Python

Agenda - 3

- Sorting
- Sequence Types
  - Tuples
- Mapping Types
  - Dict
- Del Keyword
- **Lab-5**

Sorting

- The **sorted**(list) function takes a list and returns a new list with those elements in sorted order. The **original list** is **not changed**.

```
a = [5, 1, 4, 3]
print sorted(a)  ## [1, 3, 4, 5]
print a  ## [5, 1, 4, 3]
```

- It's most common to pass a list into the sorted() function, but in fact it can take as input any sort of **iterable** collection.

- **An iterable:** Is an object capable of returning its members one at a time. Examples of iterables include all **sequence** types (such as **list**, **str**, and **tuple**) and some non-sequence types like **dict** and **file**. Iterables can be used in a **for** loop and in many other places where a sequence is needed (**zip**(), **map**(), ...).

Sorting

- The sorted() function can be customized though optional arguments. The sorted() optional argument **reverse**=**True**, e.g. sorted(list, reverse=True), makes it sort backwards.

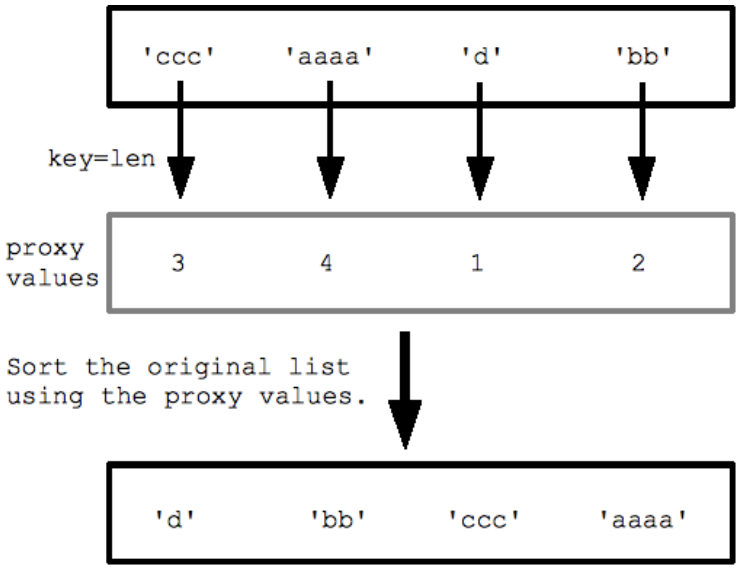
```
strs = ['aa', 'BB', 'zz', 'CC']
print sorted(strs)  ## ['BB', 'CC', 'aa', 'zz'] (case sensitive)
print sorted(strs, reverse=True)   ## ['zz', 'aa', 'CC', 'BB']
```

- For more complex custom sorting, sorted() takes an optional "**key=**" specifying a "key" **function** that transforms each element before comparison. The key function **takes** in **1 value** and **returns 1 value**, and the returned "proxy" value is used for the comparisons within the sort.

Sorting

- For example with a list of strings, specifying **key**=**len** (the built in len() function) sorts the strings by length, from **shortest** to **longest**. The sort calls len() for each string to get the list of proxy length values, and the sorts with those **proxy** values.

```
strs = ['ccc', 'aaaa', 'd', 'bb']
print sorted(strs, key=len)  ## ['d', 'bb', 'ccc', 'aaaa']
```



Sorting

- You can also pass in your **own function** as the key function, like this:

```
## Say we have a list of strings we want to sort by the last letter of the string.
strs = ['xc', 'zb', 'yd' ,'wa']

## Write a little function that takes a string, and returns its last letter.
## This will be the key function (takes in 1 value, returns 1 value).
def MyFn(s):
  return s[-1]

## Now pass key=MyFn to sorted() to sort by the last letter:
print sorted(strs, key=MyFn)  ## ['wa', 'zb', 'xc', 'yd']
```

Sequence Types
Tuples

- A **tuple** is a fixed size grouping of elements. Tuples are like lists, except they are **immutable** and do **not change size**.

- To **create** a tuple, just list the values within **parenthesis** separated by **commas**. The "**empty**" tuple is just an **empty pair** of parenthesis. Accessing the elements in a tuple is just like a list -- len(), [ ], for, in, etc. all work the same.

```
tuple = (1, 2, 'hi')
print len(tuple)  ## 3
print tuple[2]    ## hi
tuple[2] = 'bye'  ## NO, tuples cannot be changed
tuple = (1, 2, 'bye')  ## this works
```

- To create a size **1** tuple, the lone element must be followed by a **comma**.

```
tuple = ('hi',)   ## size-1 tuple
```

Sequence Types
Tuples

- Assigning a tuple to an identically sized tuple of variable names assigns all the corresponding values. If the tuples are **not** the **same size**, it throws an **error**. This feature works for **lists** too.

```
(x, y, z) = (42, 13, "hike")
print z  ## hike
(err_string, err_code) = Foo()  ## Foo() returns a length-2 tuple
```

Sequence Types
Tuples

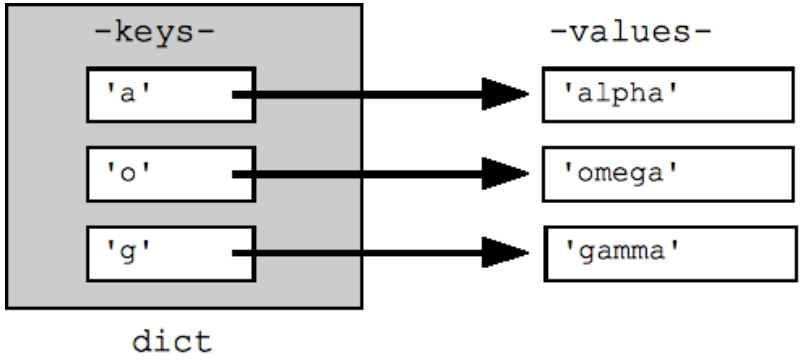
- **Strategy note:** Tuples play a sort of "**struct**" role in Python -- a convenient way to pass around a logical, fixed size bundle of values.
  - For example, if I wanted to have a **list** of **3-d coordinates**, the natural python representation would be a **list** of **tuples**, where each tuple is size 3 holding one **(x, y, z)** group.
  - Another example for the usage of tuples, is when defining non changing information

```
days = ('Sunday','Monday','Tuesday','Wensday','Thursday','Friday','Saturady')
```

Mapping Types Dictionaries

- Python's efficient **key/value** hash table structure is called a "**dict**". The contents of a dict can be written as a series of **key:value** pairs within braces { }, e.g. dict = {key1:value1, key2:value2, ... }. The "**empty dict**" is just an **empty** pair of curly braces **{}**.

- **Strings**, **numbers**, and **tuples** work as **keys**, and any type can be a **value**. Other types may or may not work correctly as keys (**strings** and **tuples** work cleanly since they are **immutable**).



Mapping Types Dictionaries

- Looking up or setting a value in a dict uses square brackets, e.g. **mydict**['**foo**'] looks up the value under the key 'foo'.

```
## Can build up a dict by starting with the the empty mydict {}
## and storing key/value pairs into the mydict like this:
## mydict[key] = value-for-that-key
mydict = {}
mydict['a'] = 'alpha'
mydict['g'] = 'gamma'
mydict['o'] = 'omega'

print mydict  ## {'a': 'alpha', 'o': 'omega', 'g': 'gamma'}
print mydict['a']     ## Simple lookup, returns 'alpha'
mydict['a'] = 6       ## Put new key/value into dict
'a' in mydict         ## True
## print mydict['z']                  ## Throws KeyError
if 'z' in mydict: print mydict['z']     ## Avoid KeyError
print mydict.get('z')  ## None (instead of KeyError)
```

Mapping Types Dictionaries

- The methods mydict.**keys**() and mydict.**values**() return lists of the keys or values explicitly and mydict.**items**() return both.

```
## Get the .keys() list:
print mydict.keys()  ## ['a', 'o', 'g']

## Likewise, there's a .values() list of values
print mydict.values()  ## ['alpha', 'omega', 'gamma']

## .items() is the dict expressed as a list of (key, value) tuples
print mydict.items()  ##  [('a', 'alpha'), ('o', 'omega'), ('g',
'gamma')]
```

Mapping Types Dictionaries

- A for loop on a dictionary iterates over its **keys** by **default**. The keys will appear in an **arbitrary** order.

```
for key in mydict: print key
## prints a g o

## Exactly the same as above
for key in mydict.keys(): print key

## Common case -- loop over the keys in sorted order,
## accessing each key/value
for key in sorted(mydict.keys()):
print key, mydict[key]

## This loop syntax accesses the whole dict by looping
## over the .items() tuple list, accessing one (key, value)
## pair on each iteration.
for k, v in mydict.items(): print k, '>', v
## a > alpha    o > omega     g > gamma
```

Mapping Types
Dictionaries

- The % operator works conveniently to substitute values from a dict into a string by name:

```
hash = {}
hash['word'] = 'garfield'
hash['count'] = 42
s = 'I want %(count) copies of %(word)s' % hash
# 'I want 42 copies of garfield'
```

Mapping Types Dictionaries

- **Strategy note:** from a performance point of view, the dictionary is one of your greatest tools, and you should use where you can as an easy way to **organize data**.
  - For example, you might read a log file where each line begins with an **ip address**, and store the data into a dict using the ip address as the **key**, and the **list of lines** where it appears as the **value**. Once you've read in the whole file, you can **look up** any ip address and instantly see its list of lines. The dictionary takes in **scattered data** and make it into something **coherent**.

  - For example, if you have a **phone book** application and you want to read the contacts from a file then a dict could be used to store entries of **names** and **numbers**. The name would be the **key** and the **number** would be the value.

Del Keyword

- The "**del**" operator does **deletions**. In the simplest case, it can remove the definition of a **variable**, as if that variable had not been defined. Del can also be used on **list** elements or slices to delete that part of the list and to delete entries from a **dictionary**.

```
var = 6
del var  # var no more!

list = ['a', 'b', 'c', 'd']
del list[0]     ## Delete first element
del list[-2:]   ## Delete last two elements
print list      ## ['b']

dict = {'a':1, 'b':2, 'c':3}
del dict['b']   ## Delete 'b' entry
print dict      ## {'a':1, 'c':3}
```

LAB – 5
TUPLES LAB

Tuples Lab

- Please download the lab from the following link:
  - https://drive.google.com/file/d/0B6Hf8UvSSqTXT0cyaWZHNHByVnM/view?usp=sharing
- Complete the script **tuples_lab.py** in **45** mins and send your solution on the following email:
  - Omar.Soliman@imtSchool.com with the following subject :
    - If you are from ITI-Smart track ES:
      - [ITI_SV_39][PY-tuples]yourfullname
    - If you are from ITI-Ismailia track ES:
      - [ITI_NC_39][PY-tuples]yourfullname

LAB – 5
TUPLES LAB

What's Next ?

- Get Certified With:
  - https://www.edx.org/course/learn-program-using-python-utarlingtonx-cse1309x
  - https://www.coursera.org/course/interactivepython1
  - https://www.coursera.org/course/interactivepython2

- More Interesting References:
  - Python Cookbook, 2nd Edition
  - https://automatetheboringstuff.com/
  - http://code.activestate.com/recipes/langs/

Eng. Mohammad A.Hekal: embeddedgeek.34@gmail.com
Omar Soliman: Omar.Soliman@imtSchool.com