02-21-2022

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

10: Dictionaries and recursion

David Kauchak

he/him/his

Lectures

Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

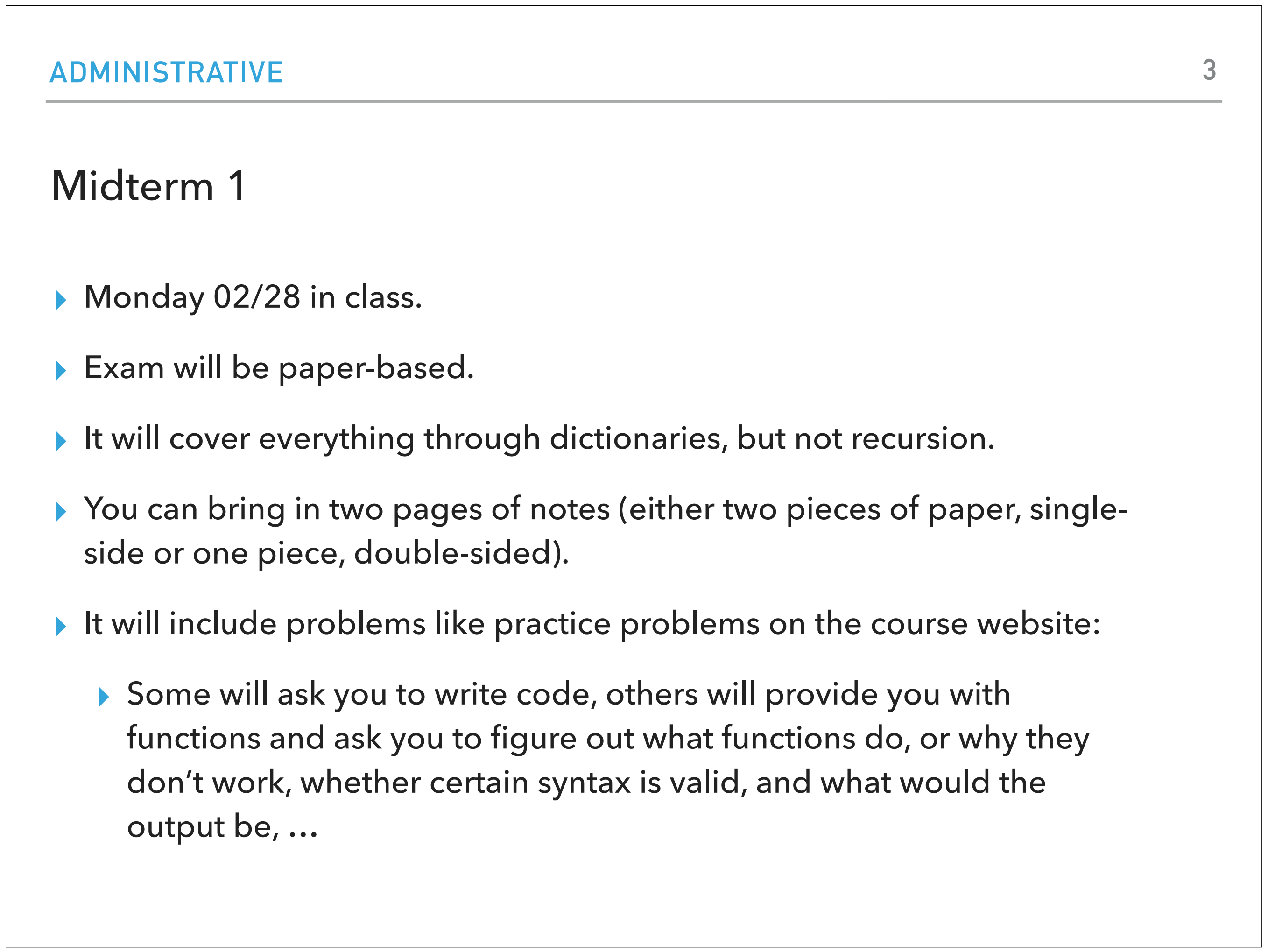
Labs

Welcome to lecture 10, where we will examine dictionaries, a new data structure, that will allow us to store pairs of keys and values, and recursion, a new way of thinking about our code.

Lecture 10: Dictionaries and recursion

- Administrative
- Dictionaries
- Recursion

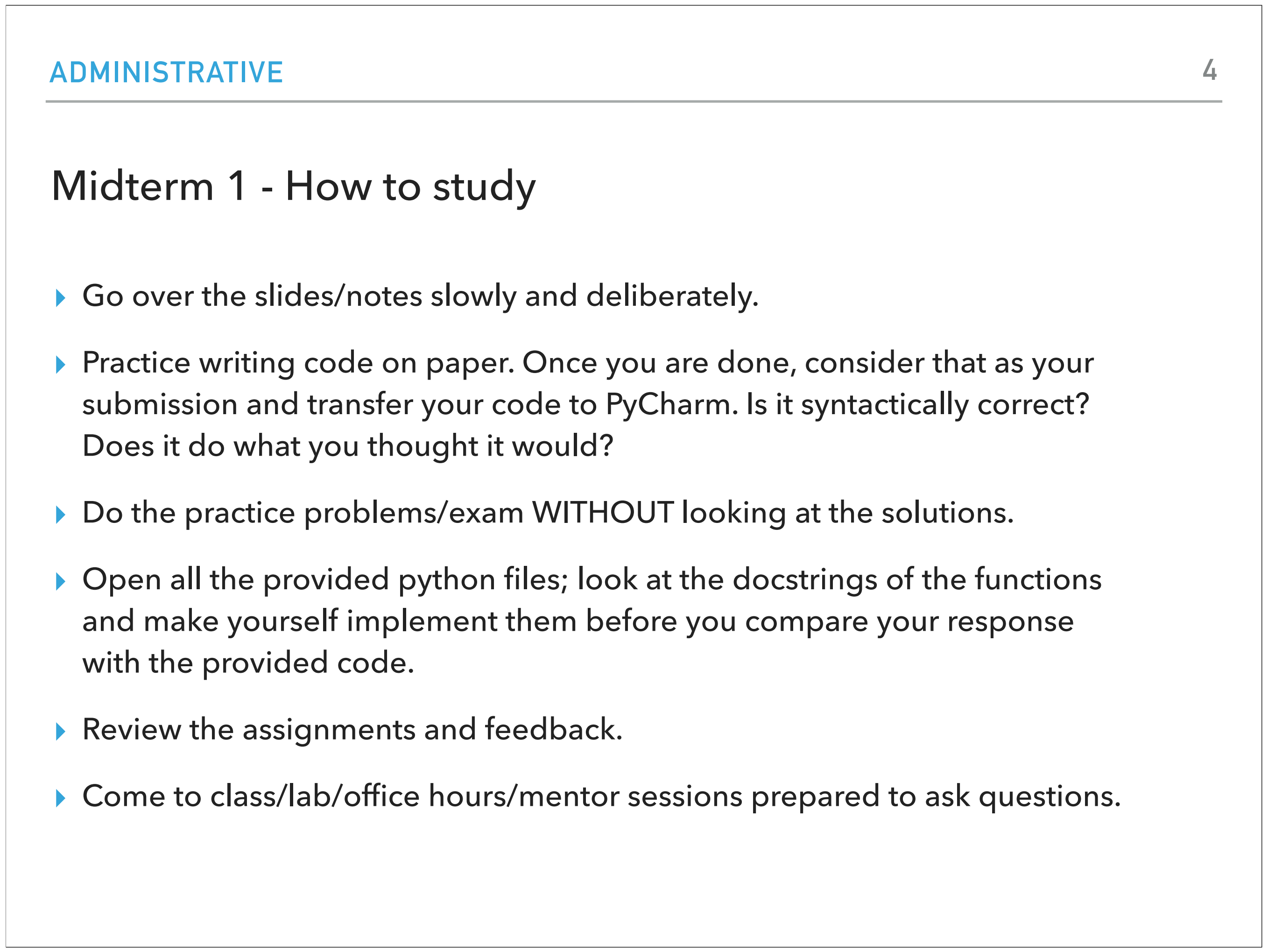
Before we start talking about dictionaries, I wanted to go over some announcements.



Midterm 1

- Monday 02/28 in class.
- Exam will be paper-based.
- It will cover everything through dictionaries, but not recursion.
- You can bring in two pages of notes (either two pieces of paper, single-side or one piece, double-sided).
- It will include problems like practice problems on the course website:
  - Some will ask you to write code, others will provide you with functions and ask you to figure out what functions do, or why they don't work, whether certain syntax is valid, and what would the output be, …

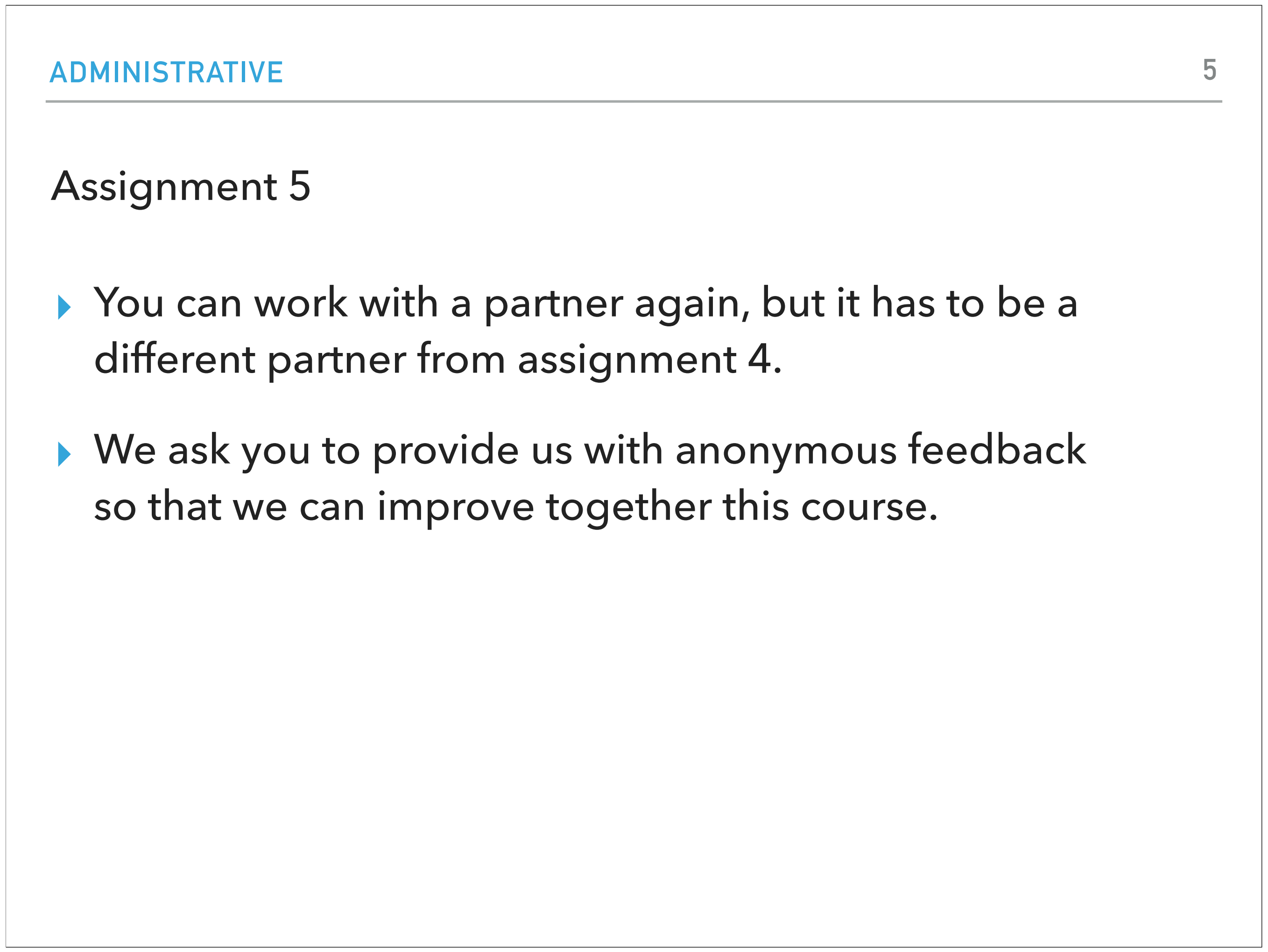
Our first midterm is coming up next week and as we have already discussed it will be paper-based. In terms of material it will cover, everything up to today's lecture, excluding recursion, is fair game. You can bring in two pages of notes with you. The exam will include problems like practice problems on the course website, such as asking you to write code, figure out what functions do, if code syntactically valid, etc.



Midterm 1 - How to study

- Go over the slides/notes slowly and deliberately.
- Practice writing code on paper. Once you are done, consider that as your submission and transfer your code to PyCharm. Is it syntactically correct? Does it do what you thought it would?
- Do the practice problems/exam WITHOUT looking at the solutions.
- Open all the provided python files; look at the docstrings of the functions and make yourself implement them before you compare your response with the provided code.
- Review the assignments and feedback.
- Come to class/lab/office hours/mentor sessions prepared to ask questions.

How do you study to do well? You should start by reviewing the slides/notes slowly and deliberately. To gain comfort with practicing writing code on paper, you need to treat this as the exam. Only once you are done, then consider that as your submission, transfer your code to PyCharm and see if it is syntactically correct and if it produces the results you thought. We also provide you with a lot of practice problems. Make sure you review these and answer them thoroughly before you look at the solutions. At the end of each presentation (or top of Dr. Dave's notes), you will find links to the python files we have used during the lecture. Look at the function definition and its doctstring. Can you reproduce it? Try it! Then compare your response with the code we provide you. You can also take a look at the assignments and the feedback that Prof Ye and the TAs have provided. And as always, come to us with questions :)



Assignment 5

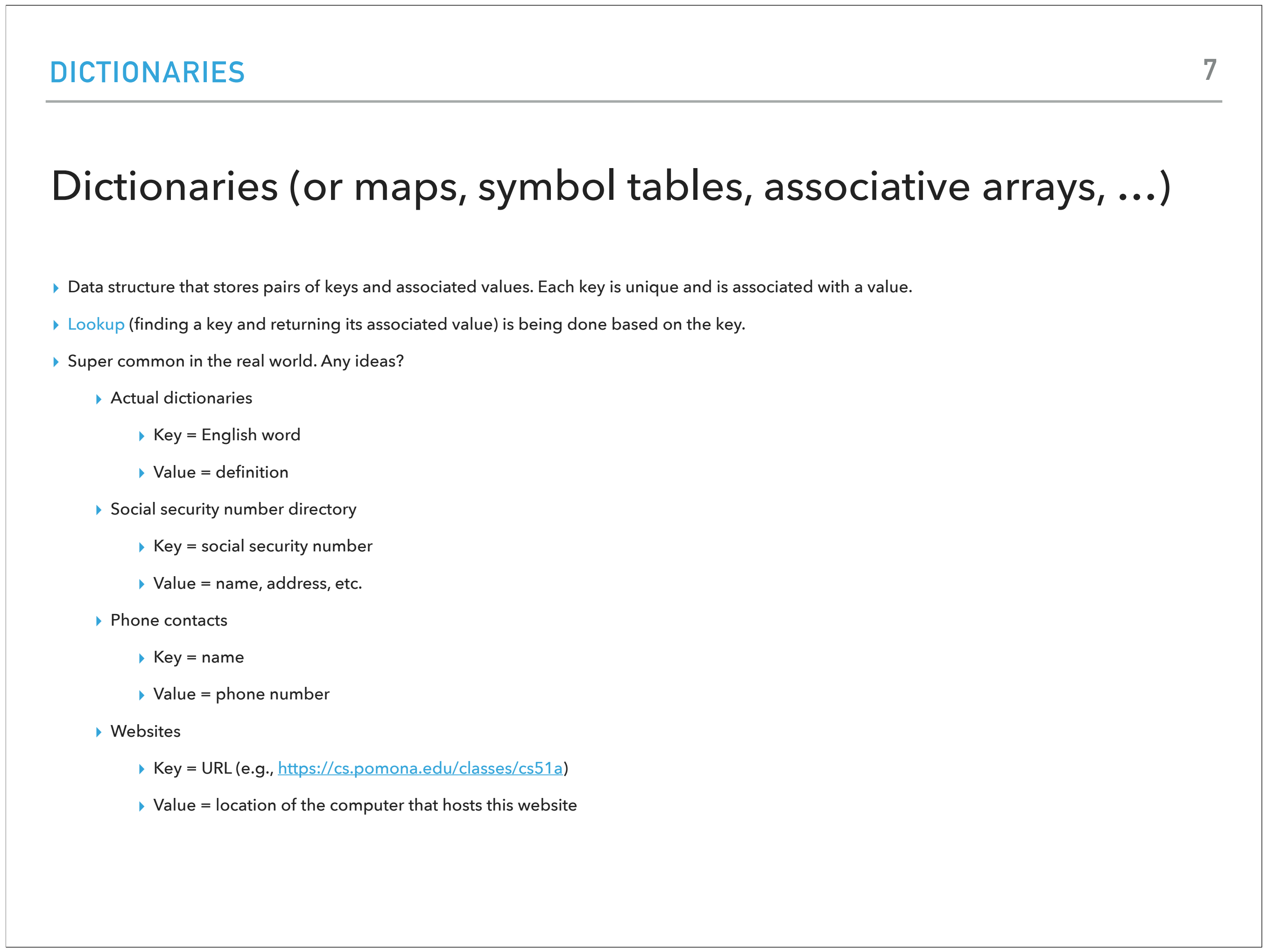
- You can work with a partner again, but it has to be a different partner from assignment 4.
- We ask you to provide us with anonymous feedback so that we can improve together this course.

We also released assignment 5 which can again be completed collaboratively but not with the same partner with assignment 4. We also ask you to provide us with anonymous feedback so we can improve together this course.

Lecture 10: Dictionaries and recursion

- Administrative
- Dictionaries
- Recursion

OK, it's time for Python!



Dictionaries (or maps, symbol tables, associative arrays, …)

- Data structure that stores pairs of keys and associated values. Each key is unique and is associated with a value.
- Lookup (finding a key and returning its associated value) is being done based on the key.
- Super common in the real world. Any ideas?
  - Actual dictionaries
    - Key = English word
    - Value = definition
  - Social security number directory
    - Key = social security number
    - Value = name, address, etc.
  - Phone contacts
    - Key = name
    - Value = phone number
  - Websites
    - Key = URL (e.g., https://cs.pomona.edu/classes/cs51a)
    - Value = location of the computer that hosts this website

Today we will start with dictionaries, a data structure that stores pairs of keys and associated values. Each key in our dictionary is unique and is associated with a single value. If we were to look up information, we would provide a key and we would get back its associated value. Dictionaries (other aliases in CS include maps, symbol tables, and associate arrays) are super common in real world. Well, you already know about real dictionaries that associate words with definitions. Others could be the social security number directory, your phone contacts, or even websites. In each of these cases, there is a unique key associated with a value.

Creating dictionaries

- Dictionaries can be created using curly braces:

```
>>> dict = {}
>>> dict
{}
```

- Dictionaries function similarly to lists, except we can put things in ANY index and can use non-numerical indices. Notice when a dictionary is printed out, we get the key AND the associated value:

```
>>> dict[15] = 1
>>> dict
{15: 1}
>>> dict[100] = 10
>>> dict
{15: 1, 100: 10}
```

In Python, you can create a dictionary using the curly braces. Dictionaries function similarly to lists, except we can put things in ANY index and can use non-numerical indices. Notice when a dictionary is printed out, we get the key AND the associated value.

Keys can be any immutable object

```
>>> dict = {}
>>> dict["dave"] = 1
>>> dict["alexandra"] = 1
>>> dict["alexandra"] = 2
>>> dict["zilong"] = 100
>>> dict
{'dave': 1, 'alexandra': 2, 'zilong': 100}
```

Keys can be anything as long as that anything is an immutable object (that is int, float, str, tuple, but not… list)

Values can be any object

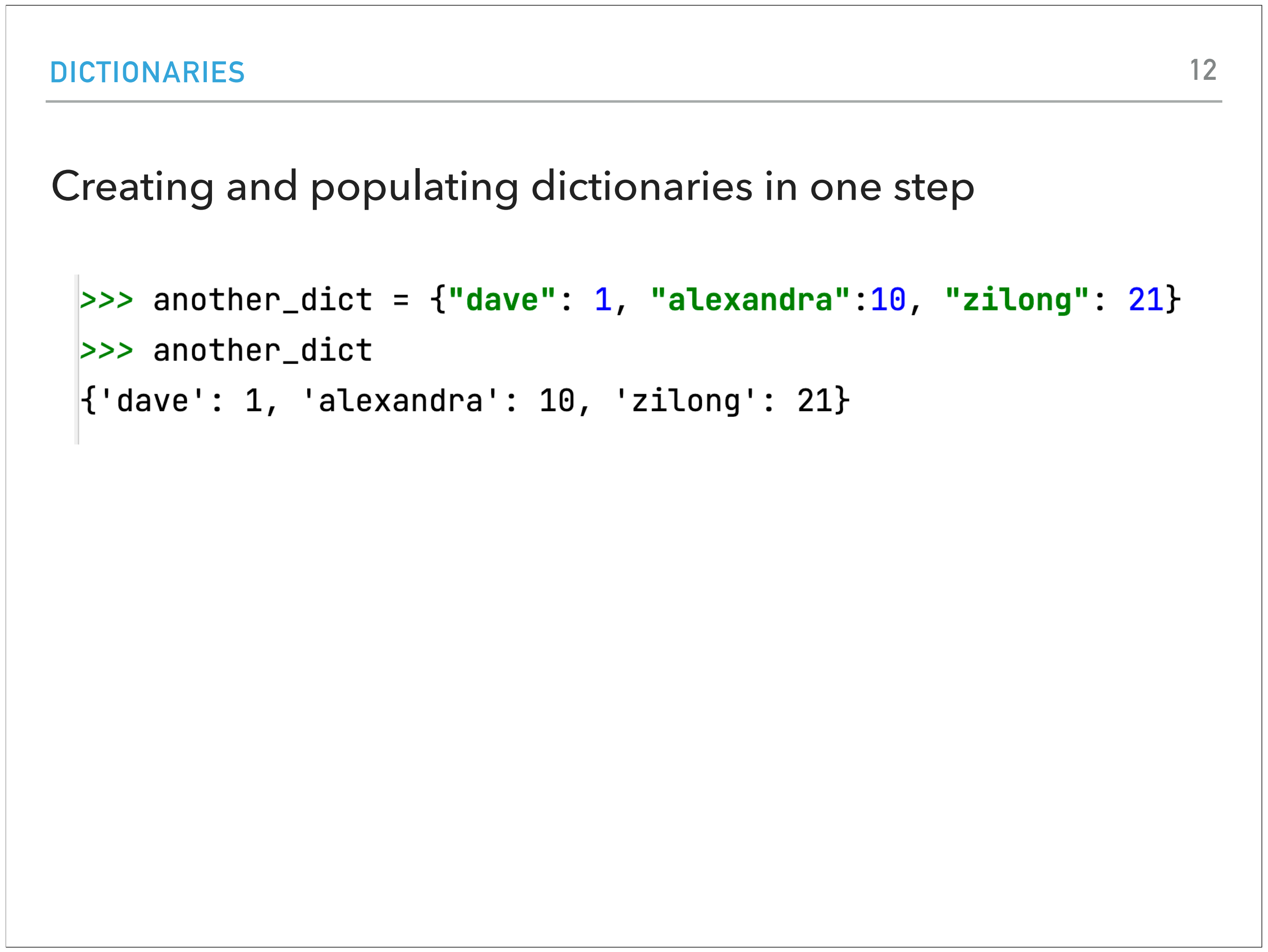
```
>>> dict = {}
>>> dict["dave"] = []
>>> dict
{'dave': []}
>>> dict["dave"].append(1)
>>> dict
{'dave': [1]}
>>> dict["dave"].append(47)
>>> dict
{'dave': [1, 47]}
```

Values on the other hand can be any time of object, including a list!

Be careful to put the key in the dictionary before trying to use it

```
>>> dict["steve"].append(1)
Traceback (most recent call last):
  File "/Library/Frameworks/Python.fra
    exec(code, self.locals)
  File "<input>", line 1, in <module>
KeyError: 'steve'
```

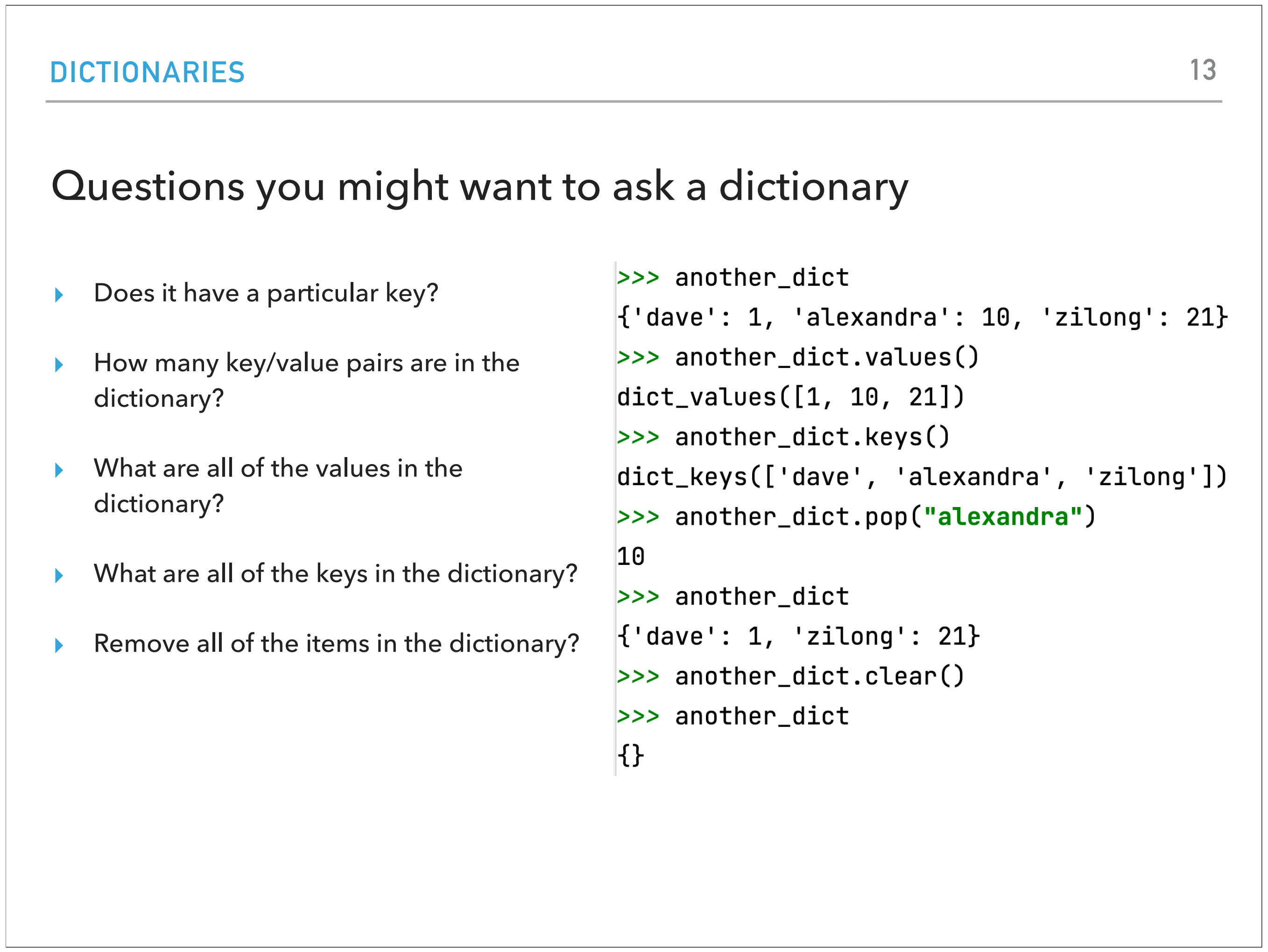
If you ask to use a value that is associated with a key that doesn't exist you will get an error.



Creating and populating dictionaries in one step

```
>>> another_dict = {"dave": 1, "alexandra":10, "zilong": 21}
>>> another_dict
{'dave': 1, 'alexandra': 10, 'zilong': 21}
```

There is also a single line of code you can use to create and populate a dictionary in one step.

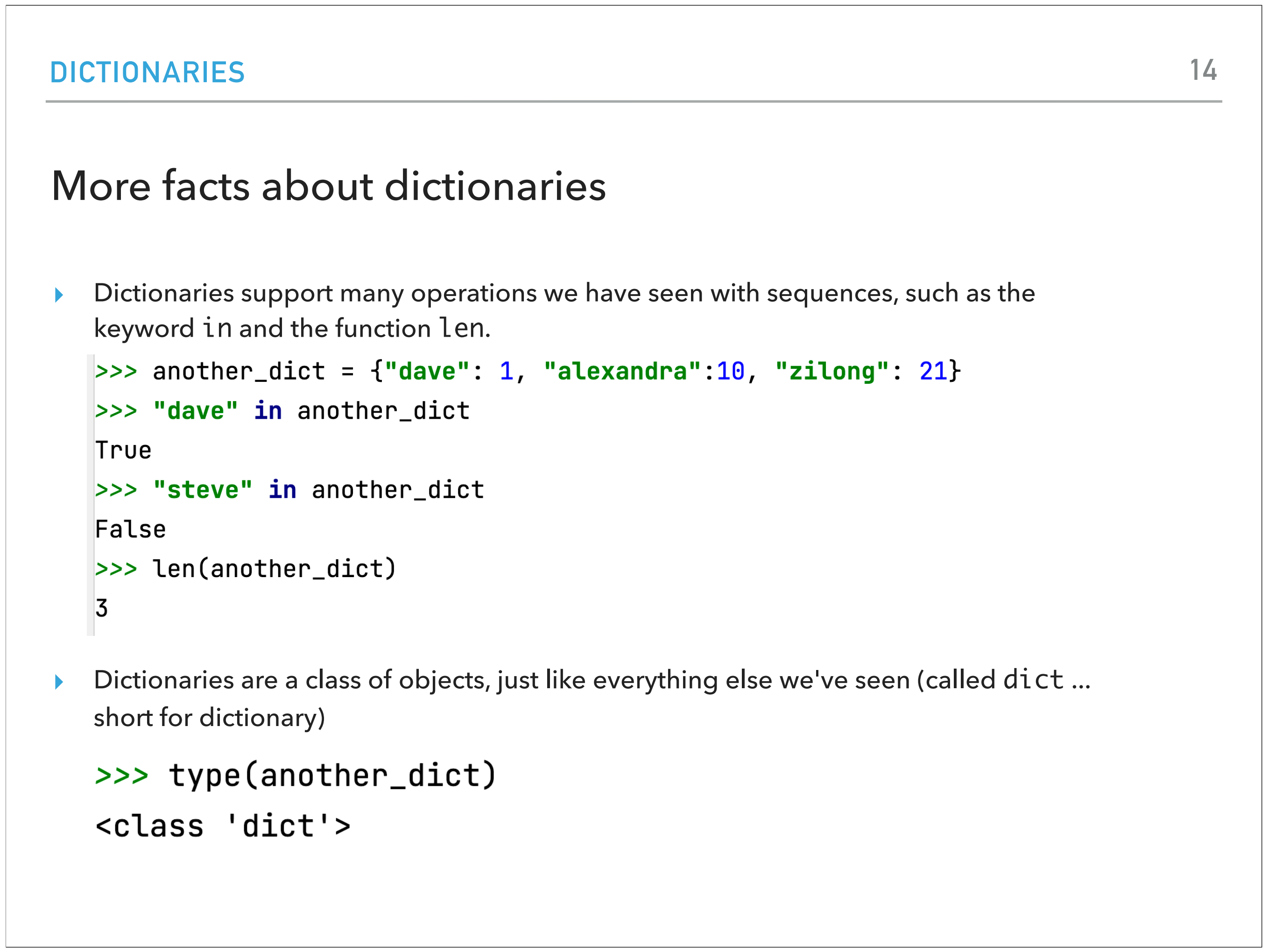


Questions you might want to ask a dictionary

- Does it have a particular key?
- How many key/value pairs are in the dictionary?
- What are all of the values in the dictionary?
- What are all of the keys in the dictionary?
- Remove all of the items in the dictionary?

```
>>> another_dict
{'dave': 1, 'alexandra': 10, 'zilong': 21}
>>> another_dict.values()
dict_values([1, 10, 21])
>>> another_dict.keys()
dict_keys(['dave', 'alexandra', 'zilong'])
>>> another_dict.pop("alexandra")
10
>>> another_dict
{'dave': 1, 'zilong': 21}
>>> another_dict.clear()
>>> another_dict
{}
```

When you have a dictionary, you might want to ask questions such as whether it has a particular key, how many key/value pairs it contains, what are its keys, what are its values, and you might want to apply actions such as removing all its items.



More facts about dictionaries

- Dictionaries support many operations we have seen with sequences, such as the keyword `in` and the function `len`.

```
>>> another_dict = {"dave": 1, "alexandra":10, "zilong": 21}
>>> "dave" in another_dict
True
>>> "steve" in another_dict
False
>>> len(another_dict)
3
```

- Dictionaries are a class of objects, just like everything else we've seen (called `dict` … short for dictionary)

```
>>> type(another_dict)
<class 'dict'>
```

Similarly to sequences, dictionaries support the in keyword and the function len. The name of the type of a dictionary is dict.

Practice time

- Let's write a function called `get_counts` that takes a list of numbers and returns a dictionary containing the counts of each of the numbers.
- Key idea:

```
>>> def get_counts(numbers):
...     d = {}
...
...     for num in numbers:
...     # do something here
...
...     return d
```

Let's practice together by writing a a function called get_counts that takes a list of numbers and returns a dictionary containing the counts of each of the numbers.

Practice time (cont')

- There are two cases we need to contend with:
  1. If the number `num` isn't in the dictionary:
     - It's our first time seeing it, so `d[num] = 1`
  2. If the number `num` is in the dictionary:
     - We need to increment the existing counter by 1:
       - `d[num] = d[num] + 1`
       - or we could also write:
         - `d[num] += 1`

There are two cases we need to consider: 1) what happens if we haven't encountered a number before (that is this is the first time we see it), and what happens if we have seen it before (we need to increase its counter by 1).

dictionaries.py

- Look at the `get_counts` function which applies this key idea.

```
>>> data = read_numbers("numbers.txt")
>>> data
[93, 27, 44, 32, 50, 60, 31, 37, 43, 73, 14, 72, 26, 73, 6, 60, 12, 40, 68, 79, 49, 71, 10, 63, 9
>>> get_counts(data)
{93: 1, 27: 2, 44: 1, 32: 1, 50: 1, 60: 3, 31: 1, 37: 4, 43: 1, 73: 2, 14: 1, 72: 2, 26: 2, 6: 3,
```

dictionaries.py contains a get_counts function that applies this key idea.

Iterating over dictionaries

- We're almost to the point where we can find the most frequent value.
  - Next, we need to go through all of the values in the dictionary to find the most frequent one.
  - There are many ways we could iterate over the things in a dictionary:
    - iterate over the values, or
    - iterate over the keys, or
    - iterate over the key/value pairs
  - Which one is most common?
    - since lookups are done based on the keys, iterating over the keys is the most common

```
for key in dictionary:
  value = dictionary[key]
```

      - `key` will get associated with each key in the dictionary.

We're almost to the point where we can find the most frequent value. We could iterate through the dictionary over its values, keys, or key/value pairs. Iterating through its keys is the most common approach, so we will go through each key, to get its associated value, and find the most frequent one.

dictionaries.py

- Look at the `print_counts` function.
  - `\t` is the tab character

```
>>> data = read_numbers('numbers.txt')
>>> counts = get_counts(data)
>>> print_counts(counts)
93  1
27  2
44  1
32  1
50  1
60  3
31  1
```

  - Notice that the keys are not in numerical order. In general, there's no guarantee about the ordering of the keys, only that you'll iterate over all of them.

Let's look now at print_counts. Notice that the keys are not in numerical order. In general, there's no guarantee about the ordering of the keys, only that you'll iterate over all of them.

dictionaries.py

- Look at the `get_most_frequent` function.
  - It might also be useful to not only get the most frequent value, but also how frequent it is.
  - Anytime you want to return more than one value from a function, a tuple is often a good option.
  - We now return a tuple and also include the `max_value` in addition to `max_key`.

```
>>> data = read_numbers("numbers.txt")
>>> get_most_frequent(data)
(37, 4)
```

get_most_frequent will return a tuple of the key with the max value.

Lecture 10: Dictionaries and recursion

- Administrative
- Dictionaries
- Recursion

Let's switch gears by having a preview of recursion.

The call stack

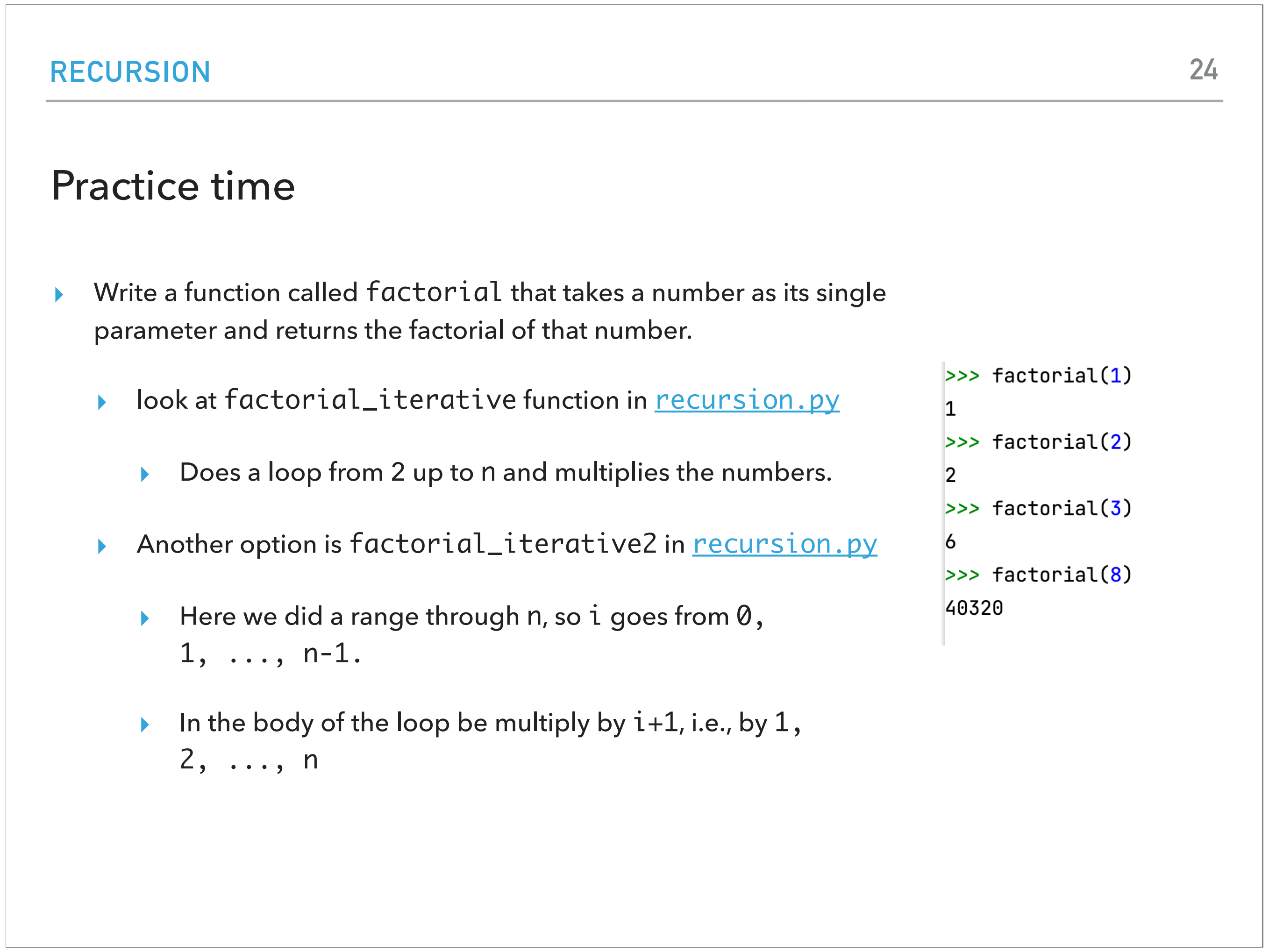
- What is displayed if we call mystery(2, 3) in `call_stack.py` code?
  - `The mystery number is: 15`
  - We can visualize each of these function calls:
    - `mystery(2, 3)`
      - `"The mystery number is: " + str(c(2, 3))`
        - `b(6) - 1`
          - `6 + a()`
            - `10`
          - `6 + 10`
        - `16 - 1`
      - `"The mystery number is: 15"`
  - The way that the computer keeps track of all of this is called the “stack"
    - As functions are called, the stack grows. New function calls are added onto the stack. When functions finish, the stack shrinks and the function call is removed. The result is given to the next function on the stack (the function that called it).

We will start by looking at what happens when we call the function mystery(2,3) in call_stack. There are multiple steps that will eventually return The mystery number is: 15”. The way that the computer keeps track of all of this is called the “stack”. As functions are called, the stack grows. New function calls are added onto the stack. When functions finish, the stack shrinks and the function call is removed. The result is given to the next function on the stack (the function that called it).

Seeing the call stack

- We can actually see the call stack either by using the debugger or by introducing an error, such as changing the `return` statement in function *a* to `return 10 + " "` (we can't add ints and strs).
-
```
Traceback (most recent call last):
  File "/Users/apaa2017/workspace/cs51A/2022sp/Lecture10/call_stack.py", line 15, in <module>
    mystery(2, 3)
  File "/Users/apaa2017/workspace/cs51A/2022sp/Lecture10/call_stack.py", line 13, in mystery
    print("The mystery number is: " + str(c(num1, num2)))
  File "/Users/apaa2017/workspace/cs51A/2022sp/Lecture10/call_stack.py", line 10, in c
    return b(num1 * num2) - 1
  File "/Users/apaa2017/workspace/cs51A/2022sp/Lecture10/call_stack.py", line 7, in b
    return num + a()
  File "/Users/apaa2017/workspace/cs51A/2022sp/Lecture10/call_stack.py", line 4, in a
    return 10 + ""
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

We can actually see the call stack either by using the debugger or by introducing an error, such as changing the return statement in function a to return 10 + " " (remember, we can't add ints and strs).

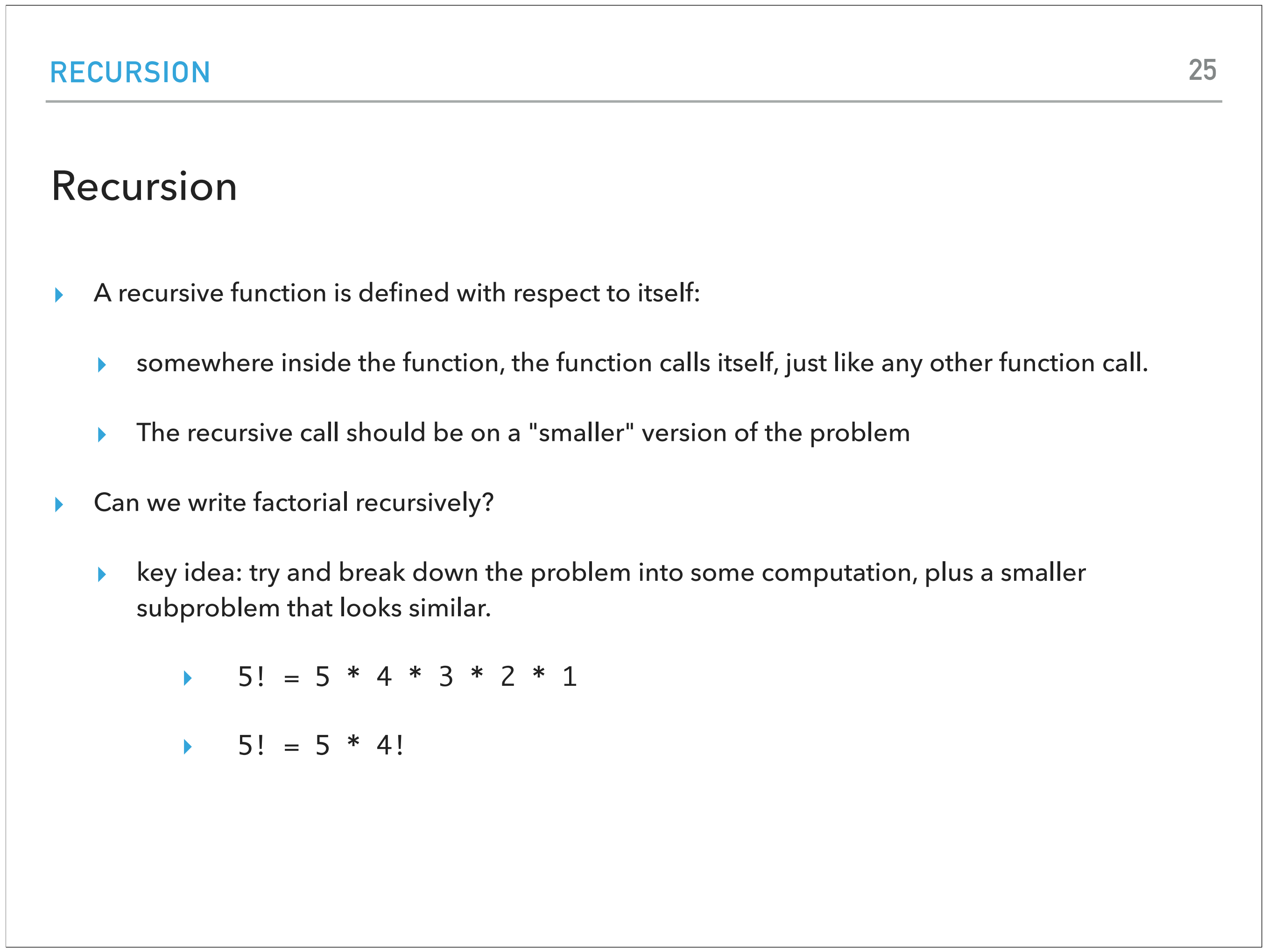


Practice time

- Write a function called `factorial` that takes a number as its single parameter and returns the factorial of that number.
  - look at `factorial_iterative` function in [recursion.py]
    - Does a loop from 2 up to `n` and multiplies the numbers.
  - Another option is `factorial_iterative2` in [recursion.py]
    - Here we did a range through `n`, so `i` goes from `0, 1, ..., n-1`.
    - In the body of the loop be multiply by `i+1`, i.e., by `1, 2, ..., n`

```
>>> factorial(1)
1
>>> factorial(2)
2
>>> factorial(3)
6
>>> factorial(8)
40320
```

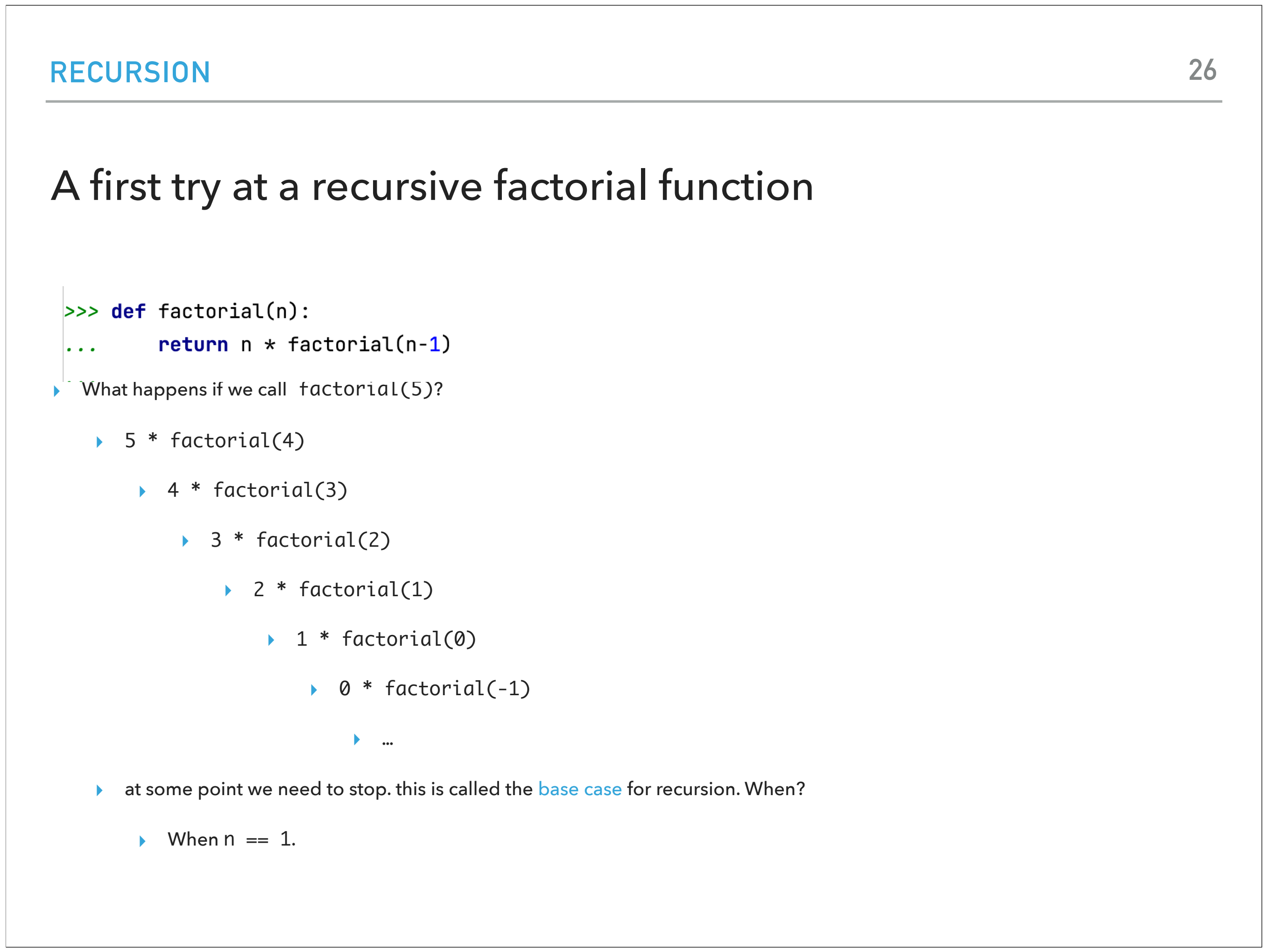
Now we will practice by writing a function called factorial that calculates the factorial of a given number. There are two obvious ways you could go about it, both including a for loop. The difference is what you pass to the range function and what you pass in the intermediate product.



Recursion

- A recursive function is defined with respect to itself:
  - somewhere inside the function, the function calls itself, just like any other function call.
  - The recursive call should be on a "smaller" version of the problem
- Can we write factorial recursively?
  - key idea: try and break down the problem into some computation, plus a smaller subproblem that looks similar.
    - `5! = 5 * 4 * 3 * 2 * 1`
    - `5! = 5 * 4!`

There is a third way we could write this function and it is recursive. A recursive function is defined with respect to itself: somewhere inside the function, the function calls itself, just like any other function call.
The recursive call should be on a "smaller" version of the problem. We can see that we could break 5! As 5*4!.

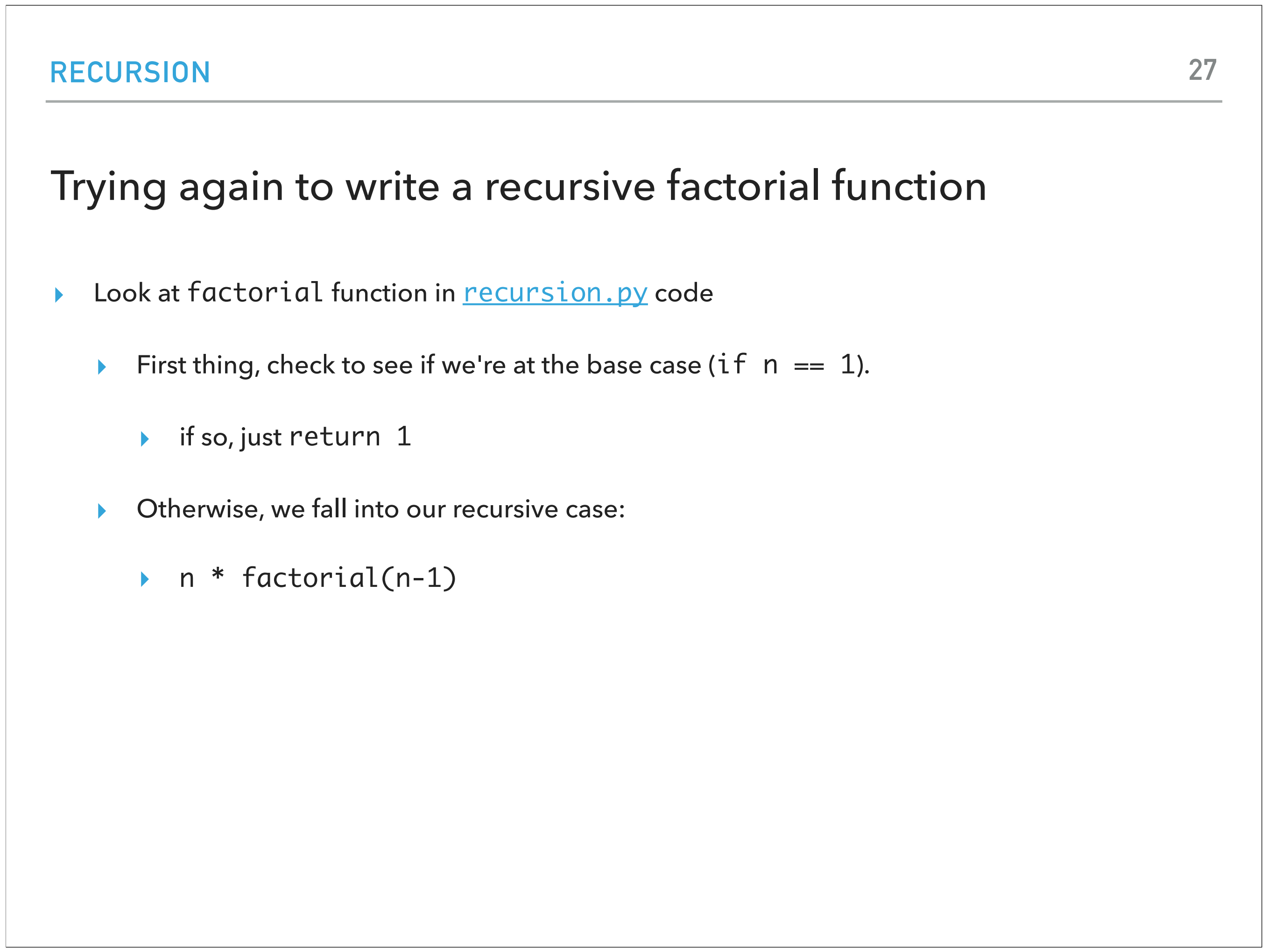


A first try at a recursive factorial function

```
>>> def factorial(n):
...     return n * factorial(n-1)
```

- What happens if we call `factorial(5)`?
  - `5 * factorial(4)`
    - `4 * factorial(3)`
      - `3 * factorial(2)`
        - `2 * factorial(1)`
          - `1 * factorial(0)`
            - `0 * factorial(-1)`
              - …
  - at some point we need to stop. this is called the base case for recursion. When?
    - When `n == 1`.

A logical step would be to try writing the function factorial(n) by returning n*factorial(n-1). The problem is that our code would never terminate because we would keep calling factorial to even instances that don't make sense (-1, -2, etc). We need to stop at what is called the base case of the recursion, in our case, when n==1.



Trying again to write a recursive factorial function

- Look at `factorial` function in [`recursion.py`]() code
  - First thing, check to see if we're at the base case (`if n == 1`).
    - if so, just `return 1`
  - Otherwise, we fall into our recursive case:
    - `n * factorial(n-1)`

We will start exactly with this check and return 1 when we hit the base case. Otherwise, we can recursively call this function to n-1.

Resources

- Textbook: Chapter 16
- `numbers.txt`
- `dictionaries.py`
- `call_stack.py`
- `recursion.py`

Practice Problems

- Practice 6 (solutions), Practice 7 (solutions)

Homework

- Assignment 5