CSCI 134 Fall 2021:
Lambda Sorting & Dictionaries

October 15, 2021

Shikha Singh, 9AM
Jeannie Albrecht, 10AM
Announcements & Logistics

• **No homework** this week!

• **Practice midterm** released on Glow
  • Two versions: with and without solutions
  • Midterm from F18 with slight modifications to fit our syllabus
  • Everything covered **through today’s lecture** is valid exam content

• **Lab 5** will be a short debugging released on Monday
  • Expect most people to finish it during scheduled lab period
  • (If you need more time, you have until Thur Oct 21 at 10pm)

• **Midterm**: Wed Oct 20th: Slots: 6 - 7:30 pm, 8 - 9:30 pm in **TCL 202**

• **Midterm review**: Monday, Oct 18th 7 - 8 pm in **TPL 203**

Do You Have Any Questions?
Last Time

• Discussed new *immutable* sequences: **tuples**
  • All sequence operations apply to tuples
  • Useful for multi-item assignment (argument unpacking)
  • Appropriate when passing data around that should *not* be mutated
• Revisited sorting and default sorting behavior
• Discussed how we can override the default sorting behavior
  • By using `reverse=True`
  • By defining a **key** function
Today’s Plan

• Discuss stable sorting in Python
  • And ways to override it using key function
• Introduce anonymous functions using lambda expressions
• Discuss a new data structure: dictionary
  • Is "unordered" and mutable
  • Ordered/sequential data structures (like lists) aren't appropriate for all use cases
  • For many applications, unordered collections are preferable
Recap: Sorting with a **key** function

- **Motivation:** Suppose we have a list of tuples that we want to sort by something other than the first item.

- **Example:** Suppose we have a list of course tuples, where the first item is the course name, second item is the enrollment cap, and third item is the term (Fall/Spring).

```python
courses = [('CS134', 74, 'Fall'), ('CS136', 60, 'Fall'),
           ('AFR206', 30, 'Spring'), ('ECON233', 30, 'Fall'),
           ('MUS112', 10, 'Fall'), ('STAT200', 50, 'Spring'),
           ('PSYC201', 50, 'Fall'), ('MATH110', 74, 'Spring')]
```

- Say we want to sort these courses by their capacity in **descending** order: courses with higher capacity should come first.

- We can accomplish this by supplying the `sorted()` function with a **key** function that tells it how to compare the tuples to each other.
Sorting with a **key** function

- **Defining a key function explicitly:**
  - We can define an explicit key function that, when given a tuple, returns the parameter we want to sort the tuples with respect to.

```python
def capacity(courseTuple):
    ''' Takes a sequence and returns item at index 1 '''
    return courseTuple[1]
```

- Once we have defined this function, we can pass it as a **key** when calling `sorted()`.
- Let us try this example in the notebook.
Python Sorting is Stable

• Python's sorting functions are **stable**
  • Items that are “equal” according to the sorting key have the same relative order as in the original (unsorted) sequence

In [1]: courses = [('CS134', 74, 'Fall'), ('CS136', 60, 'Fall'), ('AFR206', 30, 'Spring'), ('ECON233', 30, 'Fall'), ('MUS112', 10, 'Fall'), ('STAT200', 50, 'Spring'), ('PSYC201', 50, 'Fall'), ('MATH110', 74, 'Spring')]

In [2]: def term(courseTuple):
   
   
   """ Takes a sequence and returns item at index 2""
   
   return courseTuple[2]

In [3]: sorted(courses, key=term)

Out[3]: [('CS134', 74, 'Fall'), ('CS136', 60, 'Fall'), ('ECON233', 30, 'Fall'), ('MUS112', 10, 'Fall'), ('PSYC201', 50, 'Fall'), ('AFR206', 30, 'Spring'), ('STAT200', 50, 'Spring'), ('MATH110', 74, 'Spring')]

Notice the ordering of courses with Fall term and those with Spring term
Breaking Ties using **key**

- We can override this default behavior and tell `sorted` how to break ties by supplying a key that returns a tuple.

```python
In [4]: def termAndCap(courseTuple):
    ...:     return courseTuple[2], courseTuple[1]

In [5]: courses = [tuple(x.split(', ')) for x in
               ['MUS112', 10, 'Fall',
               'ECON233', 30, 'Fall',
               'PSYC201', 50, 'Fall',
               'CS136', 60, 'Fall',
               'CS134', 74, 'Fall',
               'AFR206', 30, 'Spring',
               'STAT200', 50, 'Spring',
               'MATH110', 74, 'Spring']]

In [5]: courses
Out[5]: [("MUS112", 10, "Fall"),
         ("ECON233", 30, "Fall"),
         ("PSYC201", 50, "Fall"),
         ("CS136", 60, "Fall"),
         ("CS134", 74, "Fall"),
         ("AFR206", 30, "Spring"),
         ("STAT200", 50, "Spring"),
         ("MATH110", 74, "Spring")]

In [5]: sorted(courses, key=termAndCap)
Out[5]: [("MUS112", 10, "Fall"),
         ("ECON233", 30, "Fall"),
         ("CS134", 74, "Fall"),
         ("CS136", 60, "Fall"),
         ("AFR206", 30, "Spring"),
         ("STAT200", 50, "Spring"),
         ("MATH110", 74, "Spring"),
         ("PSYC201", 50, "Fall")]
```

Notice that now the ties are broken in favor of capacity.

Lambda Notation

- It is often inconvenient to define a named function just in order to pass it as the functional argument to higher-order functions like `sorted()`
- Python provides **lambda** notation for creating **anonymous functions** (or functions with no names that cannot be called elsewhere) that can be used directly in functions like `sorted()`

---

**Right side:** Anonymous versions of left side functions using **lambda notation**

---

```
In [1]: def square(x):
    ...:     return x**x

In [2]: square(5)
Out[2]: 25

In [7]: def first(seq):
    ...:     return seq[0]
    ...:     first('zorp')
Out[7]: 'z'

In [3]: (lambda x: x**x)(5)
Out[3]: 25

In [8]: (lambda seq: seq[0])('zorp')
Out[8]: 'z'
```
Lambda Expression Syntax

A `lambda` expression has the form:

```
lambda param: bodyExpression
```

- **Keyword meaning**: "I am a function"
- **Parameter name**: of this function
- **Expression for result of this function**: It does **not** use an explicit `return`!

---

**Example 1**:

```
lambda num: num*2
```

- **I am a function**
- **That takes an argument named num**
- **And returns the result of doubling it**

---

**Example 2**:

```
lambda n: (n%2)==0
```

- **I am a function**
- **That takes an argument named n**
- **And returns a boolean that indicates whether it’s even**

Image Source: (http://cs111.wellesley.edu/spring19)
In the 1930s and 40s, **Alonzo Church** developed a model of computation called the **lambda calculus**. It is a programming language with only three kinds of expressions:

- variables, e.g. \( x \)
- functions expressed in lambda notation, e.g. \( \lambda x . x \)
- function application, e.g., \( (\lambda x . x)(y) \)

Remarkably, this simple language can express any computable program – even though it has no built-in numbers, arithmetic, booleans, conditionals, lists, loops, or recursion!

To find out more, take CS 334 or CS 361!
Sorting with Lambda

• Instead of defining an explicit named key function, we can use lambda notation to define an anonymous key function to use in sorted()

In [1]: # back to our example
courses = [('CS134', 74, 'Fall'), ('CS136', 60, 'Fall'),
            ('AFR206', 30, 'Spring'), ('ECON233', 30, 'Fall'),
            ('MUS112', 10, 'Fall'), ('STAT200', 50, 'Spring'),
            ('PSYC201', 50, 'Fall'), ('MATH110', 74, 'Spring')]

In [2]: # sort by capacity
sorted(courses, key=lambda course: course[1])

Out[2]: [('MUS112', 10, 'Fall'),
         ('AFR206', 30, 'Spring'),
         ('ECON233', 30, 'Fall'),
         ('STAT200', 50, 'Spring'),
         ('PSYC201', 50, 'Fall'),
         ('CS136', 60, 'Fall'),
         ('CS134', 74, 'Fall'),
         ('MATH110', 74, 'Spring')]
Sorting with Lambda

- Instead of defining an explicit named `key` function, we can use lambda notation to define an anonymous key function to use in `sorted()`

```python
In [1]: # back to our example
courses = [('CS134', 74, 'Fall'), ('CS136', 60, 'Fall'),
           ('AFR206', 30, 'Spring'), ('ECON233', 30, 'Fall'),
           ('MUS112', 10, 'Fall'), ('STAT200', 50, 'Spring'),
           ('PSYC201', 50, 'Fall'), ('MATH110', 74, 'Spring')]
```

```python
In [3]: # sort by term followed by capacity
sorted(courses, key=lambda c: (c[2], c[1]))
```

```python
Out[3]: [('MUS112', 10, 'Fall'),
         ('ECON233', 30, 'Fall'),
         ('PSYC201', 50, 'Fall'),
         ('CS136', 60, 'Fall'),
         ('CS134', 74, 'Fall'),
         ('AFR206', 30, 'Spring'),
         ('STAT200', 50, 'Spring'),
         ('MATH110', 74, 'Spring')]
```
More Sorting with Lambda

• Lambda notation can be applied to `.sort()` list method as well

• Can also be applied to other sequences such as strings

```python
In [1]: zipCodes = [111231, 111777, 11782, 11345, 23114, 455621]

In [2]: zipCodes.sort(key=lambda n: str(n)[-1])
   ...:     zipCodes

Out[2]: [111231, 455621, 11782, 23114, 11345, 111777]

In [3]: ids = ['id1', 'id100', 'id2', 'id22', 'id3', 'id30']

In [4]: sortedIds = sorted(ids, key=lambda x: int(x[2:]八大以来)

In [5]: sortedIds  # can you guess the output?

Out[5]: ['id1', 'id2', 'id3', 'id22', 'id30', 'id100']

In [6]: name = "SquID GamE"
   ...:     sorted(name, key=lambda x:x.lower())  # sort but ignore case

Out[6]: [' ', 'a', 'D', 'E', 'G', 'i', 'm', 'q', 'S', 'u']
**Tuples and Lambda Takeaways**

- Tuples are immutable sequences that
  - support all sequence operations such as indexing and slicing
  - are useful for argument unpacking, multiple assignments
  - are useful for handling data without aliasing issues

- `sorted()` function and `.sort()` method, by default, sorts sequences in ascending and lexicographic order
  - We can override the default sorting behavior by supplying optional parameters `key` (function), and `reverse` (Boolean)

- **Lambda expressions** are a handy way to define anonymous functions
  - useful for one-time use in higher-order functions like sorted
Sequences vs Unordered Collections

- **Sequence**: a group of items that come one after the other (there is an *ordering* of items)
  - Sequences in Python: strings, lists, tuples, ranges
- **Unordered Collection**: a group of things bundled together for a reason but without a specific ordering
- Maintaining order between items is not always necessary
  - Ordering items comes at a cost in terms of efficiency!
- For some use cases, it is better to store an unordered collection
- Python has two data structures which are unordered:
  - **Dictionaries and sets**: both of them are *mutable*
  - We will discuss dictionaries today
Dictionaries

- A **dictionary** is a **mutable** collection that maps **keys** to **values**
- Enclosed with curly brackets, and contains comma-separated items
- An item in the dictionary pair is a **colon-separated key, value pair**.
- There is no ordering between the keys of a dictionary!

```python
# sample dictionary
zipCodes = {'01267': 'Williamstown', '60606': 'Chicago',
            '48202': 'Detroit', '97210': 'Portland'}
```

- **Keys** must be an **immutable** type such as ints, strings, or tuples
- Keys of a dictionary must be **unique**: no duplicates allowed!
- **Values** can any Python object (numbers, strings, lists, tuples, etc.)
Accessing Items in a Dictionary

- Dictionaries are unordered so we cannot index into them: no notion of first or second item, etc.
- We access a dictionary using its keys as the subscript
  - If the key exists, its corresponding value is returned
  - If the key does not exist, it leads to a **KeyError**

```python
In [1]: # sample dictionary
    zipCodes = {'01267': 'Williamstown', '60606': 'Chicago',
                 '48202': 'Detroit', '97210': 'Portland'}

In [2]: # what US city has this zip code?
    zipCodes[60606]

Out[2]: 'Chicago'

value associated with key '60606'

In [3]: # what US city has this zip code?
    zipCodes[48202]

Out[3]: 'Detroit'
```
Adding a Key, Value Pair

- Dictionaries are mutable, so we can add items or remove items from it.
- To add a new key, value pair, we can simply assign the key to the value using: `dictName[key] = value`.

```python
In [5]: zipCodes['11777'] = 'Port Jefferson'
```

```plaintext
Out[6]: {'01267': 'Williamstown',
         '60606': 'Chicago',
         '48202': 'Detroit',
         '97210': 'Portland',
         '11777': 'Port Jefferson'}
```

- If the key already exists, an assignment operation as above will *overwrite* its value and assign it the new value.
Operations on Dictionaries

- Just like sequences, we can use the `len()` function on dictionaries to find out the number of keys it contains.
- To check if a key exists (or does not exist) in a dictionary, we can use the `in` (not `in`) operator respectively.

```python
In [6]: zipCodes
Out[6]: {'01267': 'Williamstown',
         '60606': 'Chicago',
         '48202': 'Detroit',
         '97210': 'Portland',
         '11777': 'Port Jefferson'}

In [7]: len(zipCodes)
Out[7]: 5

In [8]: '90210' in zipCodes
Out[8]: False

In [9]: '01267' in zipCodes
Out[9]: True
```

Should always check if a key exists before accessing its value in a dictionary.
Creating Dictionaries

• Several ways to create dictionaries:
  
  • **Direct assignment**: provide key, value pairs delimited with `{ }`
  
  • Start with empty dict and add key, value pairs
    
    • Empty dict is `{}` or `dict()`
  
  • Apply the built-in function `dict()` to a list of tuples

In [1]:

```python
# direct assignment
scrabbleScore = {'a':1, 'b':3, 'c':3, 'd':2, 'e':1,
                 'f':4, 'g':2, 'h':4, 'i':1, 'j':8,
                 'k':5, 'l':1, 'm':3, 'n':1, 'o':1,
                 'p':3, 'q':10, 'r':1, 's':1, 't':1,
                 'u':1, 'v':8, 'w':4, 'x':8, 'y':4, 'z': 10}
```

**Note**: keys may be listed in any order
Creating Dictionaries

• Direct assignment: provide key, value pairs delimited with `{ }`
• Start with empty dict and add key, value pairs
  • Empty dict is `{}` or `dict()`
• Apply the built-in function `dict()` to a list of tuples

```
In [2]:
# accumulate in a dictionary
verse = "let it be,let it be,let it be,let it be,there will be an answer,let it be"
counts = {} # empty dictionary
for line in verse.split(', '):
    if line not in counts:
        counts[line] = 1 # initialize count
    else:
        counts[line] += 1 # update count

counts
```

```
Out[2]: {'let it be': 5, 'there will be an answer': 1}
```

```
In [3]:
# use dict() function
dict([(\'a\', 5), (\'b\', 7), (\'c\', 10)])
```

```
Out[3]: {\'a\': 5, \'b\': 7, \'c\': 10}
```

**Note:** keys may be listed in any order
Example: `frequency`

- Lets write a function `frequency` that takes as input a list of words `wordList` and returns a dictionary `freqDict` with the unique words in `wordList` as keys, and their number of occurrences in `wordList` as values.

- For example if `wordList` is

  ```
  ['hello', 'world', 'hello', 'earth', 'hello', 'earth']
  ```

  the function should return a dictionary with the following items

  ```
  {'hello': 3, 'world':1, 'earth': 2}
  ```
Example: frequency

- Let's write a function `frequency` that takes as input a list of words `wordList` and returns a dictionary `freqDict` with the unique words in `wordList` as keys, and their number of occurrences in `wordList` as values.

```python
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1  # add key with count 1
        else:
            freqDict[word] += 1  # update count
    return freqDict
```
Important Dictionary Method: `.get()`

- The following code pattern is extremely common when using dictionaries:

```python
if aKey is not in myDict:
    myDict[aKey] = initVal  # add key
else:  # if already exists
    myDict[aKey] += step  # update val
```

- Instead of using `if, else` to do above, it is preferable to use the `.get()` method for dictionaries instead
Important Dictionary Method: `.get()`

- `get()` method is an alternative to using subscript to get the value associated with a key in a dictionary without checking for its existence.
- It takes two arguments: a key, and an optional default value to use if the key is not in the dictionary.
- It returns the value associated with the given key, and if key does not exist it returns the default value (if given), otherwise returns `None`.
- Syntax: `val = myDict.get(aKey, defaultVal)`
Important Dictionary Method: `.get()`

- `.get()` method **does not modify the dictionary** it is called on

```python
In [46]: ids = {'ss32': 'Shikha', 'jral': 'Jeannie', 'kas10': 'Kelly', 'lpd2': 'Lida'}

In [53]: ids.get('kas10', 'Ephelia')
Out[53]: 'Kelly'

In [54]: ids.get('srm2', 'Ephelia')
Out[54]: 'Ephelia'

In [49]: ids # .get does not change the dictionary
Out[49]: {'ss32': 'Shikha', 'jral': 'Jeannie', 'kas10': 'Kelly', 'lpd2': 'Lida'}

In [50]: print(ids.get('ks123'))
None
```
Important Dictionary Method: `.get()`

- We can now simplify this considerably using get:

```python
if aKey is not in myDict:
    myDict[aKey] = initVal # add key
else: # if already exists
    myDict[aKey] += step # update val
```

- Simpler, more efficient, and preferred alternative:

```python
myDict[aKey] = myDict.get(aKey, initVal) + step
```
Example: frequency Improved

- Let's rewrite `frequency` function using `.get()` instead of `if else`.

```python
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {}  # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1  # add key with count 1
        else:
            freqDict[word] += 1  # update count
    return freqDict
```

- What should we write instead inside the for loop?
Example: `frequency` Improved

• Let's rewrite `frequency` function using `.get()` instead of `if` else

```python
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {} # initialize accumulator as empty dict
    for word in wordList:
        if word not in freqDict:
            freqDict[word] = 1 # add key with count 1
        else:
            freqDict[word] += 1 # update count
    return freqDict
```

• What should we write instead inside the for loop?

```python
def frequency(wordList):
    """Given a list of words, returns a dictionary of word frequencies""
    freqDict = {} # initialize accumulator as empty dict
    for word in wordList:
        # what should we write instead?
        freqDict[word] = freqDict.get(word, 0) + 1
    return freqDict
```
Benefits of Dictionaries

• Dictionaries can be a more efficient alternative to lists for some operations

• When we insert into an ordered sequence like a list
  • We need to "move over" all elements to make space
  • This is an expensive operation: worst case (insert at beginning of list) takes time proportional to number of items stored in list

• When we search for an item in an list:
  • If we are not careful we might have to compare to every item stored

• Using a dictionary instead of a list means:
  • Can insert more efficiently (without having to move any other item)
  • Can support more efficient queries on average (if keys are "hashes" of values)

• To learn more about about efficiency of data structures, take CS136/CS256!
Next Time

• More with dictionaries:
  • Iterating over a dictionary
  • Dictionary methods

• Plotting with matplotlib in Python
  • Line plots, bar plots, etc

• Using dictionaries and plotting we will:
  • Find out the most frequent words in Pride and Prejudice
  • Plot their frequency in a bar plot