Sequential Data Types

Prof.Dr. Bahadır AKTUĞ
BME362 Introduction To Python

*Compiled from sources given in the references.

Sequential Data Types

- Sequential data types are needed in programming at any scale.
- Python provides six sequential data types:
  - strings
  - byte sequence
  - byte arrays
  - list
  - tuple
  - range object
- While these data seem to be quite different at first sight, they have one important common feature: they hold data sequentially.

Sequential Data Types

- The elements of a sequential data type can be accessed with indexing.
- Remember the indexing we use to access characters in a string type variable:

```
>>> s = "Programming with Python"
>>> print(s[0], s[17])
PP
```

- Accessing the elements of a list with indexing:

```
>>> l = ["Ankara", "İstanbul", "İzmir", "Adana"]
>>> print(l[1], l[2])
İstanbul İzmir
```

Sequential Data Types

- There are also functions defined for sequential data types. In Python, such functions are common to all sequential data types (string, list, tuple etc.).
- For instance, the length of a sequential data type can retreived by using "len()" function:

```
>>> s = "Programming with Python"
>>> l = ["Ankara", "İstanbul", "İzmir", "Adana"]
>>> print(len(s),len(l))
23 4
```

Lists

- In general, "lists" can be considered similar to the arrays in C, C++, Java and Matlab.
- However, "lists" in Python are much more powerful and flexible with respect to the "arrays" in classical programming languages.
- For one thing, the elements of a "list" does not have to be of the same data type (integer, string, float etc.).
- Lists can be expanded/shrinked during runtime. In static arrays, the dimension is constant during run time.
- The lists in Python are an array of sequential objects. Those objects could be any data type including other lists.

Lists

- Some feature of lists in Python:
  - The elements take place sequentially
  - The elements could be of any data type
  - The access to the elements of a list is done through indexing
  - Lists, lists including other lists (any nested object) could the elements of a list
  - The dimension is not constant
  - Lists are of mutable data type

Lists

Some examples of lists in Python:

Definition	Description
[]	empty list
[1,1,2,3,5,8]	a list of integers
[42, "JFM212", 3.1415]	a list of various data types
["Ankara", "Adana", "Bursa", "İzmir", "Gaziantep", "Antalya","Konya", "Samsun"]	a list of strings
[["Ankara","Konya", 7556900], ["New York","Londra",2193031], ["Antalya", "Samsun", 123466]]	a list containing lists as elements
["İller", ["ilçeler", ["beldeler", ["köyler", "mezralar", 1021]]]]	a nested list

Lists

- Access to the elements and sub-elements of a list:
  - Indexing is used to access to the elements.
  - If the element accessed is also a list, an additional indexing can be used.

```
>>> bilgi = [["Ali","Demir"],[[["Atatürk Cad.", "24"],
"06100"],"Ankara"]]
>>> print(bilgi[0])
['Ali','Demir']
>>> print(bilgi[0][1])
Demir
>>>  print(bilgi[1][0][1])
06100
```

Tuples

- A tuple is an "immutable" data type.
- The tuples are defined similar to lists but "()" is used instead of "[]".
- The access to the elements is similar to that of lists.
- The advantages of tuple over lists:
  - Tuples are in general faster to process
  - Minimizes the programming bugs since they are immutable
  - Tuples as immutable types can be used as "keys" in "dictionary" data type.

Tuples

- The elements of a tuple cannot be changed
- "Slicing" is similar to that of lists

```
>>> t = ("Lists", "and", "tuples")
>>> t[0]
'Lists'

>>> t[1:3]
('and', 'tuples')

>>> t[0]="new value"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
```

Concatenation and Repetition in Sequential Data Types

- Sequential data types in Python can be concatenated with "+" operator like we in strings:

```
>>> a = [1,2,5,4]
>>> b = [8,14,9]
>>> c = [45,10,6]
>>> a + b + c
[1, 2, 5, 4, 8, 14, 9, 45, 10, 6]
```

- Similarly, repetition is done through "*" operator

```
>>> a*4
[1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4, 1, 2, 5, 4]
```

Checking the presence of a specific element in sequential data types

- To check whether an element is contained in a sequential data type, "in" keyword/operator can be used:

```
>>> a = [1,2,5,4]
>>> 2 in a
True
>>> 7 not in a
True
>>> b = ("Ankara","İzmir","İstanbul")
>>> 'Ankara' in b
True
>>> 'Adana' not in b
True
```

Shallow/Deep Copy Operations in Sequential Data Types

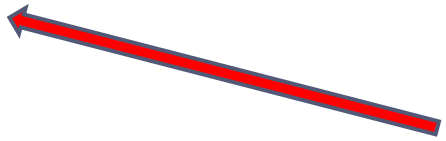
- When a new value is assigned to a variable, instead of modifying the data at the current memory address, a memory address is assigned to the variable and data is placed at the new memory address.

```
>>> x = 3
>>> y = x
>>> print(id(x), id(y))
1616756784 1616756784
>>> y = 4
>>> print(id(x), id(y))
1616756784 1616756800
>>> print(x,y)
3 4
```

Shallow/Deep Copy Operations in Sequential Data Types

- Such phenomenon is also valid for sequential data types.

```
>>> colors  = ["red","blue","green"]
>>> colors2 = colors
>>> print(id(colors),id(colors2))
4317096 4317096
>>> colors2 = ["orange","brown"]
>>> print(colors,colors2)
['red', 'blue', 'green'] ['orange', 'brown']
>>> print(id(colors),id(colors2))
4317096 33918808
```



A new memory address is assigned!

Shallow/Deep Copy Operations in Sequential Data Types

- Such phenomenon is also valid for sequential data types.

```
>>> colors  = ["red","blue","green"]
>>> colors2 = colors
>>> print(id(colors),id(colors2))
4317096 4317096
>>> colors2[0] = "orange"
>>> print(colors,colors2)
['orange', 'blue', 'green'] ['orange', 'blue', 'green']
>>> print(id(colors),id(colors2))
4317096 4317096
```

The elements of variable "colors" is also changed!

They have the same memory address!

Shallow/Deep Copy Operations in Sequential Data Types

- To overcome such problems a special "copying" operation is needed. One such method is the "**shallow copy**"

```
>>> colors = ["red","blue","green"]
>>> colors2 = colors[:]
>>> print(id(colors),id(colors2))
2678696 10456760
>>> colors2[0] = "orange"
>>> print(colors,colors2)
['red', 'blue', 'green'] ['orange', 'blue', 'green']
```

Slicing operator! (shallow copy)

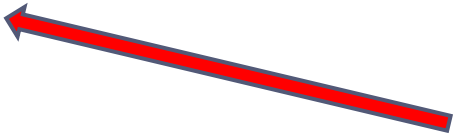
Instead of sharing a common memory address, a new memory address is assigned to the second variable!

Shallow/Deep Copy Operations in Sequential Data Types

- If the sequential type already contains another sequential type, even the shallow copy is not sufficient:

```
>>> colors = [["red","blue"],"green"]
>>> colors2 = colors[:]
>>> print(id(colors),id(colors2))
10473840 2678696
>>> colors2[0][0] = "orange"
>>> print(colors,colors2)
[['orange', 'blue'], 'green'] [['orange', 'blue'], 'green']
```

Different addresses are assigned!



The element of the variable "colors" is also changed!

Shallow/Deep Copy Operations in Sequential Data Types

- If the sequential type already contains another sequential type, a "**deep copy**" operation is needed.
- To perform a deep copy, deepcopy function from "deepcopy" module is imported.

```
>>> from copy import deepcopy
>>> colors = [["red","blue"],"green"]
>>> colors2 = deepcopy(colors)
>>> colors2[0][0] = "orange"
>>> print(orange,orange2)
[['red', 'blue'], 'green'] [['orange', 'blue'], 'green']
```

References

1 Wentworth, P., Elkner, J., Downey, A.B., Meyers, C. (2014). How to Think Like a Computer Scientist: Learning with Python (3nd edition).
2 Pilgrim, M. (2014). Dive into Python 3 by. Free online version: DiveIntoPython3.org ISBN: 978-1430224150.
3 Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.
4 Summerfield, M. (2014) Programming in Python 3 2nd ed (PIP3) : - Addison Wesley ISBN: 0-321-68056-1.
5 Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/.
6 Millman, K.J., Aivazis, M. (2011). Python for Scientists and Engineers, Computing in Science & Engineering, 13, 9-12.
7 John D. Hunter (2007). Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95.
8 Travis E. Oliphant (2007). Python for Scientific Computing, Computing in Science & Engineering, 9, 10-20.
9 Goodrich, M.T., Tamassia, R., Goldwasser, M.H. (2013). Data Structures and Algorithms in Python, Wiley.
10 http://www.diveintopython.net/
11 https://docs.python.org/3/tutorial/
12 http://www.python-course.eu
13 https://developers.google.com/edu/python/
14 http://learnpythonthehardway.org/book/