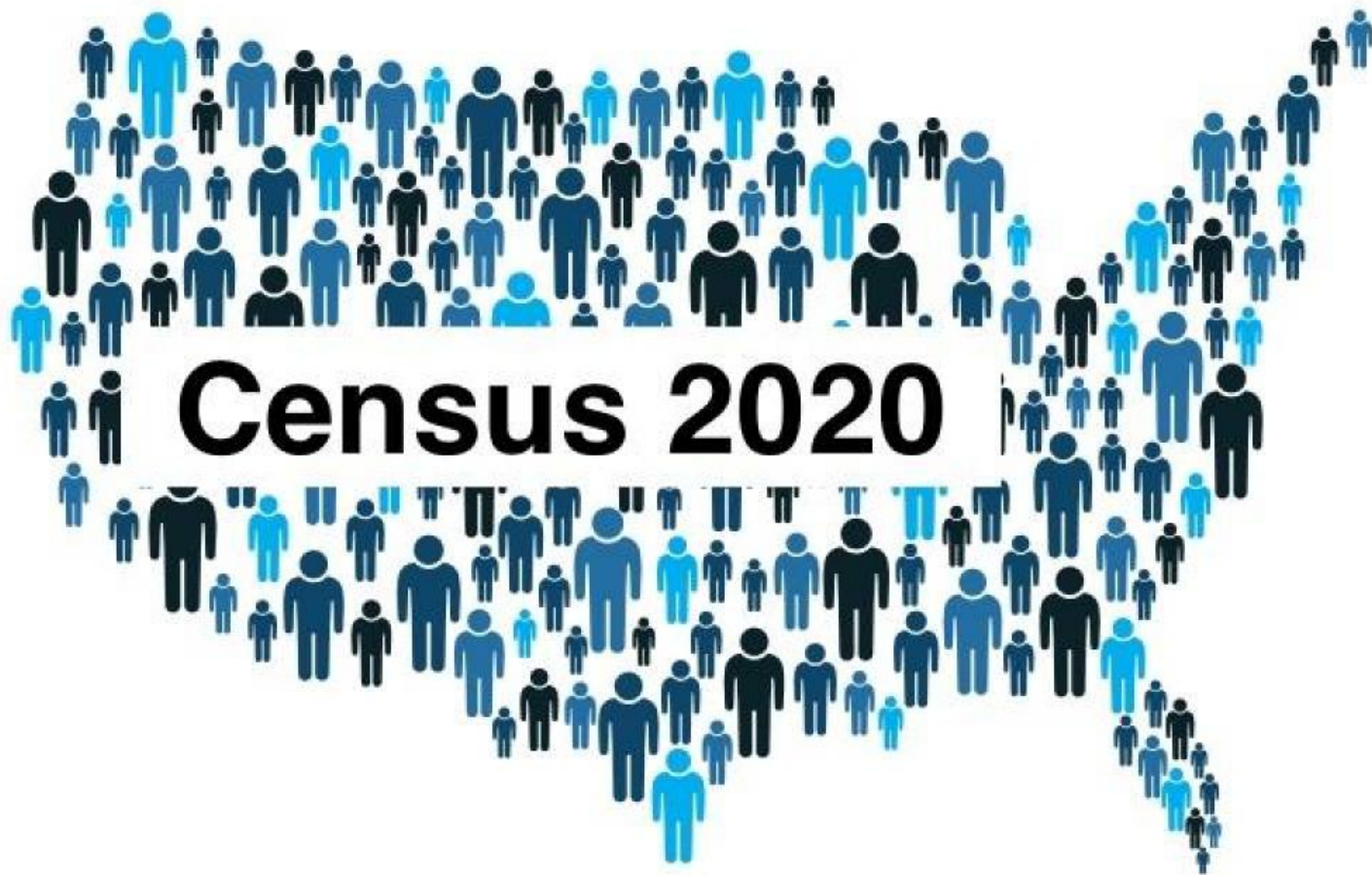


Working with Real-World Data



CS111 Computer Programming

Department of Computer Science
Wellesley College



Representation in Congress is based on population. More people, more seats.

STATE	APPORTIONMENT POPULATION (APRIL 1, 2020)	NUMBER OF APPORTIONED REPRESENTATIVES BASED ON 2020 CENSUS ²	CHANGE FROM 2010 CENSUS APPORTIONMENT
Alabama	5,030,053	7	0
Alaska	736,081	1	0
Arizona	7,158,923	9	0
Arkansas	3,013,756	4	0
California	39,576,757	52	-1
Colorado	5,782,171	8	1
Connecticut	3,608,298	5	0
Delaware	990,837	1	0
Florida	21,570,527	28	1
Georgia	10,725,274	14	0
Hawaii	1,460,137	2	0
Idaho	1,841,377	2	0
Illinois	12,822,739	17	-1
Indiana	6,790,280	9	0
Iowa	3,192,406	4	0
Kansas	2,940,865	4	0
Kentucky	4,509,342	6	0
Louisiana	4,661,468	6	0
Maine	1,363,582	2	0
Maryland	6,185,278	8	0
Massachusetts	7,033,469	9	0
Michigan	10,084,442	13	-1

US States and Capitals: Doing more with our data

	A	B	C	D	E
1	State	StatePop	Abbrev.	Capital	CapitalPop
2	Alabama	4921532	AL	Montgomery	198525
3	Alaska	731158	AK	Juneau	32113
4	Arizona	7421401	AZ	Phoenix	1680992
5	Arkansas	3030522	AR	Little Rock	197312
6	California	39368078	CA	Sacramento	513624
7	Colorado	5807719	CO	Denver	727211

Partial screenshot of the `us-states-more.csv` file, viewed with the Google Spreadsheet editor.

Some questions to answer with our data:

- Which are the **most** populated US states? **Rank** the data in that order.
- Which are the **least** populated US states? **Rank** the data in that order.
- Which US state capitals are the **most** populated? **Rank** the data in that order.
- Which US state capitals are the **least** populated? **Rank** the data in that order.
- What percentage of each US state's population lives in the state capital? **Rank** the data by that percentage from the **largest** to the **smallest**.



Moving beyond **max** and **min** by applying
sorting to sequences

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.

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.

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		

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?

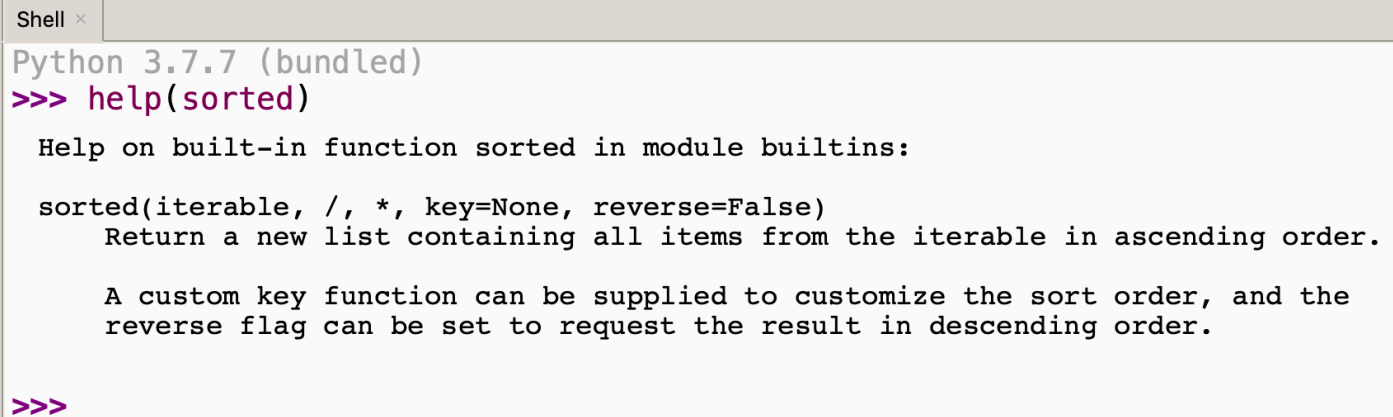
Sorting with the **key** parameter [1]

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 x  
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 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.

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

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

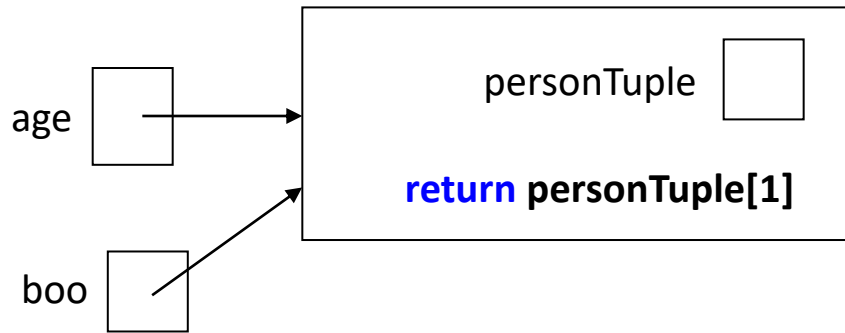
```
def age(personTuple):  
    return personTuple[1]  
  
>>> age(('Janet Doe', 25))  
25  
  
def lastName(personTuple):  
    return personTuple[0].split()[-1]  
  
>>> lastName(('Bob Miller', 31))  
'Miller'
```

```
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



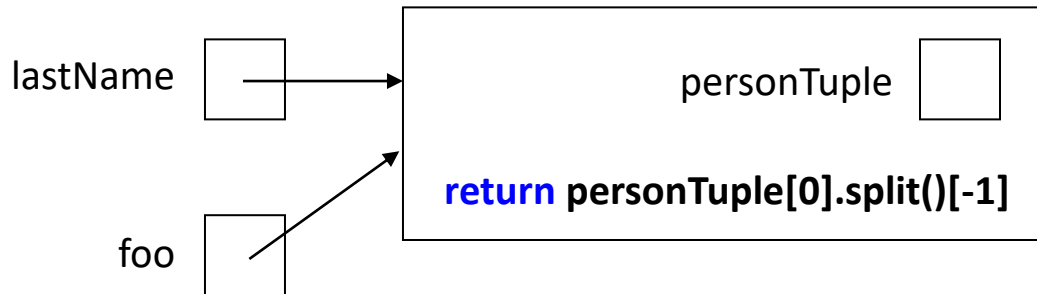
We can create two variables **boo** and **foo**, assign to them the function values, and use them as aliases for calling the two functions. See the examples below.

```
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), ('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 [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.

Can dictionaries be sorted? Explain the outputs!

```
In [31]: fruitColors = {"banana": "yellow", "kiwi":  
"green", "grapes": "purple", "apple": "red", "lemon":  
"yellow", "pomegranate": "red"}
```

```
In [32]: sorted(fruitColors)
```

```
Out[32]: ['apple', 'banana', 'grapes', 'kiwi', 'lemon',  
'pomegranate']
```

```
In [33]: sorted(fruitColors.keys())
```

```
Out[33]: ['apple', 'banana', 'grapes', 'kiwi', 'lemon',  
'pomegranate']
```

```
In [34]: sorted(fruitColors.values())
```

```
Out[34]: ['green', 'purple', 'red', 'red', 'yellow',  
'yellow']
```

```
In [35]: sorted(fruitColors.items())
```

```
Out[35]: [('apple', 'red'), ('banana', 'yellow'),  
(('grapes', 'purple'), ('kiwi', 'green'), ('lemon',  
'yellow'), ('pomegranate', 'red'))]
```


Sort a list of dictionaries

```
In [36]: peopleDctList = [{ 'name': 'Mary Beth Johnson', 'age': 18},  
                           { 'name': 'Ed Smith', 'age': 17},  
                           { 'name': 'Janet Doe', 'age': 25},  
                           { 'name': 'Bob Miller', 'age': 31}]
```

```
In [37]: sorted(peopleDctList)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell>", line 1, in <module>
```

```
TypeError: '<' not supported between instances of 'dict' and 'dict'
```

```
def byAge(personDct):  
    return personDct[ 'age' ]
```

```
In [38]: sorted(peopleDctList, key=byAge, reverse=True)
```

```
Out[38]: [{ 'name': 'Bob Miller', 'age': 31},  
           { 'name': 'Janet Doe', 'age': 25},  
           { 'name': 'Mary Beth Johnson', 'age': 18},  
           { 'name': 'Ed Smith', 'age': 17}]
```

**END
DETOUR**



US States and Capitals: Doing more with our data

	A	B	C	D	E
1	State	StatePop	Abbrev.	Capital	CapitalPop
2	Alabama	4921532	AL	Montgomery	198525
3	Alaska	731158	AK	Juneau	32113
4	Arizona	7421401	AZ	Phoenix	1680992
5	Arkansas	3030522	AR	Little Rock	197312
6	California	39368078	CA	Sacramento	513624
7	Colorado	5807719	CO	Denver	727211

Partial screenshot of the `us-states-more.csv` file, viewed with the Google Spreadsheet editor.

Some questions to answer with our data:

1. Which are the **most** populated states? **Rank** the data in that order.
2. Which are the **least** populated states? **Rank** the data in that order.
3. Which state capitals are the **most** populated? **Rank** the data in that order.
4. Which state capitals are the **least** populated? **Rank** the data in that order.
5. What percentage of each state's population lives in the state capital? **Rank** the data by that percentage from the **largest** to the **smallest**.

Questions 1 & 2: Sort by US state population

How to implement the solution with Python code:

1. Read the content of the CSV file `us-states-more.csv` using **`csv.DictReader`**, which returns a list of dictionaries.
2. Create a helper function `byStatePop`, which, given a dictionary with state data (one row from our file), returns the appropriate value. Remember that all values in the dictionary are strings, because they come from the CSV file.
3. Apply the **`sorted`** function to the list of dictionaries of state data, using the **`key`** parameter with the function `byStatePop`.
4. Look at the results, in which way are they sorted?
5. Include the function parameter **`reverse`** to change the order of sorting.
6. Use f-string formatting to print out top six results as shown below.

Top six most populated US states:

CA -> 39,368,078
TX -> 29,360,759
FL -> 21,733,312
NY -> 19,336,776
PA -> 12,783,254
IL -> 12,587,530

Top six least populated US states:

WY -> 582,328
VT -> 623,347
AK -> 731,158
ND -> 765,309
SD -> 892,717
DE -> 986,809

Questions 3 &4: Sort by capital population

How to implement the solution with Python code:

Follow the steps from the previous slide, but create appropriate functions to use with the parameter key for sorted. Try to come as close as possible to these outputs, but don't worry if you cannot. These outputs use some special f-string features for formatting.

Top six most populated US state capitals:

Phoenix (AZ)	->	1,680,992
Austin (TX)	->	978,908
Columbus (OH)	->	898,553
Indianapolis (IN)	->	876,384
Denver (CO)	->	727,211
Boston (MA)	->	692,600

Top six least populated US state capitals:

Montpelier (VT)	->	7,855
Pierre (SD)	->	13,646
Augusta (ME)	->	18,681
Frankfort (KY)	->	27,679
Juneau (AK)	->	32,113
Helena (MT)	->	32,315

Questions 5: Sort by percentage

How to implement the solution with Python code:

This will be similar to the two previous slides, by you'll have to create a helper function, **byPercentage**, which can calculate the percentage of people living in the capital of the state. This function will be used by the **sorted** function, as well as by the f-string. Try to come close to this output, but do not worry if you cannot achieve it yet.

Top six US states with the largest population percentage living in the capital:

Hawaii	24.52% of population lives in the capital, Honolulu.
Arizona	22.65% of population lives in the capital, Phoenix.
Rhode Island	17.02% of population lives in the capital, Providence.
Oklahoma	16.46% of population lives in the capital, Oklahoma City.
Nebraska	14.92% of population lives in the capital, Lincoln.
Indiana	12.97% of population lives in the capital, Indianapolis.

Test your knowledge

1. What is the output when we apply the function **sorted** on
 - a) a list,
 - b) a string,
 - c) a tuple, or
 - d) a dictionary?
2. What are some differences between the built-in function **sorted** and the list method **sort**?
3. In this lecture, we saw different invocations for the function **sorted**. With one argument, with two, and even three. When we used more than one argument, we provided the parameter name as well, for example, **reverse=True** or **key=age**. Speculate on why we needed to do that.
4. What do we need to do in order to sort a list of dictionaries? Why is that?
5. What are some other questions that you could answer with the Census data. Can you write the Python code to answer them? Try it out and let us know what you did. We might add that in our material for future semesters.