Sorting, lambda, map & filter

CS111 Computer Programming
Department of Computer Science
Wellesley College

The built-in function `sorted` creates a new list where items are ordered in ascending order.

```
In [1]: numbers = [35, -2, 17, -9, 0, 12, 19]
In [2]: sorted(numbers)
Out[2]: [-9, -2, 0, 12, 17, 19, 35]  # ascending order
```

```
In [3]: numbers
Out[3]: [35, -2, 17, -9, 0, 12, 19]  # original list unchanged
```

```
In [4]: sorted(numbers, reverse=True)
Out[4]: [35, 19, 17, 12, 0, -2, -9]  # descending order
```

**Sorting list of numbers**

**Concepts in this slide:**
- The built-in function `sorted` for sorting lists.

**To notice:**
- The function `sorted` creates a new list and doesn't modify the original list.
- The function `sorted` can take more than one parameter. For example, in In[4] it's taking `reverse=True` in addition to the list to sort.

We can apply the function `sorted` to other sequences too: strings and tuples. Similarly to sorting lists, `sorted` will again create a new list of the sorted elements.

```
In [5]: phrase = 'Red Code 1'
In [6]: sorted(phrase)
Out[6]: [' ', ' ', '1', 'C', 'd', 'd', 'e', 'e', 'o']
```

```
In [7]: phrase
Out[7]: 'Red Code 1'  # original phrase doesn't change
```

```
In [8]: digits = (9, 7, 5, 3, 1)  # this is a tuple
In [9]: type(digits)
Out[9]: tuple
```

```
In [10]: sorted(digits)
Out[10]: [1, 3, 5, 7, 9]
```

```
Question:
Can you explain the order of characters in Out[6]? Do you remember ASCII from this slide?
```

**Sorting a list of strings**

```
In [11]: phrase = "99 red balloons *floating* in the Summer sky"
In [12]: words = phrase.split()
In [13]: words
Out[13]: ['99', 'red', 'balloons', '*floating*', 'in', 'the', 'Summer', 'sky']
```

```
In [14]: sorted(words)
Out[14]: ['*floating*', '99', 'Summer', 'balloons', 'in', 'red', 'sky', 'the']
```

```
In [15]: sorted(words, reverse=True)
Out[15]: ['the', 'sky', 'red', 'in', 'balloons', 'Summer', '99', '*floating*']
```

**To notice:**
- String characters are ordered by these rules:
  a) Punctuation symbols (.,.;,*!#^)
  b) Digits
  c) Uppercase letters
  d) Lowercase letters
Sorting a list of tuples

In [16]: triples = [(8, 'a', '$'), (7, 'c', '@'), (7, 'b', '+'), (8, 'a', '!')]
In [17]: sorted(triples)
Out[17]: [(7, 'b', '+'), (7, 'c', '@'), (8, 'a', '!'), (8, 'a', '$')]

To notice:
If a tuple is composed of several items, the sorting of the list of tuples works like this:

a) Sort tuples by first item of each tuple.
b) If there is a tie (e.g., two tuples with 7), compare the second item.
c) If the second item is also the same, look to the next item, and so on.

Issue: Sorting starts always with the item at index 0. What if we want to sort by items in the other indices?

Easier than SSR:
Sorting with the key parameter

The function sorted takes several parameters, which we can find by typing help in the Canopy console:

In [18]: help(sorted)
Help on built-in function sorted in module _builtin__:
   sorted(...)
   sorted(Iterable, cmp=None, key=None, reverse=False) -> new sorted list

Notice that the first parameter is an “iterable”, meaning any object over which we can iterate (list, string, tuple). We have already seen the parameter reverse and now we’ll see key.

This specifies a function that for each element determines how it should be compared to other elements.

def age(personTuple):
    return personTuple[1]

More examples
Similar to function age, we can create a function to return the last name or the length of the name and sort by them.

Swap-Sort-Restore (SSR)

Problem: We have a list of tuples and want to sort by the second item.

In [18]: people = [('Mary Beth Jay', 18), ('Ed Smith', 17), ('Janet Doe', 25), ('Bob Miller', 31)]

# Step 1: Swap
In [19]: agesFirst = [(age, name) for name, age in people]
In [20]: agesFirst
Out[20]: [(18, 'Mary Beth Jay'), (17, 'Ed Smith'), (25, 'Janet Doe'), (31, 'Bob Miller')]

# Step 2: Sort
In [21]: agesFirstSorted = sorted(agesFirst)
In [22]: agesFirstSorted
Out[22]: [(17, 'Ed Smith'), (18, 'Mary Beth Jay'), (25, 'Janet Doe'), (31, 'Bob Miller')]

# Step 3: Restore
In [23]: sortedByAge = [(name, age) for age, name in agesFirstSorted]
In [24]: sortedByAge
Out[24]: [('Ed Smith', 17), ('Mary Beth Jay', 18), ('Janet Doe', 25), ('Bob Miller', 31)]

Breaking ties with key functions

The people2 list has many ambiguities due to first names, last names, and ages that are the same:

people2 = [('Ed Jones', 18), ('Ana Doe', 25), ('Ed Doe', 18), ('Bob Doe', 25), ('Ana Jones', 18)]

We define ageLastFirst to be a key function that will first sort by age, then by last name (if ages are equal), then by first name (if age and last name are equal).

def ageLastFirst(person):
    return (age(person), lastName(person), firstName(person))

In [36]: sorted(people2, key=ageLastFirst)
Out[36]: [('Ed Doe', 18), ('Ana Jones', 18), ('Ed Jones', 18), ('Ana Doe', 25), ('Bob Doe', 25)]

Note:
The functions lastName and firstName in the definition of ageLastFirst will be created by you in the Notebook during class.
**Lambda notation creates anonymous functions**

It is often inconvenient to define a named function just in order to pass it as the functional argument to higher-order functions (HOF) like `sorted`. (We will see other examples of these functions later in the slides).

Python provides **lambda notation** for creating an **anonymous function** (a function that doesn't have a name and cannot be called anywhere else) which can be used directly with HOFs without introducing named functions.

Here is an example that replaces the function `age` from slide 13-8 with a lambda expression:

```
In [37]: sorted(people, key=lambda personTuple: personTuple[1])
Out[37]: [('Ed Smith', 17), ('Mary Beth Johnson', 18), ('Janet Doe', 25), ('Bob Miller', 31)]
```

**Concepts in this slide:**
- Introducing lambda, a construct to create anonymous functions.

---

**Anatomy of a lambda expression**

A **lambda** expression has the form:

```
lambda param: bodyExpression
```

Keyword meaning

<table>
<thead>
<tr>
<th>parameter name</th>
<th>expression for result of this function.</th>
</tr>
</thead>
<tbody>
<tr>
<td>“I am a function”</td>
<td>It does not use an explicit <strong>return</strong>!</td>
</tr>
</tbody>
</table>

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

- I am a function that takes an argument named `num` and returns the result of doubling it.

```
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.

---

**Why lambda?**

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. the identity function `\( \lambda x . x \)`
- function application, e.g. `(\( x \). 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! (Take CS235 & CS251)
Exercises: **lambda** and Sorting

For each of the following expressions, predict the order of the sorted results:

- `sorted(people, key=lambda person: len(lastName(person)))`
- `sorted(people+people2, key=lambda p: (len(firstName(p)+lastName(p)), age(p)))`
- `sorted(people+people2, key=lambda x: (lastName(x)[-1], firstName(x)[1]))`

Below, replace the `??` with a `lambda` to sort `people2` in ascending order first by age, then by length of last name, then by first name.

`sorted(people2, key=??)`

List methods for sorting

Lists have two methods for sorting. These methods **mutate** the original list. They are `sort` and `reverse`.

- `In [60]: numbers = [35, -2, 17, -9, 0, 12, 19]`
- `In [61]: numbers.sort() # Mutates list; nothing is returned`
- `In [62]: numbers`  
  `Out[62]: [-9, -2, 0, 12, 17, 19, 35]`
- `In [63]: numbers2 = [35, -2, 17, -9, 0, 12, 19]`
- `In [64]: numbers2.reverse() # Mutates list; nothing is returned`
- `In [65]: numbers2`  
  `Out[65]: [19, 12, 0, -9, 17, -2, 35] # no sorting`
- `In [66]: numbers2.sort()`  
  `In [67]: numbers2.reverse()`  
  `In [68]: numbers2`  
  `Out[68]: [35, 19, 17, 12, 0, -2, -9]`

Higher Order Functions

Higher-order functions (HOFs) are:

- Functions that take at least some **parameters** that are themselves functions
- Functions that return functions as results

Examples of built-in HOFs in Python are:

- `sorted`
- `.sort` method (can take a **key** argument, like `sorted`)
- `map`
- `filter`
- `reduce`

**Note:**
The listed functions are all functions that take as parameters other functions (we already saw how `sorted` takes functions as values for the key parameter). In this class we will not cover functions that return other functions. Take CS 251 to learn about such functions.

Higher order list operations: `map`, `filter`

- **Map pattern:**
  - List comprehension: `[str(x) for x in numList]`
  - `map` function: `map(str, numList)`

- **Filter pattern:**
  - `def` `isEven(x):`
    - `return x%2==0`
  - List comprehension: `[x for x in numList if isEven(x)]`
  - `filter` function: `filter(isEven, numList)`

- Why use `map` and `filter` instead of list comprehensions?
  - Although `map` and `filter` can always be expressed with list comprehensions, most other higher-order functions cannot
  - Good practice with higher order functions
**Review: the list mapping pattern**

In the list mapping pattern:
- the output list is the same length as the input list;
- each element of the output list is the result of applying some function to the corresponding element of the input list.

\[
\text{mapDouble( } [8, 3, 6, 7, 2, 4] \text{ )}
\]
\[
\begin{array}{cccccc}
+2 & +2 & +2 & +2 & +2 & +2 \\
\hline
16 & 6 & 12 & 14 & 4 & 8 \\
\end{array}
\]

\[
\text{mapPluralize( } ['\text{donut}', '\text{muffin}', '\text{bagel}] \text{ )}
\]
\[
\begin{array}{c}
+s \\
\hline
['\text{donuts}', '\text{muffins}', '\text{bagels}']
\end{array}
\]

---

**map captures the list mapping pattern**

Python provides a `map` function that captures this pattern. When invoked as `map(function, elements)`, it returns a new output list that's the same length as `elements` in which every element is the result of applying `function` to the corresponding element of `elements`.

\[
\text{map} (f, [e1, e2, \ldots, en])
\]
\[
\begin{array}{ccc}
f & f & f \\
\hline
f(e1), f(e2), \ldots, f(en) \\
\end{array}
\]

---

**Using map on non-list sequences**

The `map` function can be used on any sequence, but always returns a list.

```
In [44]: map(lambda s: s.upper(), 'foo')
Out[44]: ['F', 'O', 'O']
```

```
In [45]: map(lambda s: s.upper(), ('ant', 'bat', 'cat'))
Out[45]: ['ANT', 'BAT', 'CAT']
```

For mapping over each letter of a string, we can get a string from the resulting list of strings by using the `join` method.

```
In [46]: ''.join(map(lambda s: s.upper(), 'foo'))
Out[46]: 'FOO'
```

---

**Mapping over multiple lists [1]**

The `map` function allows any number of list arguments as long as the supplied function takes a number of arguments that's the same as the number of lists.

```
In [47]: map(lambda a,b: a+b, [8,3,5], [10,20,30])
Out[47]: [18, 23, 35]
```

```
In [48]: map(lambda a,b: a*b, [8,3,5], [10,20,30])
Out[48]: [80, 60, 150]
```

```
In [49]: map(lambda a,b: (a,b), [8,3,5], [10,20,30])
Out[49]: [(8, 10), (3, 20), (5, 30)]
```

```
In [50]: map(lambda a,b,c: a+b+c, [8,3], [10,20], [7,2])
Out[50]: [87, 62]
```
Mapping over multiple lists [2]

When mapping over multiple lists, all the lists must have the same length; if not, an exception will be raised.

In [51]: map(lambda a,b: a+b, [8,3,5,6], [10,20,30])
--- Traceback (most recent call last)
<ipython-input-93-e73284726> in <module>()
----> 1 map(lambda a,b: a+b, [8,3,5,6], [10,20,30])

TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'

Concepts in this slide:
- map with many parameters performs tuple assignment, every variable gets one item of each list iteratively.

Review: the list filtering pattern

In the list filtering pattern, the output list includes only those input elements for which a certain predicate is True.

filterEvens( [8, 3, 6, 7, 2, 4])

<table>
<thead>
<tr>
<th>isEven</th>
<th>isEven</th>
<th>isEven</th>
<th>isEven</th>
<th>isEven</th>
<th>isEven</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>False</td>
<td>True</td>
<td>False</td>
<td>True</td>
<td>True</td>
</tr>
</tbody>
</table>

[8, 6, 2, 4]

Concepts in this slide:
- filter behaves differently from map with non-list sequences.

filter captures the list filtering pattern

Python provides a filter function that captures this pattern. When invoked as filter(pred, elts), it returns a new output list contains only those elements in elts for which the predicate pred returns True (in the same relative order).

filter(pred, [e1, e2, ..., en])

<table>
<thead>
<tr>
<th>pred</th>
<th>pred</th>
<th>pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool1</td>
<td>bool2</td>
<td>booln</td>
</tr>
</tbody>
</table>

Using filter on non-list sequences

The output of filter is the same type of sequence as the input.

In [51]: filter(lambda c: c not in 'aeiouy', 'facetiously')
Out[51]: 'fctsl'

In [52]: filter(lambda n: (n%2)==0, (1,2,3,4,5))
Out[52]: (2, 4) # A tuple, not a list
Limitations of **lambda**

*lambda* bodies must be a single expression.

They can't be statement (such as *print* statements, *if* statements) or sequences of statements.

For these, *map* still needs helper functions!

```python
def printDouble(num):
    print num, ',', 2*num

In [19]: map(printDouble, [8, 3, 5])
Out[19]: [8, 3, 5]
```

```python
def myAbs(num):
    if num < 0:
        return -num
    else:
        return num

In [20]: map(myAbs, [-7, 2, -6])
Out[20]: [7, 2, 6]
```

**Summary**

1. Sorting is one of the most common activities that we humans perform. This applies to software-related activities as well: sorting files in your computer by name, by date, by type; sorting students by section, by last name, by class year, by grade; sorting courses in the course browser by department; day of week, distributions, class size, time of day, etc.

2. Python offers a versatile built-in function, *sorted*, that can sort lists and other sequences, always returning a new list. *sorted* takes named parameters such as *reverse* and *key*.

3. Often we need to sort lists of tuples or lists of lists. By default, *sorted* only sorts based on the value of the first item. To sort by the value of any item in a complex element, we provide a function for the *key* parameter to indicate which item to use for sorting.

4. To avoid creating multiple helper functions only for sorting purposes (which will only be used once), we can adopt Python's syntax for *lambda* expressions. Such expressions can be used whenever a function is needed. An expression can never contain a statement (such as return, if conditional, for loops, print, etc.).

5. Functions like *sorted* that take as parameters other functions are known as **Higher Order Functions**. Python has more of them, for example: *map* and *filter*. They perform the mapping and filtering pattern we encountered in Lecture 12 and can take a *lambda* expression as their first argument (or a helper function).

6. List objects can also be sorted by two list methods: *sort* and *reverse*, which mutate the original list and don't return a new list.

7. We can use the string method *join* to create a string from a list of characters.