PYTHON DATA STRUCTURES

Slides 3

Content

- Lists
- Tuples
- Dictionaries
- Python Object Serialization
  - JSON
  - Pickle
- Python Functional Programming Basics
- Named Functions and Anonymous Functions in Python
  - Lambda Operator
- Higher-Order Functions
  - map()
  - reduce()
  - filter()

LISTS

- Lists in Python are zero-based indexed sequences of **mutable** values with the first value numbered zero. You can remove or replace elements in a list as well as append elements to the end of a list.

```
tempc = [38.4, 19.2, 12.8, 9.6]
print(tempc[0])
print(tempc)
```

```
38.4
[38.4, 19.2, 12.8, 9.6]
```

```
print(len(tempc))
```

```
4
```

LISTS count()

count() is an in built function in Python that returns count of how many times a given object occurs in list.

Syntax :

list_name.count(object)

Parameters :

Object is the things whose count is to be returned.

Returns :

count() method returns count of how many times obj occurs in list.

LISTS count()

```
list1 = [1, 1, 1, 2, 3, 2, 1]

# Counts the number of times 1 appears in list1
print(list1.count(1))

list2 = ['a', 'a', 'a', 'b', 'b', 'a', 'c', 'b']

# Counts the number of times 'b' appears in list2
print(list2.count('b'))

list3 = ['Cat', 'Bat', 'Sat', 'Cat', 'cat', 'Mat']

# Counts the number of times 'Cat' appears in list3
print(list3.count('Cat'))
```

```
4
3
2
```

LISTS sort()

- The sort function can be used to sort a list in ascending, descending or user defined order. This function can be used to sort list of integers, floating point number, string and others.

```
In [5]: cars = ['Ford', 'BMW', 'Volvo']

        cars.sort()

        print(list(cars))

['BMW', 'Ford', 'Volvo']

In [7]: cars.sort(reverse=True)
        print(list(cars))

['Volvo', 'Ford', 'BMW']
```

TUPLES

- Tuples are an **immutable** sequence of objects, though the objects contained in a tuple can themselves be immutable or mutable. Tuples can contain different underlying object types, such as a mixture of string, int, and float objects, or they can contain other sequence types, such as sets and other tuples.
- For simplicity, think of tuples as being similar to immutable lists. However, they are different constructs and have very different purposes.
- Tuples are similar to **records (row)** in a relational database table, where each record has a structure, and each field defined with an ordinal position in the structure has a meaning.

TUPLES

In [8]:
```
rec0 = "Jeff", "Aven", 46
rec1 = "Barack", "Obama", 54
rec2 = "John F", "Kennedy", 46
rec3 = "Jeff", "Aven", 46
rec0
```

Out[8]:
```
('Jeff', 'Aven', 46)
```

In [9]:
```
len(rec0)
```

Out[9]:
```
3
```

In [10]:
```
print("first name: " + rec0[0])
```

```
first name: Jeff
```

In [13]:
```
# create tuple of tuples
all_recs = rec0, rec1, rec2, rec3
print(all_recs)
all_recs
```

```
(('Jeff', 'Aven', 46), ('Barack', 'Obama', 54), ...
```

```
('John F', 'Kennedy', 46), ('Jeff', 'Aven', 46))
```

Out[13]:
```
(('Jeff', 'Aven', 46),
 ('Barack', 'Obama', 54),
 ('John F', 'Kennedy', 46),
 ('Jeff', 'Aven', 46))
```

TUPLES

```
In [16]: # create list of tuples
         list_of_recs = [rec0, rec1, rec2, rec3]
         print(list_of_recs)
         list_of_recs

[('Jeff', 'Aven', 46), ('Barack', 'Obama', 54), ...

('John F', 'Kennedy', 46), ('Jeff', 'Aven', 46)]

Out[16]: [('Jeff', 'Aven', 46),
          ('Barack', 'Obama', 54),
          ('John F', 'Kennedy', 46),
          ('Jeff', 'Aven', 46)]
```

DICTIONARIES

- Dictionaries, or dicts, in Python are unordered mutable sets of key/value pairs.
- Dict objects are denoted by curly brackets (braces) ({}), which you can create as empty dictionaries by simply executing a command such as my_empty_dict = {}.
- Unlike with lists and tuples, where an element is accessed by its ordinal position in the sequence (its index), an element in a dict is accessed by its key. A key is separated from its value by a colon (:), whereas key/value pairs in a dict are separated by commas.

DICTIONARIES

In [17]:
```
dict0 = {
    'fname':'Jeff',
    'lname':'Aven',
    'pos':'author'
}
dict1 = {'fname':'Barack', 'lname':'Obama', 'pos':'president'}
dict2 = {'fname':'Ronald', 'lname':'Reagan', 'pos':'president'}
dict3 = {'fname':'John', 'mi':'F', 'lname':'Kennedy','pos':'president'}
dict4 = {'fname':'Jeff', 'lname':'Aven', 'pos':'author'}
len(dict0)
```

Out[17]:
```
3
```

In [18]:
```
print(dict0['fname'])
```

```
Jeff
```

In [19]:
```
dict0.keys()
```

Out[19]:
```
dict_keys(['fname', 'lname', 'pos'])
```

In [23]:
```
dict0.values()
```

Out[23]:
```
dict_values(['Jeff', 'Aven', 'author'])
```

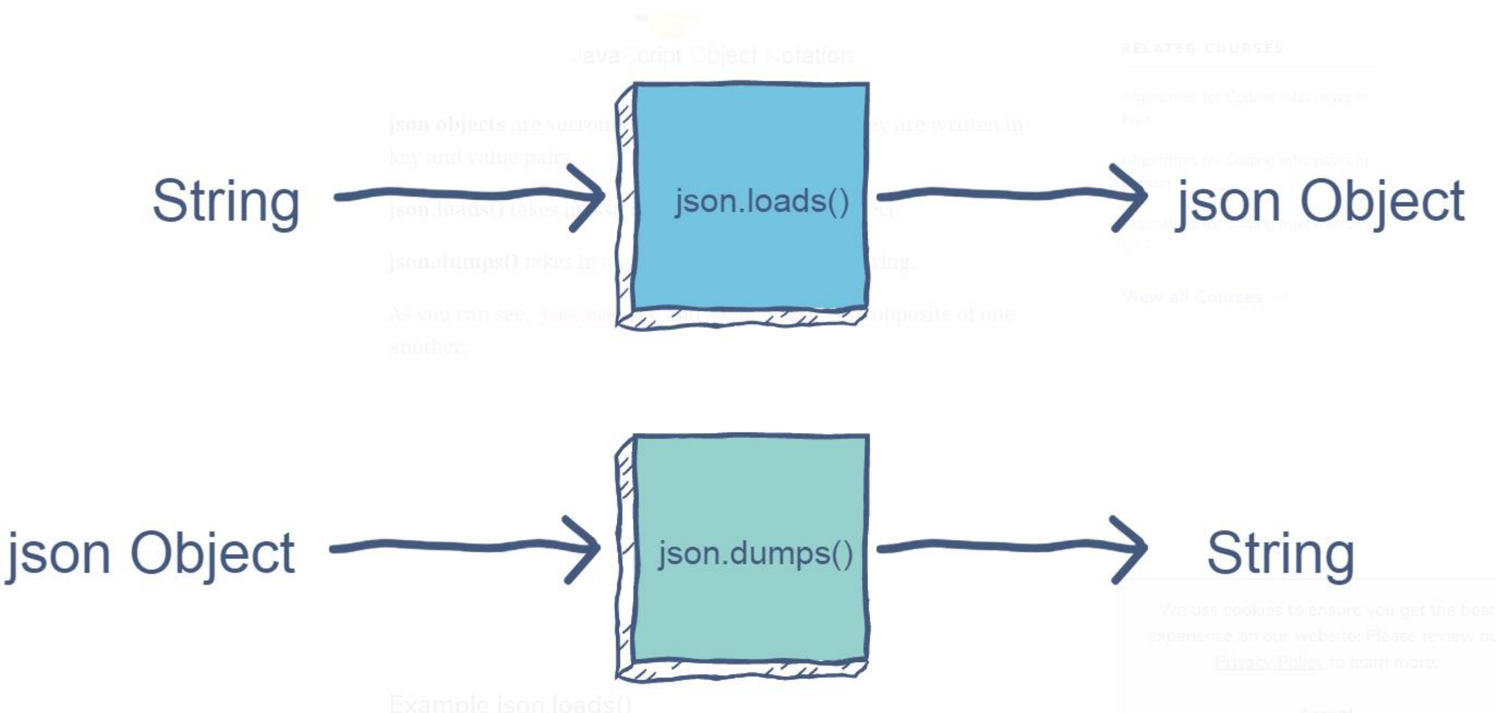
Python Object Serialization

- Serialization is the process of converting an object into a structure that can be unpacked (deserialized) at a later point in time on the same system or on a different system.
- Serialization, or the ability to serialize and deserialize data, is a necessary function of any distributed processing system and features heavily throughout the Hadoop and Spark projects.

JSON

- JSON (JavaScript Object Notation) is a common serialization format. JSON has extended well beyond JavaScript and is used in a multitude of platforms, with support in nearly every programming language. It is a common response structure returned from web services.
- JSON is supported natively in Python using the json package. A package is a set of libraries or a collection of modules (which are essentially Python files). The json package is used to encode and decode JSON. A JSON object consists of key/value pairs (dictionaries) and/or arrays (lists), which can be nested within each other.

JSON



https://www.educative.io

JSON

```
In [9]: import json
        from pprint import pprint
        json_str = '''{"people" : [
           {"fname": "Jeff", "lname": "Aven", "tags": ["big data","hadoop"]},
           {"fname": "Doug", "lname": "Cutting", "tags": ["hadoop","avro","apache","java"]},
           {"fname": "Martin", "lname": "Odersky", "tags": ["scala","typesafe","java"]},
           {"fname": "John", "lname": "Doe", "tags": []}
        ]}'''
        people = json.loads(json_str)

        len(people["people"])

Out[9]: 4
```

```
In [10]: print(people["people"][0]["fname"])

Jeff
```

```
In [11]: # add tag item to the first person
         people["people"][0]["tags"].append('spark')
```

JSON

```
In [12]: pprint(people)

{'people': [{'fname': 'Jeff',
             'lname': 'Aven',
             'tags': ['big data', 'hadoop', 'spark']},
            {'fname': 'Doug',
             'lname': 'Cutting',
             'tags': ['hadoop', 'avro', 'apache', 'java']},
            {'fname': 'Martin',
             'lname': 'Odersky',
             'tags': ['scala', 'typesafe', 'java']},
            {'fname': 'John', 'lname': 'Doe', 'tags': []}]}

In [13]: # delete the fourth person
         del people["people"][3]

In [14]: pprint(people)

{'people': [{'fname': 'Jeff',
             'lname': 'Aven',
             'tags': ['big data', 'hadoop', 'spark']},
            {'fname': 'Doug',
             'lname': 'Cutting',
             'tags': ['hadoop', 'avro', 'apache', 'java']},
            {'fname': 'Martin',
             'lname': 'Odersky',
             'tags': ['scala', 'typesafe', 'java']}]}
```

PICKLE

- Pickle is a serialization method that is proprietary to Python. Pickle is **faster than JSON**. However, it lacks the portability of JSON, which is a universally interchangeable serialization format.
- The Python pickle module converts a Python object or objects into a byte stream that can be transmitted (over network), stored (in memory), and reconstructed into its original state.
- Notice that the **load** and **dump** idioms are analogous to the way you serialize and deserialize objects using JSON (straightforward).
- The **pickle.dump** approach saves the pickled object to a file, whereas **pickle.dumps** (note that 's') returns the pickled representation of the object as a string (to the bytes) that may look strange, although it is not designed to be human readable. Also, we have **pickle.load** and **pickle.loads**.

PICKLE

```
import pickle
obj = { "fname": "Jeff",
    "lname": "Aven",
    "tags": ["big data","hadoop"]}
```

```
str_obj = pickle.dumps(obj)
```

```
print(str_obj)
```

```
b'\x80\x03}q\x00(X\x05\x00\x00\x00fnameq\x01X\x04\x00\x0
4\x00\x00\x00tagsq\x05]q\x06(X\x08\x00\x00\x00big dataq\x …
```

```
… x00\x00Jeffq\x02X\x05\x00\x00\x00lnameq\x03X\x04\x00\x00\x00Avenq\x04X\x0
  q\x07X\x06\x00\x00\x00hadoopq\x08eu.'
```

PICKLE

```
In [28]: pickled_obj = pickle.loads(str_obj)

In [29]: print(pickled_obj)

{'fname': 'Jeff', 'lname': 'Aven', 'tags': ['big data', 'hadoop']}

In [30]: print(pickled_obj["fname"])

Jeff

In [31]: pickled_obj["tags"].append('spark')
         print(str(pickled_obj["tags"]))

['big data', 'hadoop', 'spark']
```

PICKLE

```
In [32]: # dump pickled object to a string
         pickled_obj_str = pickle.dumps(pickled_obj)
         # dump pickled object to a pickle file
         pickle.dump(pickled_obj, open('object.pkl', 'wb'))

In [33]: print(pickled_obj_str)

         b'\x80\x03}q\x00(X\x05\x00\x00\x00fnameq\x01X\x04\x0
         4\x00\x00\x00tagsq\x05]q\x06(X\x08\x00\x00\x00big da
```

Python Functional Programming Basics

Python’s functional support embodies all of the functional programming paradigm characteristics that you would expect, even including the followings:

- Support for anonymous functions
- Support for higher-order functions

Named Functions and Anonymous Functions in Python

- Named functions can contain statements such as print, but anonymous functions can contain only a single or compound expression, which could be a call to another named function that is in scope.
- Named functions can also use the return statement, which is not supported with anonymous functions.
- Anonymous (unnamed) functions in Python are implemented using the lambda construct rather than using the **def** keyword for named functions. Anonymous functions accept any number of input arguments but return just one value. This value could be another function, a scalar value, or a data structure such as a list.

Lambda Operator

- The lambda operator or lambda function is a way to create small anonymous functions, i.e. functions without a name.
- These functions are throw-away functions, i.e. they are just needed where they have been created.
- Lambda functions are mainly used in combination with the functions filter(), map() and reduce().

Syntax:

lambda arguments: expression

Lambda Operator

- Difference between a named function **def** defined function and **lambda** function is shown:

```
def cube(y):
    return y*y*y;

g = lambda x: x*x*x
print(g(7))

print(cube(5))
```

```
343
125
```

Named Functions and Anonymous Functions in Python

```
In [1]: # named function
        def plusone(x):
            return x+1 # 4 space indent

        plusone(1)

Out[1]: 2

In [2]: type(plusone)

Out[2]: function

In [3]: # anonymous function
        plusonefn = lambda x: x+1
        plusonefn(1)

Out[3]: 2
```

```
In [4]: type(plusonefn)

Out[4]: function

In [6]: plusone.__name__

Out[6]: 'plusone'

In [7]: plusonefn.__name__

Out[7]: '<lambda>'
```

Higher-Order Functions

- A higher-order function accepts functions as arguments and can return a function as a result. map(), reduce(), and filter() are examples of higher-order functions. These functions accept a function as an argument.

map()

- The map() function in Python takes in a function and a list as argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item. Example:
- **map(function, iterable)**

```
tempc = [38.4, 19.2, 12.8, 9.6]
tempf = map(lambda x: (float(9)/5)*x + 32, tempc) #converts celcius to fahrenheit
print(tempf)
print(list(tempf)) # casting to list required!!
```

```
<map object at 0x0000018F75282288>
[101.12, 66.56, 55.040000000000006, 49.28]
```

reduce()

- The **reduce(fun,seq)** function is used to apply a particular function passed in its argument to all of the list elements mentioned in the sequence passed along. This function is defined in “functools” module.
- **Working :**
- At first step, first two elements of sequence are picked and the result is obtained.
- Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
- This process continues till no more elements are left in the container.
- The final returned result is returned and printed on console.

reduce()

$[s_1, \quad s_2, \quad s_3, \quad s_4]$

$\mathtt{func}(s_1, s_2)$

$\mathtt{func}(\mathtt{func}(s_1, s_2), s_3)$

$\mathtt{func}(\mathtt{func}(\mathtt{func}(s_1, s_2), s_3), s_4)$

Use of lambda() with reduce()

```
# importing functools for reduce()
import functools

# initializing list
lis = [ 1 , 3, 5, 6, 2]

# using reduce to compute sum of list
print ("The sum of the list elements is : ",end="")
print (functools.reduce(lambda a,b : a+b,lis))

# using reduce to compute maximum element from list
print ("The maximum element of the list is : ",end="")
print (functools.reduce(lambda a,b : a if a > b else b,lis))
```

```
The sum of the list elements is : 17
The maximum element of the list is : 6
```

filter()

- The filter() function returns an iterator were the items are filtered through a function to test if the item is accepted or not.
- **filter(function, sequence)**
- **function:** function that tests if each element of a sequence true or not.
- **sequence:** sequence which needs to be filtered, it can be sets, lists, tuples, or containers of any iterators.
- **returns:** returns an iterator that is already filtered

filter()

```
# function that filters vowels
def fun(variable):
    letters = ['a', 'e', 'i', 'o', 'u']
    if (variable in letters):
        return True
    else:
        return False


# sequence
sequence = ['g', 'e', 'e', 'j', 'k', 's', 'p', 'r']

# using filter function
filtered = filter(fun, sequence)

print('The filtered letters are:')
for s in filtered:
    print(s)
```

```
The filtered letters are:
e
e
```

Use of lambda() with filter()

In [2]:

```
# a list contains both even and odd numbers.
seq = [0, 1, 2, 3, 5, 8, 13]

# result contains odd numbers of the list
result = filter(lambda x: x % 2, seq)
print(list(result))

# result contains even numbers of the list
result = filter(lambda x: x % 2 == 0, seq)
print(list(result))
```

```
[1, 3, 5, 13]
[0, 2, 8]
```

Word Count Example with High Order Functions

```
In [1]: from pyspark import SparkConf
        from pyspark import SparkContext
        conf = SparkConf().setAppName('SparkStructureSlides')
        sc = SparkContext(conf = conf)

        lines = sc.textFile("input.txt")
        counts = lines.flatMap(lambda x: x.split(' ')) \
            .filter(lambda x: len(x) > 0) \
            .map(lambda x: (x, 1)) \
            .reduceByKey(lambda x, y: x + y) \
            .collect()
```

```
In [3]: for (word, count) in counts:
            print("%s: %i" % (word, count))
```

```
The: 6139
Project: 205
EBook: 5
of: 39169
Sir: 30
Arthur: 18
Conan: 3
in: 19512
series: 88
are: 3418
```