List Comprehension & List Sorting

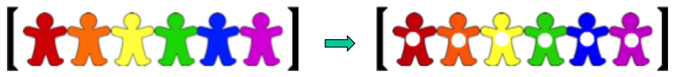
CS111 Computer Programming

Department of Computer Science
Wellesley College

Review: Mapping and Filtering

```
people = ['Katniss Everdeen', 'Peta Mellark',
          'Finnick Odair', 'Effie Trinket']
```

1. MAPPING: return a new list that results from performing an operation on each element of a given list.
E.g. Return a list of the first names in **`people`**
`['Katniss', 'Peta', 'Finnick', 'Effie']`



2. FILTERING: return a new list that results from keeping those elements of a given list that satisfy some condition
E.g. Return a list of names in **`people`** whose last names start with a vowel `['Katniss Everdeen', 'Finnick Odair']`

Simplifying mapping & filtering with list comprehension (LC)

Vocabulary: Comprehension: the act of process of comprising.

```
nums = [8, 3, 1, 2]
result = []
for x in nums:
   result.append(x*2)
print(result) # prints [16, 6, 2, 4]
```

List Comprehension for mapping

```
print ( [x*2 for x in nums] )
```

```
result = []
for n in nums:
    if n%2 == 0:
        result.append(n)
print(result) # prints [8, 2]
```

List Comprehension for filtering

```
print ( [n for n in nums if n%2 == 0] )
```

List comprehension (LC) syntax

Concepts in this slide: List comprehension creates a new list in a single statement.

A **list comprehension** is an **expression** that creates a new list. It is written using what looks like a **`for`** loop inside a pair of square brackets.

List Comprehension for mapping

```
[ elementExpr for var in sequenceExpr ]
```

List Comprehension for filtering

```
[ var for var in sequenceExpr if testExpr ]
```

To notice:
- A list comprehension itself is an **expression** that denotes a list.
- A list comprehension starts with an ***element expression*** (note that a variable references like ***var*** is an expression), for example, `x*2` or `n`.
- Never use **`.append`** in the ***element expression*** . List comprehension s avoid explicitly writing **`.append`** (though it is used behind the scenes).

List comprehensions with Mapping and Filtering

```
[ expression for item in sequence if testExpr ]
```

The example below shows a list comprehension that extracts the even numbers from a range object and creates a list of their squares. The code to the right is analogous and shows the same process with iteration.

```
result = []
for n in range(10):
    if n%2 == 0:
       result.append(n**2)
print(result)
# prints [4, 16, 36, 64]
```

List Comprehension
for filtering and mapping

```
print ( [n**2 for n in range(10) if n%2 == 0] )
```

Your turn to use list comprehension

it's your turn

```
states = ["Alabama", "Arkansas", "California", "Illinois", "Massachusetts",
          "Michigan", "Oklahoma", "Utah", "Washington"]
```

1. Create a list of the lengths of all the strings in **`states`**

2. Create a list of the of the abbreviations of **`states`**
   `['AL', 'AR', 'CA', 'IL', 'MA', 'MI', 'MO', 'WA']`

3. Create a list of strings in **`states`** that end in **`'a'`**

The solutions to these exercises are in the solution notebook for this lecture.

Sorting a list of numbers

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

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
```

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.

`sorted` with other sequences

Concepts in this slide:
sorted works with other sequence types, but always returns a list.

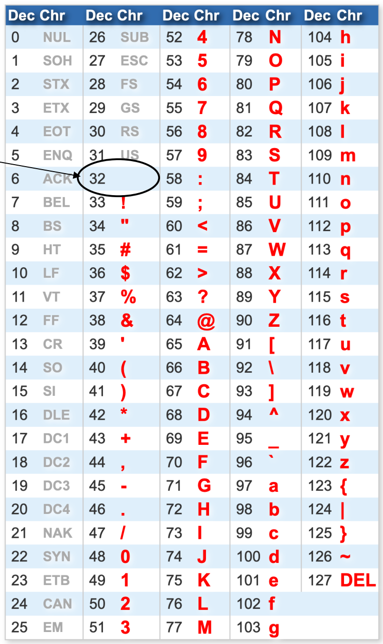
We can apply the function **`sorted`** to other sequences too: strings and tuples. Similarly to sorting lists, **`sorted`** will again create a new list of the sorted elements.

```
In [5]: phrase = 'Red Code 1'
In [6]: sorted(phrase)
Out[6]: [' ', ' ', '1', 'C', 'R', '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 the ASCII table?

ASCII Table

The space character has the code 32, making it the first of the visible string characters.



Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESC	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32		58	:	84	T	110	n
7	BEL	33	!	59	;	85	U	111	o
8	BS	34	"	60	<	86	V	112	p
9	HT	35	#	61	=	87	W	113	q
10	LF	36	$	62	>	88	X	114	r
11	VT	37	%	63	?	89	Y	115	s
12	FF	38	&	64	@	90	Z	116	t
13	CR	39	'	65	A	91	[	117	u
14	SO	40	(	66	B	92	\	118	v
15	SI	41	)	67	C	93	]	119	w
16	DLE	42	*	68	D	94	^	120	x
17	DC1	43	+	69	E	95	_	121	y
18	DC2	44	,	70	F	96	`	122	z
19	DC3	45	-	71	G	97	a	123	{
20	DC4	46	.	72	H	98	b	124	|
21	NAK	47	/	73	I	99	c	125	}
22	SYN	48	0	74	J	100	d	126	~
23	ETB	49	1	75	K	101	e	127	DEL
24	CAN	50	2	76	L	102	f		
25	EM	51	3	77	M	103	g		

Reminder

All keyboard characters are represented as numbers. The first 32 numbers (from 0 to 31) are reserved for invisible characters (mostly on old keyboards). Starting at 32 we have space, then ! and several special characters, followed by digits, uppercase letters, more special characters, lowercase letters, and concluding with other special characters.

Sorting a list of strings

Concepts in this slide: When sorting a list of strings, order is specified by the first string element.

```
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

Concepts in this slide: The mechanics of 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.

This approach to sorting tuples is known as **lexicographic ordering**, which is a generalization of dictionary ordering on strings (where each tuple element is treated as a generalized character in a sequence).

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

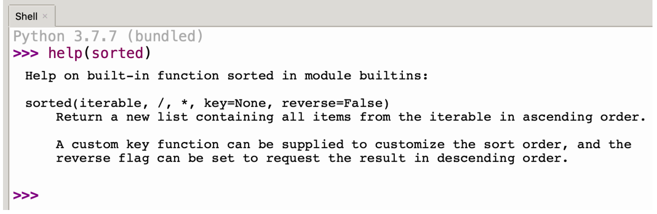
Sorting with the **key** keyword parameter [1]

Concepts in this slide: Using the parameter **key** to sort with functions.

Problem: We have a list of tuples and want to sort by the second item. For example, sort by a person's age in the list below.

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

The function **sorted** takes several parameters, which we can find by typing help in the Thonny shell.



```
Shell
Python 3.7.7 (bundled)
>>> help(sorted)
  Help on built-in function sorted in module builtins:

  sorted(iterable, /, *, key=None, reverse=False)
      Return a new list containing all items from the iterable in ascending order.

      A custom key function can be supplied to customize the sort order, and the
      reverse flag can be set to request the result in descending order.

>>>
```

The first parameter is an "iterable", meaning, any object over which we can iterate (list, string, tuple). We have already seen the keyword parameter **reverse** and now we'll see the keyword parameter **key**.

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

Sorting with the **key** keyword parameter [2]

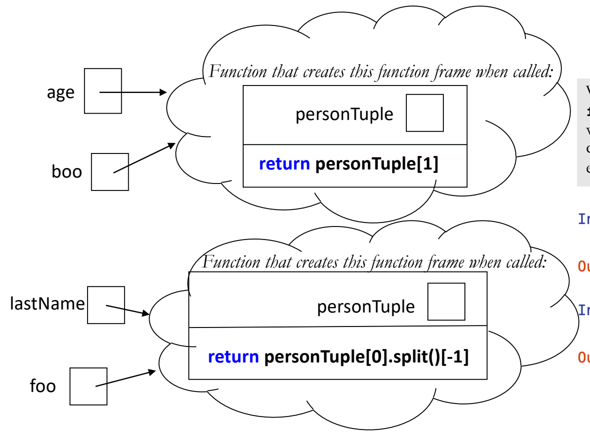
Concepts in this slide: Using the parameter **key** to sort with functions.

```
def age(personTuple):                def lastName(personTuple):
    return personTuple[1]                return personTuple[0].split()[-1]

>>> age(('Janet Doe', 25))           >>> lastName(('Bob Miller', 31))
25                                   'Miller'
```

```
In [19]: sorted(people, key=age)
Out[19]: [('Ed Smith', 17),
          ('Mary Beth Johnson', 18),
          ('Janet Doe', 25),
          ('Bob Miller', 31)]

In [20]: sorted(people, key=lastName)
Out[20]: [('Janet Doe', 25),
          ('Mary Beth Johnson', 18),
          ('Bob Miller', 31),
          ('Ed Smith', 17)]
```

To notice:
The parameter **key** is assigned as a value a function name. While usually **age** and **lastName** will be invoking a function, here they are used as values (no parentheses). Functions in Python are values just like numbers and strings. We use names to refer to these values.

Function names refer to function values


age
boo
Function that creates this function frame when called:
personTuple
return personTuple[1]
lastName
foo
Function that creates this function frame when called:
personTuple
return personTuple[0].split()[-1]

We can create two variables **boo** and **foo**, assign to them the function values, and use them as **aliases** for calling the two functions. See the examples below.

```
In [31]: boo = age
         boo(('Janet Doe', 25))
Out[31]: 25

In [32]: foo = lastName
         foo(('Ed Smith', 17))
Out[32]: 'Smith'
```

Functions in Python are values. Their names point to a place in memory where the function values are stored, as the diagrams above show.

```
In [29]: age
Out[29]: <function __main__.age(personTuple)>

In [30]: lastName
Out[30]: <function __main__.lastName(personTuple)>
```

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), ('Bob Doe', 25), ('Ed Doe', 18),
           ('Ana 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 [21]: sorted(people2, key=ageLastFirst)
Out[21]: [('Ed Doe', 18),
          ('Ana Jones', 18),
          ('Ed Jones', 18),
          ('Ana Doe', 25),
          ('Bob Doe', 25)]
```

Note:
The functions **age** and **lastName** were defined in Slide 12, the function **firstName** is an exercise in the Notebook.

Mutating list methods for sorting

Concepts in this slide: Two new list methods: **sort** and **reverse**. They mutate the original list.

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

```
In [22]: numbers = [35, -2, 17, -9, 0, 12, 19]
In [23]: numbers.sort() # Mutates list; nothing is returned
In [24]: numbers
Out[24]: [-9, -2, 0, 12, 17, 19, 35]

In [25]: numbers2 = [35, -2, 17, -9, 0, 12, 19]
In [26]: numbers2.reverse() # Mutates list; nothing is returned
In [27]: numbers2
Out[27]: [19, 12, 0, -9, 17, -2, 35] # no sorting

In [28]: numbers2.sort()
In [29]: numbers2.reverse()
In [30]: numbers2
Out[30]: [35, 19, 17, 12, 0, -2, -9]
```

Note:
The method **sort** similarly to **sorted** takes the parameters key and reverse as needed.

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 value for the **`key`** parameter to indicate which item to use for sorting.
4. Functions like **`sorted`** that take as parameters other functions are know as Higher Order Functions. We will not see more of these functions in Python, but look for them in higher level CS courses.
5. 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.

Answers to exercises in Slide 6:

```
1) [len(state) for state in states]
2) [state[:2].upper() for state in states]
3) [state for state in states if state.endswith('a')]
```