Python: Data Structures

FOSSEE

1 Basic Looping

1.1 `while`

```
In []: a, b = 0, 1
In []: while b < 10:
  ...:     print b,
  ...:     a, b = b, a + b # Fibonacci Sequence
  ...:
```

Basic syntax of **`while`** loop is:

```
while condition:
    statement1
    statement2
```

All statements are executed, till the condition statement evaluates to True.

1.2 `for` and `range`

`range(start, stop, step)`
returns a list containing an arithmetic progression of integers.
Of the arguments mentioned above, both start and step are optional.
For example, if we skip third argument, i.e `step`, default is taken as 1. So:

```
In []: range(1,10)
Out[]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note: stop value is not included in the list.
Similarly if we don't pass `first` argument (in this case `start`), default is taken to be 0.

```
In []: range(10)
Out[]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In case third argument is mentioned(`step`), the jump between consecutive members of the list would be equal to that.

```
In []: range(1,10,2)
Out[]: [1, 3, 5, 7, 9]
```

`for` and `range`

As mentioned previously **`for`** in Python is used to iterate through the list members. So **`for`** and `range` can be used together to iterate through required series. For example to get square of all numbers less then 5 and greater then equal to 0, code can be written as:

```
In []: for i in range(5):
 ....:     print i, i * i
 ....:
 ....:
0 0
1 1
2 4
3 9
4 16
```

2 list

```
In []: num = [1, 2, 3, 4] # Initializing a list
In []: num
Out[]: [1, 2, 3, 4]
```

2.1 Accessing individual elements

```
In []: num[1]
Out[]: 2
```

Note: Index of list starts from 0.

```
In []: num[5]  # ERROR: throws a index error
IndexError: list index out of range
In []: num[-1]
Out[]: 4
```

Note: `-1` points to last element in a list. Similarly to access third last element of a list one can use:

```
In []: num[-3]
Out[]: 2
```

2.2 `list` operations

```
In []: num += [9, 10, 11] # Concatenating two lists
In []: num
Out[]: [1, 2, 3, 4, 9, 10, 11]
```

`list` provides `append` function to append objects at the end.

```
In []: num = [1, 2, 3, 4]
In []: num.append(-2)
In []: num
Out[]: [1, 2, 3, 4, -2]
```

Working of `append` is different from + operator on list. Till here both will behave as same. But in following case:

```
In []: num = [1, 2, 3, 4]

In []: num + [9, 10, 11]
Out[]: [1, 2, 3, 4, 9, 10, 11]

In []: num.append([9, 10, 11]) # appending a list to a list

In []: num
Out[]: [1, 2, 3, 4, [9, 10, 11]] # last element is a list
```

when one attempts to append a list(in above case [9, 10, 11]) to a list(num) it adds list as a single element. So the resulting list will have a element which itself is a list. But + operator would simply add the elements of second list.

2.3 Miscellaneous

```
In []: num = [1, 2, 3, 4]
In []: num.extend([5, 6, 7])  # extend list by adding elements
In []: num
Out[]: [1, 2, 3, 4, 5, 6, 7]
In []: num.reverse()  # reverse the current list
In []: num
Out[]: [7, 6, 5, 4, 3, 2, 1]
In []: num.remove(6) # removing first occurrence of 6
In []: num
Out[]: [7, 5, 4, 3, 2, 1]
In []: len(num) # returns the length of list
Out[]: 6
In []: a = [1, 5, 3, 7, -2, 4]
In []: min(a) # returns smallest item in a list.
Out[]: -2
In []: max(a) # returns largest item in a list.
Out[]: 7
```

2.4 Slicing

General syntax for getting slice out of a list is
`list[initial:final:step]`

```
In []: a = [1, 2, 3, 4, 5]
In []: a[1:-1:2]
Out[]: [2, 4]
```

Start slice from second element(1), till the last element(-1) with step size of 2.

```
In []: a[::2]
Out[]: [1, 3, 5]
```

Start from beginning(since `initial` is blank), till last(this time last element is included, as `final` is blank), with step size of 2.
Apart from using `reverse` command on list, one can also use slicing in special way to get reverse of a list.

```
In []: a[-1:-4:-1]
Out[]: [5, 4, 3]
```

Above syntax of slice can be expressed as, "start from last element(`-1`), go till fourth last element(`-4`), with step size `-1`, which implies, go in reverse direction. That is, first element would be `a[-1]`, second element would be `a[-2]` and so on and so forth."
So to get reverse of whole list one can write following slice syntax:

```
In []: a[-1::-1]
Out[]: [5, 4, 3, 2, 1]
```

Since `final` is left blank, it will traverse through whole list in reverse manner.

Note: While `reverse` reverses the original list, slicing will just result in a instance list with reverse of original, which can be used and worked upon independently.

2.5 Containership

`in` keyword is used to check for containership of any element in a given list.

```
In []: a = [2, 5, 4, 6, 9]
In []: 4 in a
Out[]: True

In []: b = 15
In []: b in a
Out[]: False
```

3 Tuples

Tuples are sequences just like Lists, but they are **immutable**, or items/elements cannot be changed in any way.

```
In []: t = (1, 2, 3, 4, 5, 6, 7, 8)
```

Note: For tuples we use parentheses in place of square brackets, rest is same as lists.

```
In []: t[0] + t[3] + t[-1] # elements are accessed via indices
Out[]: 13
In []: t[4] = 7 # ERROR: tuples are immutable
```

Note: elements cant be changed!

4 Dictionaries

Dictionaries are data structures that provide key-value mappings. They are similar to lists except that instead of the values having integer indexes, they have keys or strings as indexes.
A simple dictionary can be created by:

```
In []: player = {'Mat': 134,'Inn': 233,
          'Runs': 10823, 'Avg': 52.53}
```

For above case, value on left of ':' is key and value on right is corresponding value. To retrieve value related to key 'Avg'

```
In []: player['Avg']
Out[]: 52.530000000000001
```

4.1 Element operations

```
In []: player['Name'] = 'Rahul Dravid' #Adds new key-value pair.
In []: player
Out[]:
{'Avg': 52.530000000000001,
 'Inn': 233,
 'Mat': 134,
 'Name': 'Rahul Dravid',
 'Runs': 10823}
In []: player.pop('Mat') # removing particular key-value pair
Out[]: 134
In [21]: player
Out[21]: {'Avg': 52.530000000000001, 'Inn': 233,
          'Name': 'Rahul Dravid', 'Runs': 10823}

In []: player['Name'] = 'Dravid'
In []: player
Out[23]: {'Avg': 52.530000000000001, 'Inn': 233,
          'Name': 'Dravid', 'Runs': 10823}
```

Note: Duplicate keys are overwritten!

4.2 containership

```
In []: 'Inn' in player
Out[]: True
In []: 'Econ' in player
Out[]: False
```

Note: Containership is always checked on 'keys' of dictionary, never on 'values'.

4.3 Methods

```
In []: player.keys() # returns list of all keys
Out[]: ['Runs', 'Inn', 'Avg', 'Mat']

In []: player.values() # returns list of all values.
Out[]: [10823, 233,
        52.530000000000001, 134]
```

5 Sets

are an unordered collection of unique elements.
Creation:

```
In []: s = set([2,4,7,8,5]) # creating a basic set
In []: s
Out[]: set([2, 4, 5, 7, 8])
In []: g = set([2, 4, 5, 7, 4, 0, 5])
In []: g
Out[]: set([0, 2, 4, 5, 7]) # No repetition allowed.
```

Some other operations which can be performed on sets are:

```
In []: f = set([1,2,3,5,8])
In []: p = set([2,3,5,7])
In []: f | p # Union of two sets
Out[]: set([1, 2, 3, 5, 7, 8])
In []: f & p # Intersection of two sets
Out[]: set([2, 3, 5])
In []: f - p # Elements in f not is p
Out[]: set([1, 8])
In []: f ^ p # (f - p) | (p - f)
Out[]: set([1, 7, 8]))
In []: set([2,3]) < p # Test for subset
Out[]: True
```