CSC340/AI320 - Artificial Intelligence Fall 2022

Lab 2

INSTRUCTOR: DR. DIAA SALAMA ABDEL MONEIM
ENG. NOHA EL MASRY
ENG. NOUR ELHUDA ASHRAF

Python types and operations(continued):

Primitive (Built-in) data types

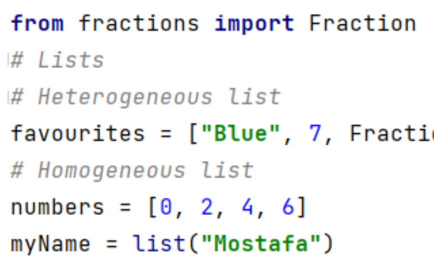
- ### Lists

The Python **list** object is the most general sequence provided by the language. Lists are **positionally ordered collections of arbitrarily typed objects** (can collect homogenous or heterogenous data), and they have **no fixed size**.

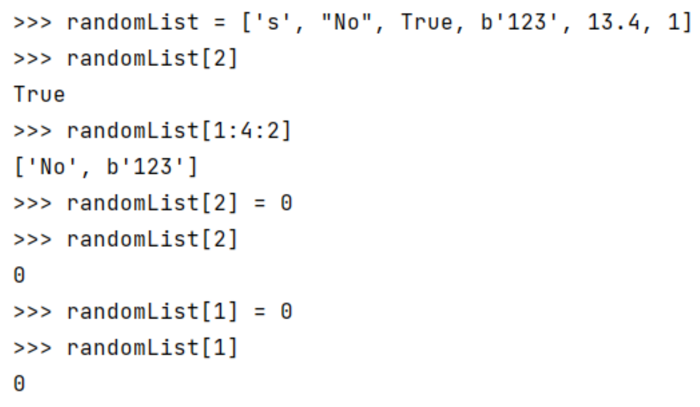
They are also **mutable** (unlike strings), lists can be modified in place by assignment to offsets as well as a variety of list method calls. Lists allow data duplication (unlike sets).

Because they are sequences, lists support all the sequence operations we discussed for strings; the only difference is that **the results are usually lists instead of strings**. Python does not have arrays; the corresponding data structure is list.

Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically one-third to one-fifth the size of equivalent C++ or Java code.



```python
from fractions import Fraction
# Lists
# Heterogeneous list
favourites = ["Blue", 7, Fraction(5,6), 3.9]
# Homogeneous list
numbers = [0, 2, 4, 6]
myName = list("Mostafa")
```



```
>>> randomList = ['s', "No", True, b'123', 13.4, 1]
>>> randomList[2]
True
>>> randomList[1:4:2]
['No', b'123']
>>> randomList[2] = 0
>>> randomList[2]
0
>>> randomList[1] = 0
>>> randomList[1]
0
```

Changing the contents of a list is only done by assigning an item or a set of items with specific values, while concatenation and multiplication don't change the original list.

```
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
>>> randomList + [1,2,3]
['s', 0, 0, b'123', 13.4, 1, 1, 2, 3]
>>> randomList * 2
['s', 0, 0, b'123', 13.4, 1, 's', 0, 0, b'123', 13.4, 1]
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
```

Although lists have **no fixed size**, Python still doesn't allow us to reference items that are not present. **Indexing off the end** of a list is always a mistake, but so is **assigning off the end**.

```
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
>>> randomList[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> randomList[6] = 'Python'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

Changing the contents of a list is only done by assigning an item or a set of items with specific values, while concatenation and multiplication don't change the original list.

```
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
>>> randomList + [1,2,3]
['s', 0, 0, b'123', 13.4, 1, 1, 2, 3]
>>> randomList * 2
['s', 0, 0, b'123', 13.4, 1, 's', 0, 0, b'123', 13.4, 1]
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
```

Although lists have **no fixed size**, Python still doesn't allow us to reference items that are not present. **Indexing off the end** of a list is always a mistake, but so is **assigning off the end**.

```
>>> randomList
['s', 0, 0, b'123', 13.4, 1]
>>> randomList[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> randomList[6] = 'Python'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list assignment index out of range
```

One nice feature of Python's core data types is that they support **arbitrary nesting**, we can nest them in any combination, and as deeply as we like.

```
>>> mat1 = [[1,2,3],[4,5,6],[7,8,9]]
>>> mat1
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> mat1[1]
[4, 5, 6]
>>> mat1[1][2]
6
```

In addition to sequence operations and list methods, Python includes a more advanced operation known as a **list comprehension expression**, which turns out to be a powerful way to process structures like our matrix.

```
>>> col1 = [row[1] for row in mat1]
>>> col1
[2, 5, 8]
```

Python provides powerful methods for lists, and because lists are mutable, most list methods also change the list object in place, instead of creating a new one.

Function	Input	Returned Value
append()	object	list w/ appended obj at last
insert()	index, object	list w/ object at index
extend()	list	concatenated list
remove()	object value	list w/o object
pop()	object index	list w/o object
index()	element value	index of the element
del [n]/list_name	index or list name	list w/o nth element/deleted list
clear()	-	remove all elements
len()	-	# of elements in the list
sort()	-	ascendingly sorted list
sort(reverse = True)	-	descending sorted list
sort(key = function)	customized function	customized sorted list
reverse()	-	reverse list order
copy()	-	copy a list to another one
+	more than one list	concatenated list

The Map Function built-in function applies a function to items in a sequence and collects all the results in a new list.

```
>>> list(map(abs, [-1, -2, 0]))
[1, 2, 0]
```

• Dictionaries

Python dictionaries are something completely different, they are not sequences at all, but are instead known as **mappings**.

Mappings are also collections of other objects, but they **store objects by key instead of by relative position**. In fact, mappings don't maintain any reliable left-to-right order; they simply map keys to associated values.

Dictionaries, the only mapping type in Python's core objects set, are also **mutable**: like lists, they may be changed in place and can grow and shrink on demand.

```
# Dictionaries
myShoppingDictionary = {'food': 'Bread', 'quantity': 4, 'color': 'White'}
```

We can index this dictionary by **key** to fetch and **change the keys' associated values.**

```
>>> myShoppingDictionary = {'food': 'Bread', 'quantity': 4, 'color': 'White'}
>>> myShoppingDictionary["food"]
'Bread'
>>> myShoppingDictionary['quantity'] += 1
>>> myShoppingDictionary['quantity']
5
>>> myShoppingDictionary['Quantity']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Quantity'
```

It is perhaps more common to see dictionaries built up in different ways (**it's rare to know all your program's data before your program runs**). For example, start with an empty dictionary and fill it out one key at a time.

Unlike **out-of-bounds assignments in lists, which are forbidden**, **assignments to new dictionary keys create those keys**.

```
>>> graduateInfo = {}
>>> graduateInfo['Name'] = "Mostafa"
>>> graduateInfo['Age'] = "26"
>>> graduateInfo['Job'] = "Teaching Assistant"
>>> graduateInfo
{'Name': 'Mostafa', 'Age': '26', 'Job': 'Teaching Assistant'}
```

We can also make dictionaries by passing to the **dict type name** either keyword arguments (**a special name=value syntax in function calls**), or the result of **zipping together sequences of keys and values obtained at runtime** (e.g., from files).

```
>>> mostafa = dict(age=26, job="Teaching Assistant", gov='Cairo')
>>> mostafa
{'age': 26, 'job': 'Teaching Assistant', 'gov': 'Cairo'}
>>> haytham = dict(zip(['age', 'job', 'gov'], [28, "Teaching Assistant", 'Cairo']))
>>> haytham
{'age': 28, 'job': 'Teaching Assistant', 'gov': 'Cairo'}
```

We can nest dictionaries into each other or any other types of objects, we can access the components of this structure much as any list-based matrix, but this time most indexes are dictionary keys, not list offsets.

```
>>> empRec = {'name': {'first': "Mostafa", 'last': "Badr"}, 'jobs': ["TA", 'dev'], 'age': 25.5}
>>> empRec
{'name': {'first': 'Mostafa', 'last': 'Badr'}, 'jobs': ['TA', 'dev'], 'age': 25.5}
>>> empRec['name']
{'first': 'Mostafa', 'last': 'Badr'}
>>> empRec['name']['last']
'Badr'
>>> empRec['jobs'][-1]
'dev'
>>> empRec['jobs'].append("proctor")
>>> empRec
{'name': {'first': 'Mostafa', 'last': 'Badr'}, 'jobs': ['TA', 'dev', 'proctor'], 'age': 25.5}
```

As mappings, **dictionaries support accessing items by key only**, we can **assign to a new key to expand a dictionary**, **fetching a nonexistent key is still a mistake**.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'b': 2, 'c': 3}
>>> D['e'] = 99
>>> D
{'a': 1, 'b': 2, 'c': 3, 'e': 99}
>>> D['f']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'f'
```

We can't always know what keys will be present when we write our code, the dictionary **in** membership expression allows us to query the existence of a key and branch on the result with a Python if statement.

```
>>> 'f' in D
False
>>> if not 'f' in D:\
... print("'f' is not in dictionary D")
...
'f' is not in dictionary D
```

In Python 3.7 and later, dictionaries are changeable. The matter is different for Python 3.6 or earlier

Function	Input	Output
clear()	-	Removes all the elements from the dictionary
copy()	-	a copied dictionary
fromkeys()	keys + values	a constructed dictionary
get()	key	the value of the specified key
items()	-	a list of all pairs
keys()	-	a list of all keys
values()	-	a list of all values
pop()	key + value (optional)	a dictionary w/o the item
popitem()	-	a dictionary w/o the last inserted item
setdefault()	key + value (optional)	the value of the specified key. If the key does not exist, insert the key with the specified value
update()	key + value	a dictionary with inserted item

- ## Tuples

The tuple object is like **a list that cannot be changed**, **tuples are sequences**, like lists, but they are **immutable**, like strings.

They're used to represent fixed collections of items.

They are normally coded in **parentheses** instead of square brackets, and they support **arbitrary types**, **arbitrary nesting**, and **usual sequence operations**.

```
>>> T = (1, 2, 3, 4)
>>> T
(1, 2, 3, 4)
>>> T + (5, 6)
(1, 2, 3, 4, 5, 6)
>>> T[0]
1
>>> T[1:3]
(2, 3)
```

The primary distinction for tuples is that they **cannot be changed once created**. That is, they are **immutable** sequences

```
>>> T[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Like lists and dictionaries, tuples support **mixed types and nesting**, but they don't grow and shrink because they are immutable.

```
>>> empRec = (507, 50.26, 'Mostafa', ['dev', 'TA', 'proctor'])
>>> empRec[1]
50.26
>>> empRec[3][1]
'TA'
```

How to modify content of a tuple?

```
>>> empRecList = list(empRec)
>>> empRecList[3].append("Assistant Supervisor")
>>> empRecList
[507, 50.26, 'Mostafa', ['dev', 'TA', 'proctor', 'Assistant Supervisor']]
>>> empRec = tuple(empRecList)
>>> empRec
(507, 50.26, 'Mostafa', ['dev', 'TA', 'proctor', 'Assistant Supervisor'])
```

Unpacking to a variable list can be done by using **asterisk** (*). Asterisk association is not limited to the last variable it may be associated to any other.

```
>>> (id, age, name, *jobs) = empRec
>>> id
507
>>> age
50.26
>>> name
'Mostafa'
>>> jobs
[['dev', 'TA', 'proctor', 'Assistant Supervisor']]
```

At last, tuple has only two built-in function **count**() for counting number of elements in a tuple and **index**() to retrieve the index of a specific element inside a tuple. **Using any list built-in functions with tuple will result in an error**.

```
>>> empRec
(507, 50.26, 'Mostafa', ['dev', 'TA', 'proctor', 'Assistant Supervisor'])
>>> empRec.count('Mostafa')
1
>>> empRec.index("Mostafa")
2
```

- ## Files

File objects are Python code’s main interface to external files on your computer. They can be used to read and write text memos, audio clips, Excel documents, saved email messages, and whatever else you happen to have stored on your machine.

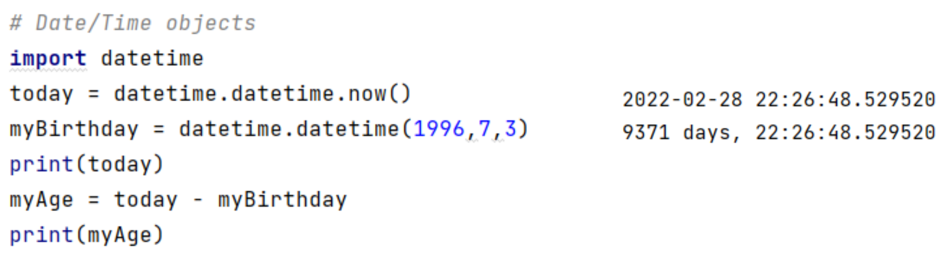
Function	Input	Description
open()	file pathname/ file name + file operation	
read(n)	file + number of character to be read	reads n characters from a file
readline()	-	read a single line
write()	text to be written/appended	write text in file
close()	-	close the opened file

File Operations:

1. "r" - Read - Default value. Opens a file for reading, error if the file does not exist.
2. "a" - Append - Opens a file for appending, creates the file if it does not exist.
3. "w" - Write - Opens a file for writing, creates the file if it does not exist.
4. "x" - Create - Creates the specified file, returns an error if the file exists.
5. "t" - Text - Default value. Text mode.
6. "b" - Binary - Binary mode (e.g. images).

- ## Time and Date

Before using Time/Date objects you need to import datetime package.



```python
# Date/Time objects
import datetime
today = datetime.datetime.now()
myBirthday = datetime.datetime(1996,7,3)
print(today)
myAge = today - myBirthday
print(myAge)
```

```
2022-02-28 22:26:48.529520
9371 days, 22:26:48.529520
```

You can reformat dates using strftime() function

Functions and Generators:

In simple terms, a function is a device that groups a set of statements so they can be run more than once in a program (a packaged procedure invoked by name).

Functions also can compute a result value and let us specify parameters that serve as function inputs and may differ each time the code is run.

Functions are also the most basic program structure Python provides for maximizing code reuse, and lead us to the larger notions of program design.

As we'll see, functions let us split complex systems into manageable parts. By implementing each part as a function, we make it both reusable and easier to code.

- ### Functions

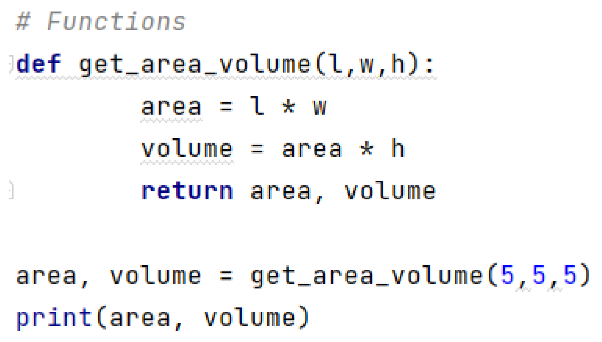
  To define a function in Python, we use **def** keyword. Function prototype in Python differs in missing the type of the returned object.

  def function_name(parameter1, parameter2):

    #function Body

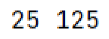
    #return statements

In Python function can return multiple objects.



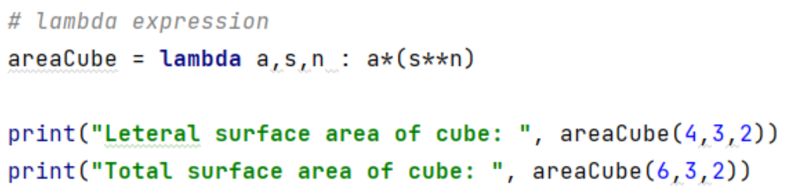
```
# Functions
def get_area_volume(l,w,h):
        area = l * w
        volume = area * h
        return area, volume

area, volume = get_area_volume(5,5,5)
print(area, volume)
```



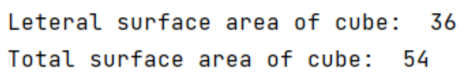
```
25 125
```

Python has a special anonymous function called lambda can take any number of arguments but can only have one expression.



```
# lambda expression
areaCube = lambda a,s,n : a*(s**n)

print("Leteral surface area of cube: ", areaCube(4,3,2))
print("Total surface area of cube: ", areaCube(6,3,2))
```



```
Leteral surface area of cube:  36
Total surface area of cube:  54
```

Note: Lambda is commonly used when you want to pass a function as an argument to higher-order functions.

Another special function is called map(function, iterator). It returns a map object(which is an iterator) of the results after applying the given function to each item of a given iterable such as list, tuple, etc.

Exercises:

For all following exercise, write the main function to test your implementation.

1. **Write a Python function to check whether a number is perfect or not.**

   ```
   Input/Output Scenario:
   Please enter a number: 28
   28 is a perfect number
   Please enter a number: 9
   9 is not a perfect number
   ```

2. **Write a Python function that takes a list and returns a new list with unique elements of the first list.**

   ```
   Input/Output Scenario:
   Please enter a sample list: [1,2,3,3,3,4,4,5]
   The unique list: [1, 2, 3, 4, 5]
   ```

3. **Write a Python program to convert a given list of tuples to a list of lists**

   ```
   Original list of tuples: [(1, 2), (2, 3, 5), (3, 4), (2, 3, 4, 2)]
   Convert the said list of tuples to a list of lists: [[1, 2], [2, 3, 5], [3, 4], [2, 3,
   4, 2]]
   ```

4. **Write a Python program to find the key of the maximum value and minimum value in a dictionary.**

5. **Write a Python program to sort the dictionary once by key.**

Extra Questions:

6. **Write a Python program that accepts a separator and a separated sequence of words as input and prints the words in a hyphen-separated sequence after sorting them alphabetically.**

    Input/Output Scenario:
    Please enter the separator: ,
    Please enter the sequence: Topics,Intelligence,Artificial,Selected

    The ordered sequence is Artificial-Intelligence-Selected-Topics

7. **Write a Python program to check whether a list contains a sublist.**

    Input/Output Scenario:
    Please enter a list: [12,5,6,11,21,13]
    Please enter a sublist to be checked: [1,5,6]
    [1,5,6] is not a sublist of [12,5,6,11,21,13]