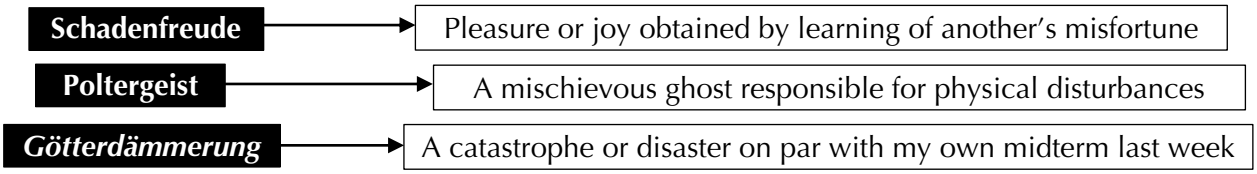
Lecture 16: Dictionaries and Tuples in Python

Written by Ben Yan, from materials by Jerry Cain.

Dictionary Type

Like most of our contemporary programming languages, Python offers a data structure that enables us to associate pairs of data values. Although this type of structure is more commonly referred to as a ***map***, **hash table**, or **hash map**, Python rechristens this data type as a ***dictionary***—evoking the imagery of a handheld English dictionary in the library **where you look up definitions (*values*) by their words (*keys*).**



As such, a ***dictionary*** can be viewed as a **collection of key-value pairs**, like the aggregates and maps we saw in JavaScript. The key is most often a string, but not always (unlike JavaScript!), and can be virtually of any immutable type, e.g., a string, a number, or a ***tuple*** of immutable values—more on that later!

You can create an empty dictionary as follows.

```
dict = {}
```

You can also initialize some key-value pairs when you initialize the dictionary itself, for example,

```
voteDict = {
      "Democratic Party": 47000000,
      "Republican Party": 39000000,
      "Independent": 4000000
}
```



You must explicitly use double quotes (`"`) when declaring string keys, which was optional in JavaScript. **When you look up a value in a dictionary, you supply the key using the square-bracket notation**, as in

```
dict[key]
```

If the **`key`** is defined the dictionary, this selection returns the corresponding value. However, if **`key`** is not defined, Python raises a **`KeyError`** exception, and the program crashes—unlike JavaScript, where **`dict[key]`** quietly resolves to ***undefined***. **Dictionary selections are assignable and modifiable**, e.g.,

```
dict[key] = value
```

sets the value associated with a key, and creates the new key-value pair if the key isn't already defined.

You can iterate through the keys of a dictionary using a **`for`** loop in Python, which has the template:

```
for key in dict:
      value = dict[key]
      … code to work with the individual key and value …
```

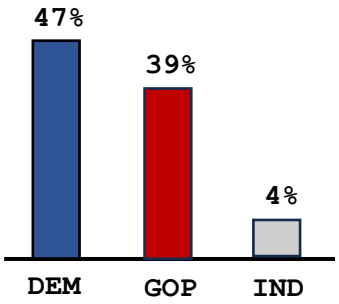
You can also utilize the **`.items()`** method to iterate through keys and values together, as done below.

```
for key, value in dict.items():
    … code to work with the individual key and value …
```

Let's do some dictionary warmups! In the spirit of the upcoming U.S. election day, we write a function **`convertToPercents`** that, given a **`voteDict`** structured like the one on the previous page, **converts all dictionary values, i.e. votes that each party received, to percentages of the total number of votes**.

```
def convertToPercents(voteDict)
    # tabulates the total number of votes across all parties
    totalVotes = 0
    for party in map:
        totalVotes += voteDict[party]

    # divide all votes by total to get percentage
    for party in map:
        voteDict[party] /= totalVotes
```



This **`voteDict`** is passed in by reference, meaning our function modifies its values directly! As such, if our **`voteDict`** was the one declared previously (with **`totalVotes`** = 100 million), it would be altered to:

```
voteDict = {
    "Democratic Party": 0.47,
    "Republican Party": 0.39,
    "Independent": 0.04
}
```

Tuples in Python

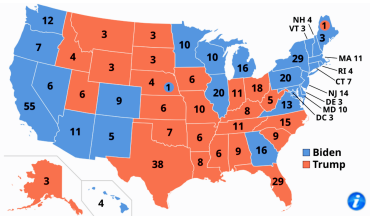
A tuple is a Python type which is akin to a small list, though written with surrounding parentheses, e.g.,

```
PRIMARY_COLORS = ("red", "green", "blue")
TURQUOISE = (64, 224, 208)
```

Unlike a list, a tuple is ***immutable***, meaning its contents cannot be changed once it's created. It's thus useful when you have a fixed collection of things, e.g., the coordinates of a 2D point **`(x,y)`**, or RGB color triplets.

Since tuples are immutable, you can also use them as keys of a dictionary, provided the elements of the tuple key are themselves of immutable type. This is neat as the key can be a composite of information, not just one value. Here's an example that records electoral votes for a presidential candidate, party, and year.

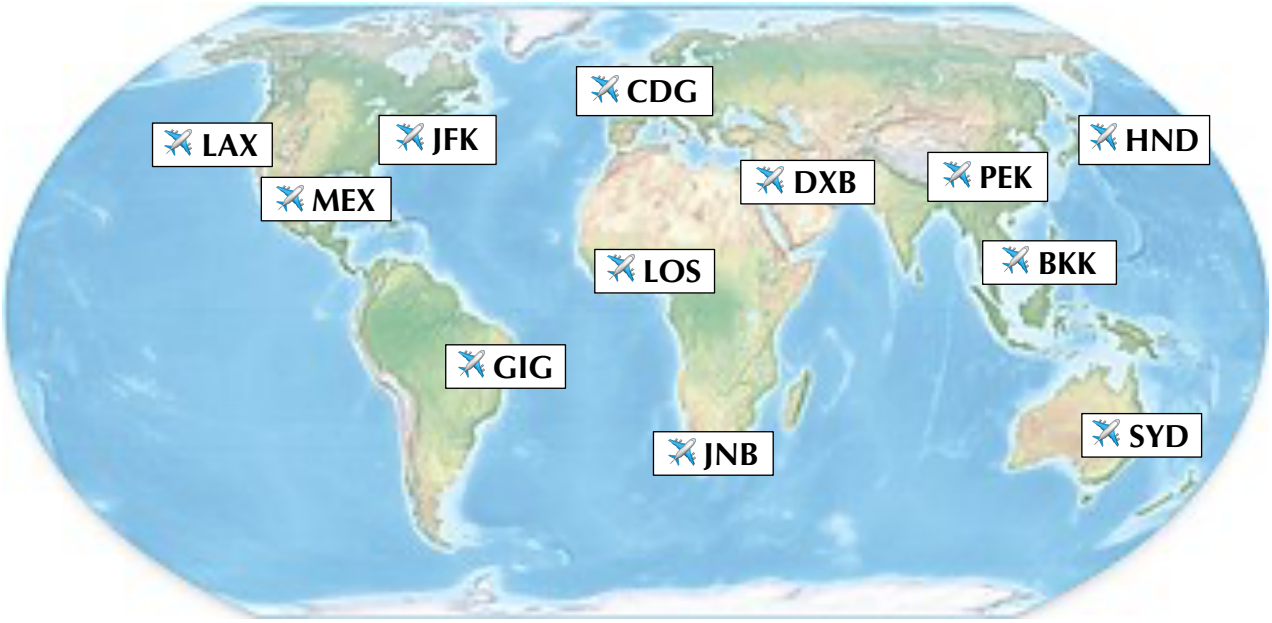
```
voteDict = {
    ("Barack Obama", "DEM", 2012): 332,
    ("Joe Biden", "DEM", 2020): 306,
    ("George H.W. Bush", "GOP", 1988): 426,
    … more entries omitted for conciseness!
}
```



Furthermore, note that tuples, like lists, support indexing, slicing, the **`len`** operation, and loops. The *key* ☺ difference is that they can't be changed! As such, there's no **`append`**, but you can just create a new tuple.

Reading Dictionaries from Files

Dictionaries are often used to store and organize data ingested from external text files. A scintillating example from Eric Robert's textbook uses a dictionary to map 3-letter airport codes to their locations.



The association list—of global airport codes and cities—is stored in a text file that appears like this.

```
ATL: Atlanta, GA, USA
PEK: Beijing, China
DXB: Dubai, United Arab Emirates
LAX: Los Angeles, CA, USA
HND: Tokyo, Japan
ORD: Chicago, IL, USA
LHR: London, England, United Kingdom
HKG: Hong Kong, Hong Kong
PVG: Shanghai, China
CDG: Paris, France
```

Our itinerary for today is to read a text file, e.g., ***AirportCodes.txt***, like the one above, and synthesize a dictionary that maps each airport code to the corresponding city, which can be useful for travel plans.

```
{
      "ATL": "Atlanta, GA, USA",
      "PEK": "Beijing, China",
      "DXB": "Dubai, United Arab Emirates",
      "LAX": "Los Angeles, CA, USA",
      "HND": "Tokyo, Japan",
      ... more entries omitted for conciseness ...
}
```

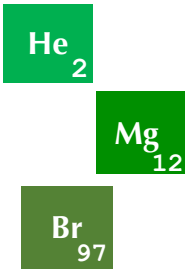
To that end, we'll implement a more generic **`readDictionary`** function, **which given a `filename`, can read any text file storing key-value pairs, where the key & value are divided by some `separator`**, e.g., a colon **`":"`**. The function returns a dictionary mapping each key to its corresponding value, and discards any leading and trailing whitespace from both the key and value as they're written in the file.

For instance, the function should also work on this file mapping each periodic table symbol to its name.

```
He -> Helium
Li -> Lithium
Cu -> Copper
Mg -> Magnesium
K -> Potassium
Bk -> Berkelium
```



```
{
    "He": "Helium",
    "Li": "Lithium",
    "Cu": "Copper",
    "Mg": "Magnesium",
    "K": "Potassium",
    "Br": "Berkelium",
}
```

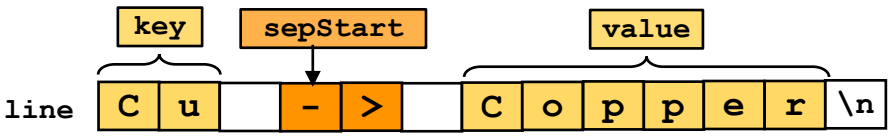


Let's begin! We'll need to first create an empty dictionary, as well as set up the *boilerplate*—code that is rewritten over and over again, evocative of a printing press—to open the file and iterate over each line.

```
dictionary = { }
with open(filename) as file:
    for line in file:
        # TODO: Add code here to parse the line, and update the dictionary!
```

As for what goes inside the **`for`** loop, **we need to find the index of the separator marker**. The key is everything **before** the separator, minus the surrounding whitespace, and the value is everything **after**.

```
sepStart = line.find(separator)
key = line[:sepStart].strip() # removes surrounding whitespace!
value = line[sepStart + len(separator):].strip()
```



Lastly, we add the key-value pair to the dictionary, which can be accomplished with a single line ☺.

```
dictionary[key] = value
```

A full implementation of **`readDictionary`** is implanted below. Note there's a **`return dictionary`** statement at the end, so that the caller function, by providing a file, receives its reward of the dictionary.

```
def readDictionary(filename, separator=":"):
    dictionary = { }

    with open(filename) as f:
        for line in f:
            sepStart = line.find(separator) # index of the separator
            key = line[:index].strip()      # removes surrounding whitespace!
            value = line[index + len(separator):].strip()
            dictionary[key] = value

    return dictionary
```

Searching By Airport Code

One useful application of an **airport code-to-location dictionary**—as pictured on the right—is a program that allows a user to **repeatedly look up different airport codes, and learn the locations they map to**.

Below is an excerpt of a Python script, ***FindAirportsByCode.py***, which does exactly that.

```
{
  "PEK": "Beijing, China",
  "CDG": "Paris, France",
  "HND": "Ota, Tokyo, Japan",
  "MEX": "Mexico City, Mexico",
  more entries omitted for conciseness ...
}
```

```
from dictfile import readDictionary # we can use this method, defined in another file ☺

def FindAirportsByCode():
      airportDictionary = readDictionary("AirportCodes.txt")
      while True:
            code = input("> Enter airport code: ")
            if code == "": break # user enters a blank line, so loop & program is terminated

            searchForAirport(airportDictionary, code)

if __name__ == "__main__": # Main function / starting point of program's execution
     FindAirportsByCode()
```

Note the **readDictionary** method can be stored in a separate script, ***dictfile.py***, for decomposition purposes, but that an **import** statement exists in Python to empower access to another file's methods!

Also, the above implementation relies on a helper function, **searchForAirport**, which looks up the **code** in the **airportDictionary**—but if it isn't there, prints an error message letting the user know.

```
def searchForAirport(code, airportDictionary):
      if code in airportDictionary:
            print(airportDictionary[code])
     else:
            print("There is no airport code " + code)
```

The next time your plane ticket comes with a layover at an unfamiliar airport code, you'll know exactly where it is now. Here's a sample run of this script (I looked up faraway places I'm hoping to visit soon).

Terminal: Find Airport By Code

```
> Enter airport code: PEK
Beijing, China

> Enter airport code: SIN
Changi, East Region, Singapore

> Enter airport code: JERRYCAIN
There is no airport code JERRYCAIN

> Enter airport code:
```

Searching By Airport Location

An alternative, and equally useful, application is to **find all the airports—identified by their code—that match a given location query**, e.g., **`"Beijing"`**, **`"France"`**, **`"England"`**. An airport code is considered a **match** if its location contains the query as a substring.

```
{
  "PEK": "Beijing, China",
  "PKX": "Daxing, Beijing, China",
  "PVG": "Pudong, Shanghai, China",
  "SHA": "Shanghai, China",
  more entries omitted for conciseness ...
}
```

Here's an excerpt of ***FindAirportsByLocation.py***, which performs exactly as advertised. The user is prompted to repeatedly enter location queries, and the program responds with all airports that match.

```
from dictfile import readDictionary

def FindAirportsByLocation():
      airportDictionary = readDictionary("AirportCodes.txt")
      while True:
            query = input("> Enter search string: ")
            if query == "": break # user enters a blank line, so input loop is terminated!

            searchAirports(airportDictionary, query)

if __name__ == "__main__": # "Main" function, or starting point of program execution
    FindAirportsByLocation()
```

Here's the implementation of the helper **`searchAirports`**, which loops through all key-value pairs of **`airportDictionary`**. We print out an airport's code and location, **only if the location string *contains* the user's query**, e.g., if **`query="Japan"`**, then **`location="Chiba, Tokyo, Japan"`** is a match ✓.

```
def searchAirports(airportDictionary, locationQuery):
      for code, location in airportDictionary.items():
            if locationQuery in location: # query appears in this airport's location string
                  print(code + ": " + location)
```

Here's a sample run of this ***FindAirportsByLocation.py*** script, ending with me entering an empty line.

Terminal: Find Airports By Location

```
> Enter search string: Beijing
PEK: Beijing, China
PKX: Daxing, Beijing, China

> Enter search string: Japan
HND: Tokyo, Japan
KIX: Greater Osaka Area, Japan
OKA: Naha, Okinawa Prefecture, Japan
... many more entries omitted for conciseness (there are a lot!)

> Enter search string:
```