AUG
Faculty of Computer and Information Sciences
Ain Shams University

Session 3
More Python
(Strings, Dictionaries & Files)

Strings

A string in Python is a sequence of characters, and like many other popular programming languages, it can be considered as an array (list) of characters.
It is surrounded by either single, or double quotes.

```
'hello' is the same as "hello"
```

Triple quotes can be used in Python to represent multiline strings.

```
'''Multiple
line
string'''
```

We can print a string as follow:

```
print('Hello World!')
```

or assign it to variable:

```
name = 'AUG'
print(name )
```

Escape Characters

If we tried to print a single or double quote, it generates an error as they are an indication for a text, so they can't come alone.

An escape character is a backslash ‘\’ followed by the character you want to insert like (\', \", \\, \n, \t).

```
txt = "She said \"Never let go\"."
print(txt) # Output: She said "Never let go".
```

Indexing & slicing

Python strings can be indexed using the same notation as lists since strings are lists of characters. A single character can be accessed with square bracket notation ([index]), or a substring can be accessed using slicing ([start:end]).

And like lists, indexing with negative numbers counts from the end of the string.

```
str = 'yellow'
str[1]      # => 'e'
str[4:6]    # => 'ow'
str[:4]     # => 'yell'
str[-1]     # => 'w'
str[-3:]    # => 'low'
```

If you try to access an index that doesn’t exist, an IndexError is generated.

```
fruit = "Berry"
char = fruit[6]
IndexError: string index out of range
```

Change strings

Strings are immutable in Python. This means that once a string has been defined, it can’t be changed. We can reassign different strings to the same name, but we cannot delete or remove characters from it. This is unlike data types like lists, which can be modified once they are created.

```
str = 'hello'
Str[0] = 'k'
TypeError: 'str' object does not support item assignment
```

In keyword

The ‘in’ keyword is used to determine if a letter or a substring exists in a string.

It returns ‘True’ if a match is found, otherwise ‘False’ is returned.

```
game = "Popular Nintendo Game: Mario Kart"

print("l" in game) # Output: True
print("Mario" in game) # Output: True
print("x" in game) # Output: False
print("he" not in game) # Output: True
```

Iterating through a string

To iterate through a string in Python, “for...in” notation is used.

```
str = "hello"
for c in str:
    print(c)

# Output: h
# Output: e
# Output: l
# Output: l
# Output: o
```

String concatenation

To combine the content of two strings into a single string, Python provides the ‘+’ operator. This process of joining strings is called concatenation.

```
x = 'One fish, '
y = 'two fish.'
z = x + y
print(z)
# Output: One fish, two fish.
```

Important built-in functions

- .upper()

  Return string with all characters converted to uppercase

```
name = 'Aug'
print(name )
# output: AUG
```

- .lower()

  Return string with all characters converted to lowercase

```
name = 'HelLO'
print(name )
# output: hello
```

- len(string)

  Return length of the string.

```
print( len('Hello'))
# output: 5
```

- .strip()

  Used to remove characters from the beginning and end of a string.

  - If no argument is passed, the default is to remove whitespaces.

```
str = '    Apples   and Oranges    '
print(str.strip())
#  output:  'Apples   and Oranges'
```

  - If an argument is passed, it will be the set of characters to be stripped.

```
str = '...+...lemons and limes...-...'
print(str.strip('.'))
#  output:  '+...lemons and limes...-'
```

- .replace(arg1, arg2)

  Used to replace the occurrence of the first argument (the old substring to be replaced) with the second argument (the new

substring that will replace every occurrence of the first one) within the string.

```
fruit = 'Strawberry'
print(fruit.replace('r', 'R'))
# output: StRawbeRRy
```

- .split()

  Used to split string into list of items (list of strings).
  - If no argument is passed, the default is to split on whitespaces.

    ```
    str = 'Hello World'
    print(str.split())
    # output: ['Hello', 'World']
    ```

  - If an argument is passed, it will be the delimiter on which string to be splitted.

    ```
    str = 'Hello World'
    print(str.split('o'))
    # output: ['Hell', 'W', 'rld']
    ```

- .find(string)

  Returns the index of the first occurrence of the string passed as an argument, and returns `-1` if no occurrence is found.

  ```
  str = 'We are here.'
  print(str.find('e'))
  # output: 1
  ```

- .count(string)

  Returns the number of times a specified value occurs in a string.

  ```
  str = 'We are here.'
  print(str.count('e'))
  # output: 4
  ```

- .join(list of strings)

  Concatenates a list of strings together to create a new string joined with the desired delimiter. It's run on the delimiter and the array of strings to be concatenated together is passed in as an argument.

```
str = '-'.join( ['Codecademy', 'is', 'awesome'])
print(str)
#  output: Codecademy-is-awesome
```

- .format(arguments)
  - Replaces empty brace '{}' placeholders in the string with its arguments.

```
msg1 = 'Fred scored {} out of {} points.'
print(msg1.format(3, 10))
#  output: Fred scored 3 out of 10 points.
```

  - If keywords are specified within the placeholders, they are replaced with the corresponding named arguments to the method.

```
msg2 = 'Fred {verb} a {adjective} {noun}.'
print(msg2.format(adjective='fluffy', verb='tickled', noun='hamster'))
#  output: Fred tickled a fluffy hamster.
```

Dictionaries

They're used to store data in 'key : value' pairs. Dictionaries can be considered as an object which has some properties (keys) and each property has its value (values).
Pairs can be of any data type – string, integer, list, another dictionary, boolean, etc. 'Keys' must always be an immutable data type, such as strings, numbers and also must be unique but 'values' don't.

The syntax for a Python dictionary begins with the left curly brace ({), ends with the right curly brace (}), and contains zero or more 'key : value' items separated by commas (,). The 'key' is separated from the 'value' by a colon (:).

```
dict = {
    'brand': 'x',
    'model': 'y',
    'year': 2013,
    'available': True
}
```

Accessing & writing data

You can access the items of a dictionary by referring to its key name, inside square brackets.

```
my_dictionary = {"song": "Estranged", "artist": "Guns N' Roses"}
                print(my_dictionary["song"])
                     # Output: Estranged
```

But attempting to access a value with a key that does not exist will cause a KeyError.

```
my_dictionary = {"song": "Estranged", "artist": "Guns N' Roses"}
                print(my_dictionary["year"])

                          KeyError: 'year'
```

Also, to add or modify data in the dictionary, we access it by the same way. If the key already exists, the old value will be overwritten, else, a new property (key) will be added with the assigned (value).

```
dict = {"name": "Ahmed", "job": "Programmer", "gender": "Male"}

# modify value of an exist key
dict["job"] = "Developer"
# => dict = {"name": "Ahmed", "job": "Developer", "gender": "Male"}

# add new property
dict["salary"] = 7500
# => dict = {"name": "Ahmed", "job": "Developer", "gender": "Male",
#            "salary": 7500}
```

Del keyword

Used to delete an item with specified key name, or deleting the whole dictionary.

```
dict = {"brand": "x", "model": "y", "year": "2015", "available":
                                    True}

# deleting specific item
del dict["available"]
# => dict = {"brand": "x", "model": "y", "year": "2015"}

# deleting the whole dictionary
del dict
```

Looping through a dictionary

You can loop through a dictionary using “for...in” notation.

When looping through a dictionary, the return values are the ‘keys’ of the dictionary.

```
dict = {"brand": "x", "model": "y", "year": "2015", "available":
                                    True}

for d in dict:
print(d, dict[d])
```

```
# Output: brand x
# Output: model y
# Output: year 2015
# Output: available True
```

Dictionary functions

- .copy()

  Copy the dictionary into another, and if one of them is changed, the other won't be affected (shadow copying).

- .update()

  Combine two dictionaries in one.

  For dict1.update(dict2), the key-value pairs of dict2 will be written into the dict1 dictionary.

  For keys in *both* dict1 and dict2, the value in dict1 will be overwritten by the corresponding value in dict2.

```
dict1 = {'color': 'blue', 'shape': 'circle'}
dict2 = {'color': 'red', 'number': '42'}
dict1.update(dict2)
print(dict1)
# output: {'color': 'red', 'shape': 'circle', 'number': '42'}
```

- .pop(keyname)

  Remove a specific item and return its value. It can take an extra parameter to return if the key doesn't exist, else, it raises an error.

- .popitem()

  Remove the last inserted item and return the item as a tuple (key, value).

- .clear()

  Remove all items from the dictionary.

- .get(keyname)

  Return value of item with specific key. It takes an optional value as the second argument, and returns the value for the specified `key` if it exists, else,it returns the second argument and if it isn't specified then `None` is returned.

- keys()

  Return list of keys (the first object in the key-value pair). When an item is added to the dictionary, the object also gets updated.

```
dict = {'a': 'ant', 'b': 'bumblebee', 'c': 'cheetah'}
x = dict.keys()
print(x)
# output: ['a', 'b', 'c']
dict['d'] = 'dog'
print(x)
# output: ['a', 'b', 'c', 'd']
```

- .values()

  Return list of values (the second object in the key-value pair). When an item is changed, the object also gets updated.

```
dict = {'a': 'ant', 'b': 'bumblebee', 'c': 'cheetah'}
x = dict.values()
print(x)
# output: ['ant', 'bumblebee', 'cheetah']
dict['d'] = 'dog'
print(x)
# output: ['ant', 'bumblebee', 'cheetah', 'dog']
```

- .items()

  Return list of items (key-value pairs). When an item is changed, the object also gets updated.

```
dict = {'a': 'ant', 'b': 'bumblebee', 'c': 'cheetah'}
x = dict.items()
print(x)
# output: [('a', 'ant'), ('b', 'bumblebee'), ('c', 'cheetah')]
dict['d'] = 'dog'
print(x)
# output: [('a', 'ant'), ('b', 'bumblebee'), ('c', 'cheetah'), ('d', 'dog')]
```

Files

When we want to read from or write to a file, we need to open it first, and when we are done, it needs to be closed.

1. Open a file.
2. Perform operations.
3. Close the file.

```
myFile = open(('fileName.txt', mode))
#some operations
myFile.close()
```

Note: if file exists in the same location only type ‘fileName.txt’, otherwise, type the total file path.

File modes

- ‘r’: to read from the file.
- ‘a’: to append to the end of the file (without affecting existing text).
- ‘w’: to overwrite any existing text in file.

Note: to read and write at the same time, add ‘+’ sign (‘r+’, ‘a+’, ‘w+’).
‘r+’ starts from the beginning of file & ‘a+’ starts from the end of file.

Note: if there is no file with the specified name it will give an error in ‘r’ mode, otherwise, it will be created.

Reading from text files

Files return strings even if we store numbers, so if we want to use them as numbers not strings we have to use ‘int()’ or ‘float()’ functions.

- .read()

  Return the whole text in the file.

- .readline()

  Return one line then the cursor moves to the next line in the file.

- .readlines()

  Return all lines in a list, where each item represents a line.

- Using ‘in’ keyword

```
myFile = open(('fileName.txt', mode)
for line in myFile:
    print(line)
myFile.close()
```

Writing in text files

- .write(string)

  Write a given string in the file, if it’s on ‘a’ mode it writes at the end of file, if it’s on ‘w’ mode it will overwrite any text found.

- .writelines(list of strings)

  Write a given list of strings in the file, all on the same line if a line break isn’t inserted inside the list.

```
myFile = open(('fileName.txt', 'w')
myFile.writelines(['first sentence!', 'second sentence!'])
#file will look like:
first sentence!second sentence!
myFile.writelines(['first sentence!', '\n',  'second sentence!'])
#file will look like:
first sentence!
second sentence!
myFile.close()
```

FASTA files

In bioinformatics, FASTA is a writing method used to write files containing nucleotide sequences (DNA or RNA sequences) or amino acids (protein) sequences.

> Sequence1 ID
Sequence1

> Sequence2 ID
Sequence2

> Sequence3 ID
Sequence3

Reading from FASTA files

```
from Bio import SeqIO

fasta_sequence = SeqIO.parse(open('input_file.fasta', 'fasta')

for seq_record in fasta_sequence:
    print(seq_record.id)
    print(repr(seq_record.seq))
    print(len(seq_record))
```