CS2110 Lecture 16 March 1, 2021

- DS5 due tonight
- HW4 due Thursday
- Test 2 Friday, in class

Last time

- DS5, HW4, dictionaries, tuples

Today

- Default and optional arguments to functions
- HW4
  - Sorting for HW4
- zip, generators, conditional expressions, and list comprehensions

Default/optional argument values and keyword arguments to functions

Note: although it is used a few places in the book (e.g. 9.18), it's not discussed much. See official Python docs for more info.

With sort/sorted, there are optional arguments that control how the sort works: E.g.

```
>>> myL  = [1,2,3,4,5]
>>> sorted(myL, reverse = True)
[5, 4, 3, 2, 1]
>>> myL = [ ["ABC", 100], ["MNO", 60] , ["XYZ", 80] ]
>>> sorted(myL)
[['abc', 100], ['mno', 60], ['xyz', 80]]
>>> sorted(myL, key = lambda pair: pair[1])
[['mno', 60], ['xyz', 80], ['abc', 100]]
>>> sorted(myL, key = lambda pair: pair[1], reverse = True)
[['abc', 100], ['xyz', 80], ['mno', 60]]
```

The 'key = …' and 'reverse = …' are called keyword arguments and are very convenient.

Default argument values and keyword arguments to functions

```
def foo(a, b = 0, c = True):
    if c == True:
        return a + b
    else:
        return a – b
```

Parameters b and c are optional! They have default values so you don't need to provide more than one argument when calling foo.

```
>>> foo(3)
 3
```

Default argument values and keyword arguments to functions

```
def foo(a, b = 0, c = True):
    if c == True:
        return a + b
    else:
        return a – b
```

```
>>> foo(3,1)
4
>>> foo(3,1,False)
2
```

Default argument values and keyword arguments to functions

```
def foo(a, b = 0, c = True):
    if c == True:
        return a + b
    else:
        return a – 1
```

It sometimes makes code easier to read to use the parameter name when making the function call.

```
>>> foo(3, b=1)
4
>>> foo(3, b=1,c=False)
2
>>> foo(3, c=False, b=1)
2
>>> foo(b=1, c=False, 3)
Error!
>>> foo(c=False, b=1, a = 3)
2
```

Unnamed arguments must come first, before keyword arguments!

And BE CAREFUL. Mutable default arguments are evaluated once, at function definition (different than some other languages). See bar in lec16a.py

HW4

It is interesting, and not hard if you do a little bit at a time. Get it working bit by bit.

1. Read the file, storing all the messages and their labels (spam/ham). E.g.
   - Two separate lists: ham list [[‘text’, ‘me’, ‘later!’], [‘…’, …], ...] and spam list [[‘call’, ‘1412’, ‘to’, ‘win’], …] *(I recommend this option)*
   - Or one list [[‘spam’, [‘call’, ‘1412’, ‘to’, ‘win’]], [‘ham’, [‘text’, ‘me’, ‘later!’]], […], …]
   - Note: don’t keep ham/spam label/tag as part of message. I’ve seen people do this and then write special case code to ignore ‘ham’/’spam’ when processing message words in step 2 below – this can yield errors.
2. Create a ham and a spam dictionary. For each message, extract its words, and update spam or ham dictionary of word counts accordingly
   - for ‘text me later!’ increment ‘text’, ‘me’, ‘later’ entries in ham dict
3. Use the two dictionaries to compute and print some statistics
   - get total spam/ham word counts and unique word counts
   - extract most common words from dictionaries
   - print stats

HW4

1. File has some non-Ascii characters.

   use: open(fileName, encoding = 'utf-8')

2. To break line into tokens – individual elements of a line, learn how to use string **split**

   for line in fileStream:

   lineAsList = line.split()

   lec15split.py

3. get rid of extra stuff "…cool!?" learn how to use string **strip** (and/or lstrip, rstrip)

   - I *strongly recommend against* using replace() method
   - *Don't put "" (empty string) as a word in your dictionaries*

for HW4, sorting is very helpful

Why?

You'll have two dictionaries of the form:
{'free': 23, 'you': 50, 'go': 10, 'zoo': 1}

You'll need to extract words in order from most to least frequent?

sort/sorted "work" on dictionaries but do they do what we want?

```
>>> d = {'free': 23, 'you': 50, 'go': 10, 'zoo': 1}
>>> sorted(d)
['free', 'go', 'you', 'zoo']
```

helpful???

For HW4, sorting is helpful

But suppose we have a list of tuples instead of a dictionary.

tl = [('free', 23), ('you', 50), ('go', 10), ('zoo', 1)]

```
>>> sorted(tl)
[('free', 23), ('go', 10), ('you', 50), ('zoo', 1)]
```

Now helpful? Not very.
sorts based on whole tuple

sorted (and sort) have two useful optional arguments:

key: you provide a little function that to apply to item to generate key to use to sort

reverse: provide True if you want list from largest to smallest instead of default of smallest to largest

For HW4, sorting is helpful

sorted (and sort) have two useful optional arguments:

1) key: a little function that is applied to item to generate key to use to sort

```
>>> tl = [('free', 23), ('you', 50), ('go', 10), ('zoo', 1)]
>>>sorted(tl, key = item1)
>>>sorted(tl, key = lambda item: item[1])
```

if function item1 exists

But don't need to write a separate function. 'lambda' allows you to define an (anonymous) function anywhere

```
[('zoo', 1), ('go', 10), ('free', 23), ('you', 50)]
```

yes - better!

So … now also use the other optional argument - reverse

2) reverse: True if you want list from largest to smallest instead of default of smallest to largest

```
>>> sorted(tl, key = lambda item: item[1], reverse = True)
[('you', 50), ('free', 23), ('go', 10), ('zoo', 1)]
```

That's what we want!

lec16.py

For HW4, sorting is very helpful

How do you use this stuff in HW4?

You will create two dictionaries of word counts - one for ham, and one for spam

And you'll want to extract items with highest counts.

1. Saw (three slides back) that

    sort(dict)

didn't quite give us what we needed

2. Saw (in last two slides) that we can usefully sort list of tuples

So … can you make a list of tuples [… (word, count) …] from a dictionary?

- use list(d.items())
- use list(zip(list(d.keys(), list(d.values()))) what is zip?

It's perfectly fine to do things that way in HW4. It turns out you can also sort the dictionary directly using sorted and keywords key and reverse. Experiment and see if you can figure out how ..

Zip function (not in our text)

zip is a convenient function for “re-packaging” sequences

Given two (or more) sequences, zip returns an **iterator** of tuples, where the i-th tuple contains the i-th element from each of the argument sequences.

zip([1,2,3], [‘a’, ‘b’, ‘c’]) is *like* the tuple of tuples:

((1, ‘a’), (2, ‘b’), (3, ‘c’))

but isn’t exactly that…

```
>>> zippedStuff = zip([1,2,3], [‘a’, ‘b’, ‘c’])
>>> zippedStuff
<zip object at 0x1045c2c48>
```

??? It’s an **iterator**

lec16.py

Quick look at iterators (not required)

zip returns a zip object, which is a kind of **iterator.** Iterators (and generators) are important more advanced Python concepts that you are not required to know. Look up at python.org or sites like w3schools.com

```
>>> zippedStuff = zip([1,2,3], [‘a’, ‘b’, ‘c’])
>>> zippedStuff
<zip object at 0x1045c2c48>
```

What can you do with that zip object?

1) Turn it into a list:

```
>>> zippedStuffList = list(zippedStuff)
>>> zippedStuffList
[(1, 'a'), (2, 'b'), (3, 'c')]
```

common/useful

What can you do with iterator object like result of zip?

```
>>> zippedStuff = zip([1,2,3], [‘a’, ‘b’, ‘c’])
>>> zippedStuff
<zip object at 0x1045c2c48>
```

2) use it with for loops

```
>>> for element in zippedStuff:
        print(element)
(1, ‘a’)
(2, ‘b’)
(3, ‘c’)
```

What can you do with iterator object like result of zip?

```
>>> zippedStuff = zip([1,2,3], [‘a’, ‘b’, ‘c’])
>>> zippedStuff
<zip object at 0x1045c2c48>
```

3) ask for the items one by one

```
>>> next(ZippedStuff)
(1, ‘a’)
>>> next(zippedStuff)
(2, ‘b’)
>>> next(zippedStuff)
(3, ‘c’)
>>> next(zippedStuff)
Traceback (most recent call last):
  File "<pyshell#7>", line 1, in <module>
    next(zippedStuff)
StopIteration
```

next is a special built-in func. for working with iterators

What can you do with iterator object like result of zip?

```
>>> zippedStuff = zip([1,2,3], [‘a’, ‘b’, ‘c’])
```

BUT BE CAREFUL! You can only “use” iterator once!

```
>>> for element in zippedStuff:
        print(element)
(1, ‘a’)
(2, ‘b’)
(3, ‘c’)
>>> list(zippedStuff)
[]
>>> next(zippedStuff)
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    next(zippedStuff)
StopIteration
>>>
```

Nothing’s left!

Nothing’s left!

zip

Typically, we'll use list(zip(…)) or tuple(zip(…))

zipObj = zip([1, 2, 3], ['uno', 'dos', 'tres'], ['mot', 'hai', 'ba'])

zippedList = list(zipObj) (or zippedList = tuple(zipObj)

[(1, 'uno', 'mot'), (2, 'dos', 'hai'), (3, 'tres', 'ba')]

Notice: in example above three lists were zipped (rather than two in earlier examples – can zip any number)

What happens when lists/things being zipped aren't the same length? *Try it* …

Conditional expressions (common but not in text?)

not required for exams but you should understand if you see them

Can be used in place of an if/else where both return some value

```
def oddEven(n):
    if n%2 == 1:
        return 'odd'
    else:
        return 'even'
```

```
def oddEven(n):
    return 'odd' if n%2==1 else 'even'
```

Is this better?

It's *shorter* but I don't find it as readable

They **are** clear and useful sometimes.

lec16.py

Chapter 10.22: list comprehensions

Python provides another shorthand - **list comprehensions** – concise expressions for constructing lists.

Consider common pattern:

```
result = []
for item in someList:
    result.append(someFunc(item))
```

Can do this is one line with list comprehension:

```
result = [someFunc(item) for item in someList]
```

“apply someFunc to each item in someList and gather the results in a new list”

list comprehensions

Examples:

```
>>> [i * i for i in range(5)]
???
[0, 1, 4, 9, 16]
>>> [s.lower() for s in [“Hi”, “Bye”]]
???
['hi', 'bye']
```

Can also use an if:

```
>>> [num for num in [1, -2, 3, -4, 5] if num > 0]
???
[1, 3, 5]
```

lec16.py

list comprehensions

Can also have more than one ‘for’ in a comprehension

```
>>> [(i,j) for i in range(10) for j in range(5)]
```

???

```
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (2, 0), (2, 1), (2, 2), (2, 3), (2, 4), (3, 0), (3, 1), (3, 2), (3, 3), (3, 4), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4)]
```

```
>>> [(i,j) for i in range(10) for j in range(i)]
```

???

```
[(1, 0), (2, 0), (2, 1), (3, 0), (3, 1), (3, 2), (4, 0), (4, 1), (4, 2), (4, 3), (5, 0), (5, 1), (5, 2), (5, 3), (5, 4), (6, 0), (6, 1), (6, 2), (6, 3), (6, 4), (6, 5), (7, 0), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), (7, 6), (8, 0), (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6), (8, 7), (9, 0), (9, 1), (9, 2), (9, 3), (9, 4), (9, 5), (9, 6), (9, 7), (9, 8)]
```

list comprehensions

Example: matrices are common in science and engineering and are often depicted as a table having n rows and m columns. Below is a matrix with 3 rows and 5 columns:

1 12 3 4 5

5 -1 4 1 1

0 -3 4 0 9

It's easy to represent a matrix in Python using a list of sublists, where each sublist represents one row of the matrix. Thus the matrix above would be represented as: [[1, 12, 3, 4, 5], [5, -1, 4, 1, 1], [0, -3, 4, 0, 9]]

If A, B are matrices **CAN** multiply them using list comprehension:

```
def matrixMult(A, B):
    return [ [sum([ A[i][j] * B[j][k] for j in range(len(B))]) for k in
        range(len(B[0])) ] for i in range(len(A))]


def mmatrixMult2(A, B):
    # Multiply row by (transposed w/zip) col
    return([[ sum([ a*b for (a,b) in zip(row,col) ]) for col in zip(*B) ] for row in A ])
```

I don't do this – for me, too concise/dense to be easily readable and clear. I use comprehensions for small simple things.

dictionary comprehensions and generator expressions

There are also dictionary comprehensions:

```
{ i : i*i for i in range(4) }
```

And, there are things that look like comprehensions but are called generator expressions:

```
>>> genSq = (i*i for i in range(4))
```

yields a generator object, which is an iterator, so you can use it with next(…) like we did with zip object.

(List and dictionary comprehensions and generator expressions are not required for this class; you don't need to know them for exams/homework.)

Next Time

- Image editing/manipulation
- Information about test 2