency.py sequence.fasta
```

Here, `sequence.fasta` is the filename containing the data you want to investigate. The parameters entered after the script name is available to the python script itself. It is stored in a list of strings that can be found in the module `sys`. E.g., consider a script named `args.py`

```
import sys
print(sys.argv) # argv is short for argument vector
```

Then running the script:

```
In [ ]: %run args Was it a car or a cat I saw
```

will print

```
['args.py','Was','it','a','car','or','a','cat','I','saw']
```

Modify the script so that it reads the filename from the command line (to be found in `sys.argv[1]`).

(e) Apply your script to `ecoli-proteome.faa` and to one or more of the cromosomal proteomes of drosophila melangoster: `drosophila-proteome-chrX.faa`, `drosophila-proteome-chr2.faa`, `drosophila-proteome-chr3.faa`.
Are there noticeable differences in the amino acid composition between escherichia coli and drosophila melangoster?
How well do the frequencies correspond to the published results found in the publication `amino-acid-composition.pdf` (Fig. 1, p. 601):

Bogatyreva, N.; Finkelstein, A.V.; Galzitskaya, O.V. Trend of amino acid composition of proteins of different taxa. *Journal of Bioinformatics and Computational Biology* **4** (2006), 597–608.

Ex.7 *(*) Optional:*
A nice word puzzle and an even better programming exercise (Ex. 12.4 of Think Python)
Here's another Car Talk Puzzler (`http://www.cartalk.com/content/puzzlers`):

> What is the longest English word, that remains a valid English word, as you remove its letters one at a time?
>
> Now, letters can be removed from either end, or the middle, but you can't rearrange any of the letters. Every time you drop a letter, you wind up with another English word. If you do that, you're eventually going to wind up with one letter and that too is going to be an English word—one that's found in the dictionary. I want to know what's the longest word and how many letters does it have?
>
> I'm going to give you a little modest example: Sprite. Ok? You start off with sprite, you take a letter off, one from the interior of the word, take the r away, and we're left with the word spite, then we take the e off the end, we're left with spit, we take the s off, we're left with pit, it, and I.

Write a program to find all words that can be reduced in this way, and then find the longest one.

This exercise is a little more challenging than most and a few hints are given for solving this problem are given in *Think Python.* When trying this exercise please note that the provided list of words `words.txt` does not contain the one letter words 'a' and 'i'.