More on Dictionaries

Python

The Hungarian Phrase Book

A Teaser on Iterators

- Iterators are the hidden engine of many Python features
  - Iterators are almost like lists
    - You always can get the next element
      - Unless you are at the end of a list
  - But they are not lists:
    - All the elements in the list have to be there before the list can be used
      - They need to be stored in memory
      - Which uses up space
      - And can be disastrous if there are just too many

A Teaser on Iterators

- Iterators are only created when there is a need
- Iterators are often hidden from view
- But we will have to use them
    - For our purposes:
        - We can make them explicitly into lists because we are just not working with millions of data items
        - But hopefully, once we get to play with the grown-ups …
- Seriously, we get back to iterators

Multi-Dictionaries

- Problem:
  - Instead of associating one value with a key, we want to associate several values:
    - a “multi-dictionary”
- Solution:
  - The values of the dictionaries should be lists (or sets — coming week)

Multi-Dictionaries

- Example:
    - We want to pass through a file and create an index of important words with their occurrences

```
with open("alice.txt", encoding = "latin-1") as infile:
    dicc = {}
    word_number = 0
    for line in infile:
        for word in line.split():
            word = word.strip(":,.?![]'")
            word = word.lower()
            word_number +=1
            if len(word)>8:
                if word in dicc:
                    dicc[word].append(word_number)
                else:
                    dicc[word]=[word_number]
```

Calculating on Values

- Assume you have a dictionary with numerical values
  - For example: a dictionary with the prices of stocks on September 15, 2018

```
dstocks = {“tata”: 2063.30,
           “hdfc”: 2029.20,
           “hiul”: 1630.15,
            …
          }
```

  - You want the average, the maximum, the minimum … price

Solution

- You can access the values of a dictionary through the values method.
    - values( ) returns an iterator of all the values in the dictionary

```
>>> dst = {"apple": 256.34, "fb": 145.23, "ibm": 98.34, "ms": 198.75}
>>> dst.values()
dict_values([256.34, 145.23, 98.34, 198.75])
>>> max(dst.values())
256.34
>>> sum(dst.values())/len(dst.values())
174.665
```

Calculating with keys

- Problem:
  - You want to calculate on the keys of a dictionary
- Solution:
  - The keys( ) method returns an iterator of the keys of a dictionary

Finding the most common item in a list

- We use a dictionary as a counter.
  - First way: We can do so by ourselves.
    - Create a dictionary
      - Pass through the list

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        counter[x]=counter.get(x, 0)+1
```

get specifies a default value, it is otherwise equivalent to counter[x]

Finding the most common item in a list

- If we do not want to use get, we can just check whether the list-item is already in the dictionary

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        if x in counter:
            counter[x]+=1
        else:
            counter[x]=1
```

Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
    - Notice that we are interested in the key, not the value

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        counter[x]=counter.get(x, 0)+1
    highest_seen = 0
    for x in counter:
        if counter[x]>highest_seen:
            best_key = x
            highest_seen = counter[x]
    return best_key
```

highest_seen contains the highest encountered **value**

Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
  - Notice that we are interested in the key, not the value

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        counter[x]=counter.get(x, 0)+1
    highest_seen = 0
    for x in counter:
        if counter[x]>highest_seen:
            best_key = x
            highest_seen = counter[x]
    return best_key
```

highest_seen is adjusted whenever we see a higher value in the counter

Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
    - Notice that we are interested in the key, not the value

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        counter[x]=counter.get(x, 0)+1
    highest_seen = 0
    for x in counter:
        if counter[x]>highest_seen:
            best_key = x
            highest_seen = counter[x]
    return best_key
```

but we also need to remember the key, which we record in best_key

Finding the most common item in a list

- After counting, we pass through the dictionary to find the maximum element.
  - Notice that we are interested in the key, not the value

```
def most_frequent(lista):
    counter = {}
    for x in lista:
        counter[x]=counter.get(x, 0)+1
    highest_seen = 0
    for x in counter:
        if counter[x]>highest_seen:
            best_key = x
            highest_seen = counter[x]
    return best_key
```

because the key with the highest counter value is the result that we return

Finding the most common item in a list

- But we can also use the work of others
  - The `Counter` in the collections module
    - You create a *new object* of type Counter

```
from collections import Counter

def most_frequent(lista):
    ctr = Counter()
```

Defines a new object called ctr
ctr is an object of type Counter

Finding the most common item in a list

- Counters are (updated) like dictionaries
  - But they have a default value of 0

```
from collections import Counter

def most_frequent(lista):
     ctr = Counter()
     for item in lista:
          ctr[item] += 1
```

Here we add 1 to the value of `ctr[item]`

No need to initialize!

Finding the most common item in a list

- Counters have a method called `most_common`
  - Argument is the number of most common items
  - Returns a list of pairs

```
from collections import Counter

def most_frequent(lista):
    ctr = Counter()
    for item in lista:
        ctr[item] += 1
    return ctr.most_common(1)[0][0]
```

- **Get a list of one elements.**
- **Get the first (and only) element of the list**
- **Get the first coordinate of that element**

Memoization

- (Some) Computer Scientists love recursion
  - A function calls itself
    - This is super-elegant and the more mathematically inclined pine for this elegance
  - But it is not necessarily very fast
    - The more engineeringly inclined think its a waste

Recursion

- When it works
  - Factorials
    - The factorial of *n* is *n* (*n*-1) (*n*-2) (*n*-3) … (4) (3) (2) (1)
    - Define it to be one for negative or zero *n*

Recursion

- This implementation has the function factorial call itself

```
def factorial(number):
    if number<1:
        return 1
    else:
        return number*factorial(number-1)
```

- **Here we are calling on the function itself**
- **Will call factorial(number-1), which will call factorial(number-2), which will call factorial(number -3) … until we call factorial on 1, in which case the recursion stops.**

Recursion

- This implementation has the function factorial call itself

```
def factorial(number):
    if number<1:
        return 1
    else:
        return number*factorial(number-1)
```

- The base case:
    - We cannot call recursion infinitely often, so we need one.

Recursion

- The Fibonacci numbers
  - The Fibonacci numbers are defined recursively
  - $f_0 = 0, \qquad f_1 = 1, \qquad f_n = f_{n-1} + f_{n-2}$

```
def fibonacci(number):
    if number <= 0:
        return 0
    if number == 1:
        return 1
    return fibonacci(number-1)+fibonacci(number-2)
```

Recursion

- But this implementation is inane!
  - Takes too long even for small numbers.
    - We can use the time-module in order to obtain the cpu-time
      - We do so once before and after execution of the function
    - This yields approximately the time it takes to execute the function

Recursion

- We just write a function that measures the time

```
def measure(function, number):
    start = time.time()
    function(number)
    print(number, time.time()-start)
```

Recursion

- Now we try it out with factorial and fibonacci
  - Not a problem with factorial

```
27 1.52587890625e-05
28 1.5974044799804688e-05
29 1.52587890625e-05
30 1.5735626220703125e-05
31 1.811981201171875e-05
32 1.71661376953125e-05
33 1.7881393432617188e-05
34 1.7881393432617188e-05
35 1.9073486328125e-05
36 1.9788742065429688e-05
37 1.8835067749023438e-05
38 2.09808349609375e-05
39 2.193450927734375e-05
```

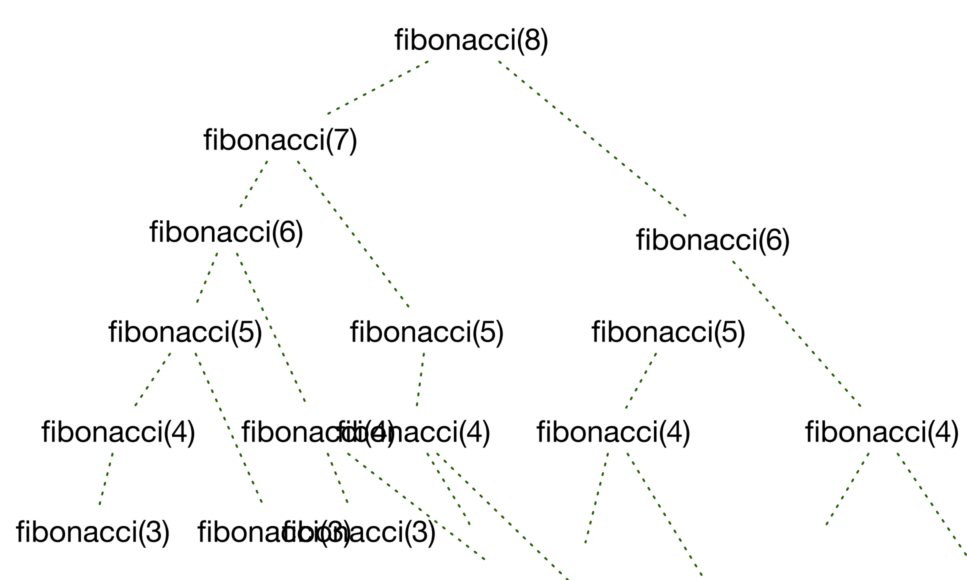
Recursion

- But disastrous for Fibonacci
- It takes 34 seconds in order to calculate fibonacci(39).

```
28 0.17530512809753418
29 0.27112603187561035
30 0.43769311904907227
31 0.7113552093505859
32 1.1374599933624268
33 1.846013069152832
34 2.994562149047851б
35 4.856478929519653
36 7.856333971023 56
37 12.681456804275513
38 20.597038030624 39
39 33.98105502128601
```

Recursion

- What is the problem?
  - Look at what happens if we calculate fibonacci(9).
  - We calculate fibonacci(8) and fibonacci(7)
    - Since the first one also calculates fibonacci(7), we calculate fibonacci(7) twice.
    - And it gets worse for fibonacci(6), fibonacci(5), …



Memoization

- A simple trick to speed up recursive functions is to remember values that we have already calculated.
- Create a dictionary (possibly global) that stores values already calculated
  - Before any calculation check whether the desired value is in the dictionary
  - If we calculate something, we put the value into the dictionary

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- Defining the dictionary

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- Check whether value is in the dictionary

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- **Calculation is necessary**

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- But we store the result in the dictionary in case we use it in the future

Memoization

```
fdic={0: 0, 1:1}

def fibonacci2(number):
    if number in fdic:
        return fdic[number]
    else:
        retval = fibonacci2(number-1)+fibonacci2(number-2)
        fdic[number] = retval
        return retval

for i in range(41):
    measure(fibonacci2, i*50)
```

- And now we measure