Introduction to Computation for the Humanities and Social Sciences

CS 3

Chris Tanner

Lecture 10

"I heard you like lists, so I decided to put a list in your list"

— Xzibit

sys.argv and dictionaries

- **sys.argv** allows the user to specify arguments which your program can then use, instead of relying on hard-coded, manually-defined values from the programmer (e.g., the input file)

- example was posted on the course website last week: dictionary_example.py

Good Coding Style

- Put a space on both sides of an operator:

```
count = count + 1
```
good

```
count=count+1
```
bad

- Put a space after each comma (just like in English)

good

```
def function_name(a, b):
    return a*2 + b*b
```

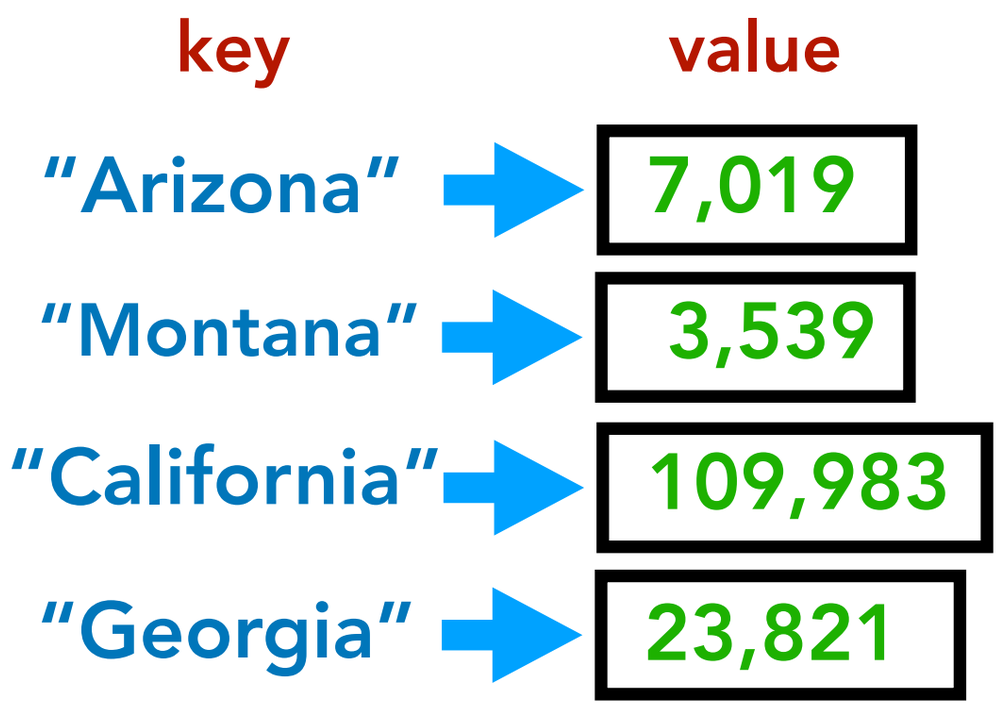
- No space before "(" when you call a function:

good

```
num_words = count_words(filename)
```

Dictionaries

Important: each **key** in a dictionary is unique from the others and is only present at most one time — it's impossible to store the same key multiple times.



Dictionaries

- First, you <u>must always</u> initialize the dictionary (curly brackets!):

```
students = {}
```

```
students[course_num] = roster_list
```

key

value

dictionary's variable name

REAL-TIME CODING

- word counts
- words that start w/ each letter (key -> list)

Example of Updating a key's value

```
# counts how many times each word occurs in a file
word_counts = {}
input_file = open(filename, 'r')
words = input_file.read().split(" ")
for current_word in words:

    if current_word in words: # update the count
        word_counts[current_word] = word_counts[current_word] + 1

    else: # initialize the count
        word_counts[current_word] = 1
```

Iterate through the Dictionary's Keys

```
# iterate through the keys
for word in word_counts.keys():
    print(word + " appears " + str(word_counts[word]) + " times")
```

value

Dictionary Values

- As mentioned in last lecture, dictionaries' keys must be single-valued, but their values can be data structures other than just single-valued

- Values can be `lists, sets, tuples,` or even another `dictionary`!

- See text_analysis.py for an example (the in-class demo)

Lists

- `Lists` are useful data structures, but sometimes you don't want repeated items in a list (e.g., our in-class example of a list words that start with a given letter)

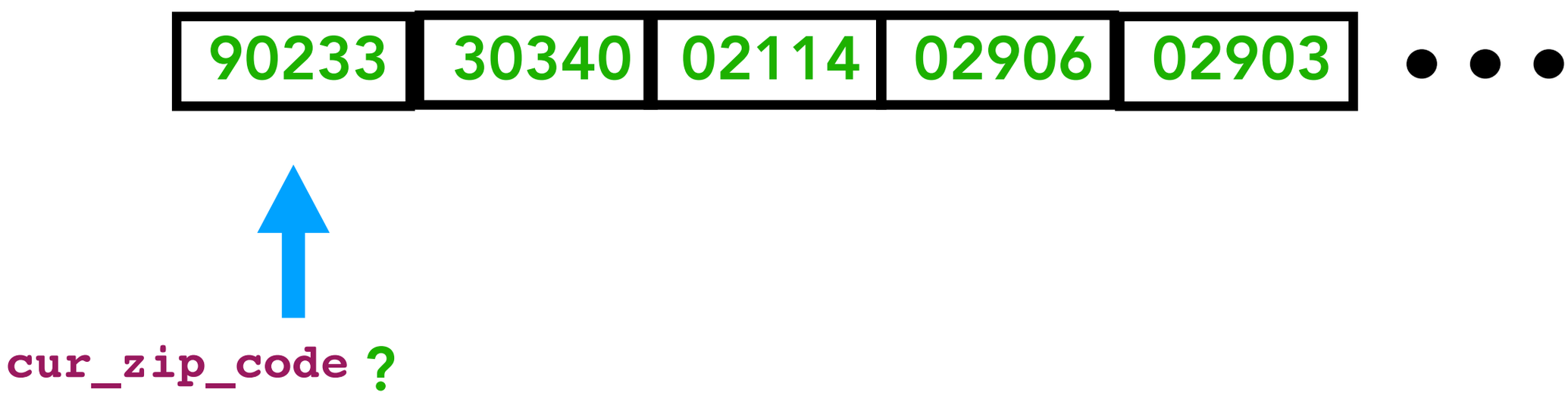
```
# with lists, you'd need to check if the item isn't present
senators = []
for line in input_file:
    cur_senator = line.strip().split(" ")[0] # grabs 1st column of text

    if cur_senator not in senators: # don't add duplicates
        senators.append(cur_senator)
```

Lists

- `Lists` are useful data structures, but if you want to check if an item is in the list, under the hood Python looks at every single item in the list until it finds it or not.

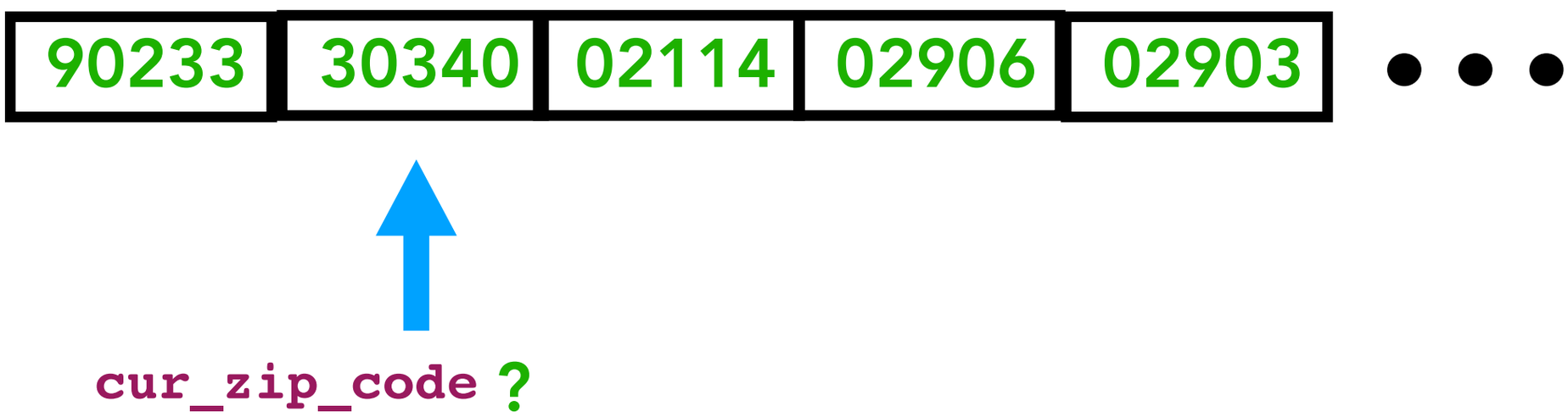
```
if cur_zip_code not in zip_codes: # don't add duplicates
    zip_codes.append(cur_zip_code)
```



Lists

- `Lists` are useful data structures, but if you want to check if an item is in the list, under the hood Python looks at every single item in the list until it finds it or not.

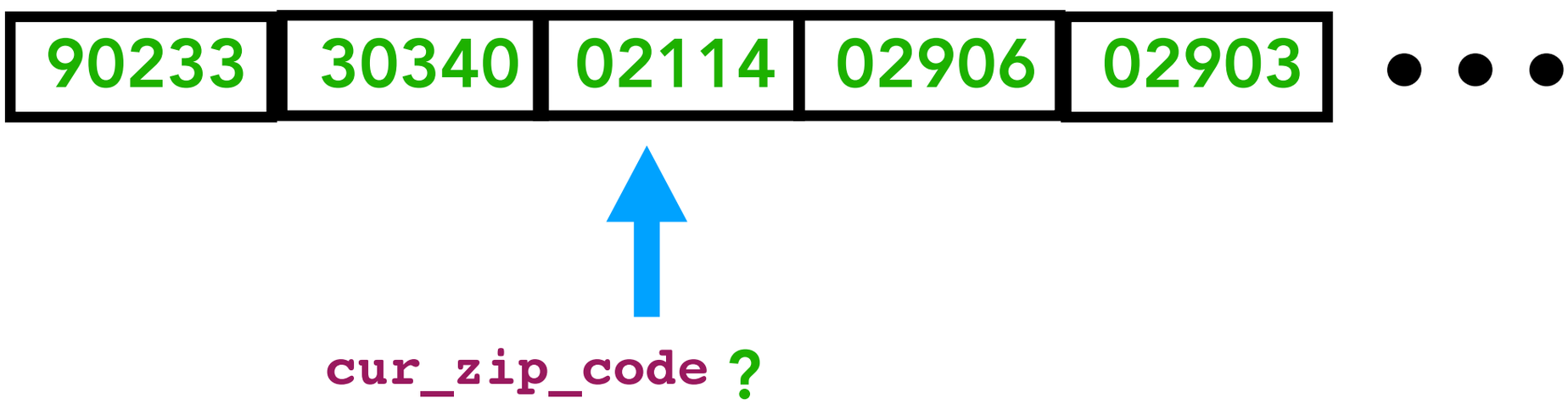
```
if cur_zip_code not in zip_codes: # don't add duplicates
    zip_codes.append(cur_zip_code)
```



Lists

- `Lists` are useful data structures, but if you want to check if an item is in the list, under the hood Python looks at every single item in the list until it finds it or not.

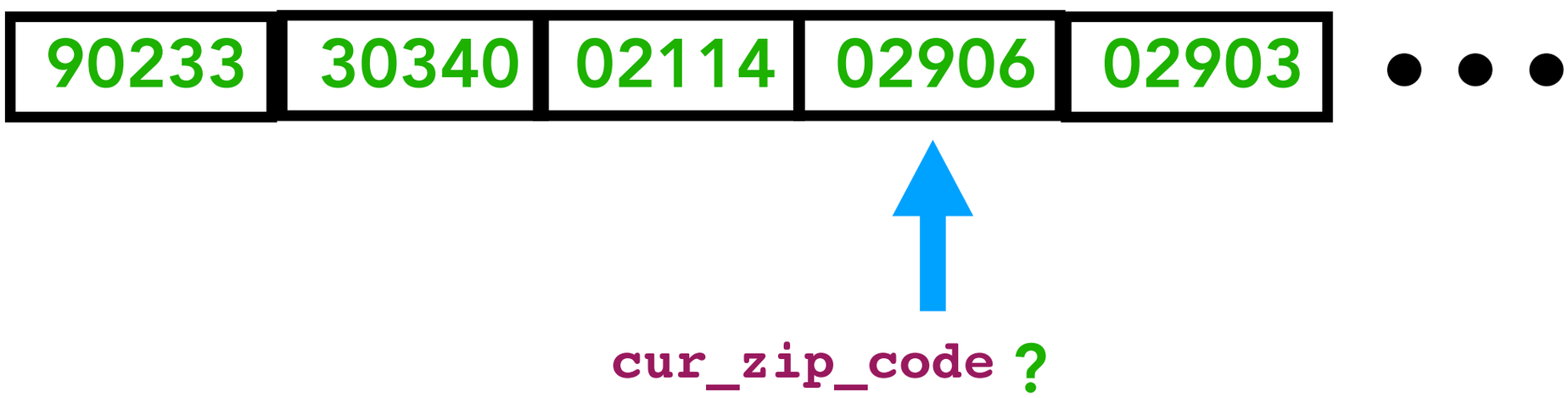
```
if cur_zip_code not in zip_codes: # don't add duplicates
    zip_codes.append(cur_zip_code)
```



Lists

- `Lists` are useful data structures, but if you want to check if an item is in the list, under the hood Python looks at every single item in the list until it finds it or not.

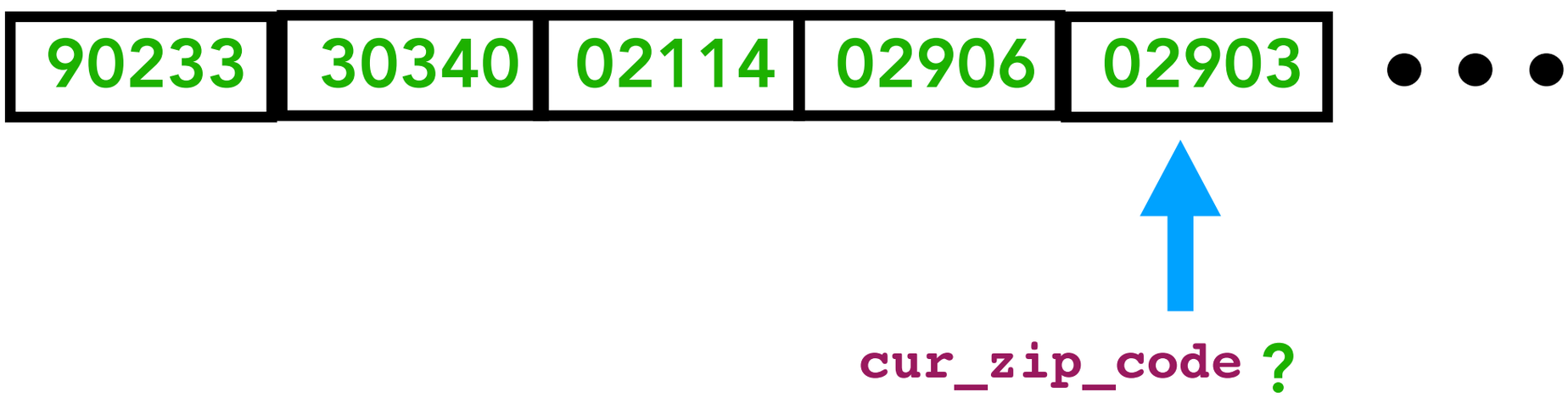
```
if cur_zip_code not in zip_codes: # don't add duplicates
    zip_codes.append(cur_zip_code)
```



Lists

- `Lists` are useful data structures, but if you want to check if an item is in the list, under the hood Python looks at every single item in the list until it finds it or not.

```
if cur_zip_code not in zip_codes: # don't add duplicates
    zip_codes.append(cur_zip_code)
```



Sets

New Data Structure!

- very similar to `Lists`
- `A Set is` an unordered collection of items
- Collections which don't preserve the order in which elements are added are called **unordered**
- Doesn't support indexing

```
a = {1,2,3}
a.add(4)
if 2 in a:
  print(a)
```

Sets

New Data Structure!

- very similar to `Lists`, but
- is incapable of storing duplicate items (maintains just 1 copy)

```
students = set()
students.add("jackie")
students.add("emily")
students.add("hank")
students.add("emily")
print(students)
```

```
>>> students = {'emily', 'jackie', 'hank'}
```

Sets

New Data Structure!

- checking if an item is contained in the Set happens instantly (doesn't need to check each item one by one)

```
students = set()
students.add("jackie")
students.add("emily")
students.add("hank")
students.add("emily")

if cur_student in students:
    print(student + "is present!")
else:
    print(student + "is NOT present!")
```

Sets

New Data Structure!

- The downside: `Sets` have no indices and no order!

- So, you can't access specific items. If you need to maintain an order for your items, use a `List`.

```
students = set()
students.add("jackie")
students.add("emily")
students.add("hank")
students.add("emily")
students[0]
```

CRASHES