Final Review Session

Brahm Capoor

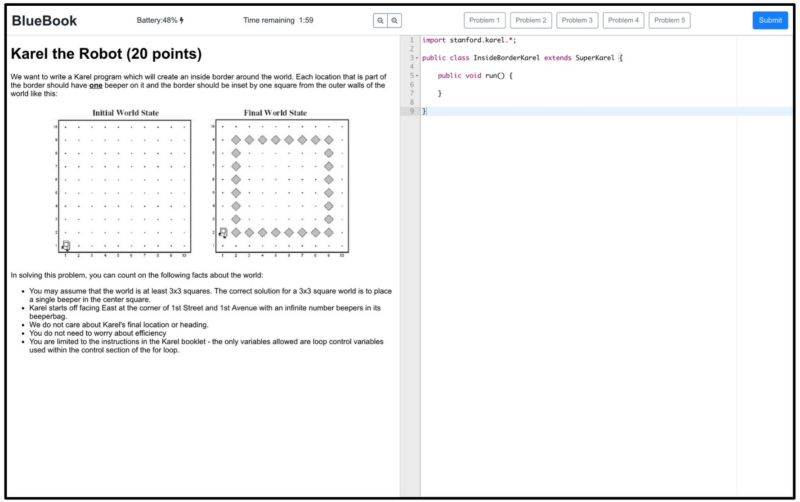
BlueBook

Download for Mac here

Download for Windows here

Handout here

Make sure to have it installed and set up before the exam



Where to find practice problems

Final study resources handout

Section Handouts (especially this week's)

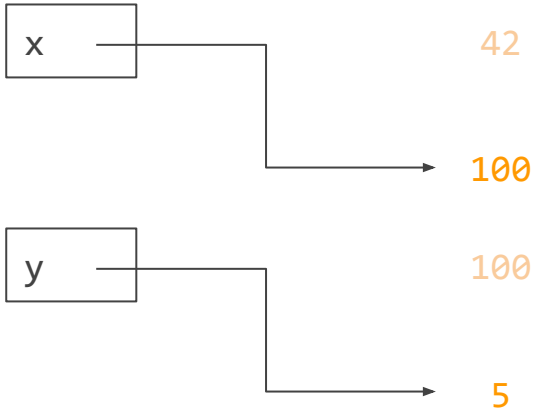
Scattered throughout these slides

Lecture slides and homework

Midterm Review Greatest Hits

Variables: how we store information in a program

```
x = 42  # assigning a variable
x = 100 # reassigning the variable
y = x   # copying the value from
        # x into y
y = 5   # we only change y, not x
```



When we define a function, we make two promises:

1. The inputs, or parameters, to the function
2. What we're returning

```
def my_function(a, b):
    a += 2
    b = 7
    c = a * b
    return c + 1
```

Other useful things to know about functions

Functions can't see each other's variables unless they're passed in as parameters or returned

As a consequence, it's fine to have variables with the same name in different functions

A function can only change its own variables, and not those of its caller

Printing vs Returning

Programs have a information flow, and a text output area, and those are separate.

- When a function returns something, that's information flowing out of the function to another function
- When a function prints something, that's information being displayed on the text output area (which is usually the terminal)

A useful metaphor is viewing a function as a painter inside a room

- Returning is like the painter leaving the room and telling you something
- Printing is like the painter hanging a painting inside the room
- The painter can do either of those things without affecting whether they do the other thing

Printing is sometimes described as a *side effect*, since it doesn't directly influence the flow of information in a program

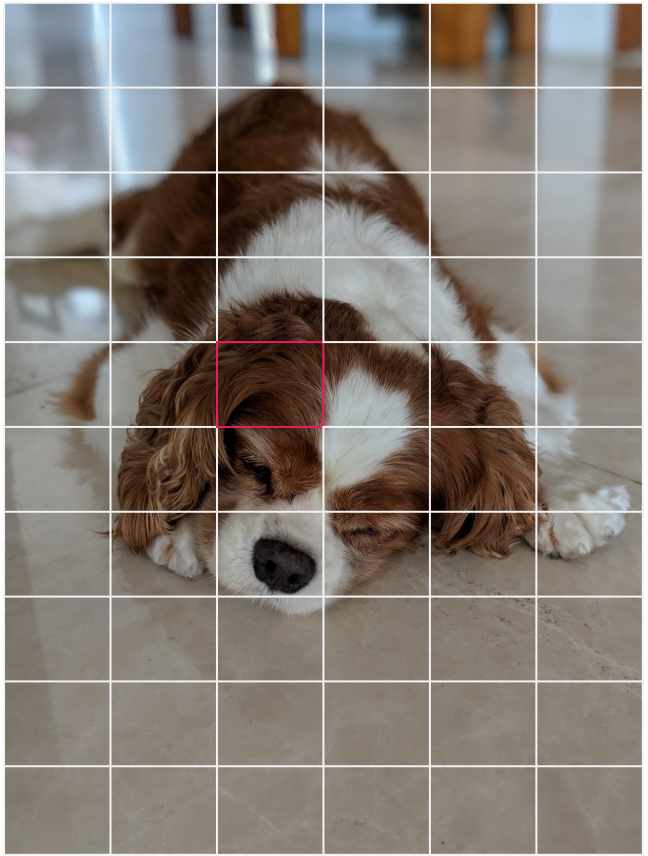
Images

```
img = SimpleImage('buddy.png')

height = img.height  # 10

width = image.width  # 6

pixel = img.get_pixel(2, 4)
```



Pixels

A pixel represents a single color, and is decomposed into three components, each of which is out of 255:

- How red the color is
- How green the color is
- How blue the color is

```
pixel = img.get_pixel(42, 100)

red_component = pixel.red

pixel.green = 42

pixel.blue = red_component // 2
```

Pixels

A pixel represents a single color, and is decomposed into three components, each of which is out of 255:

- How red the color is
- How green the color is
- How blue the color is

```
pixel = img.get_pixel(42, 100)

red_component = pixel.red

pixel.green = 42

pixel.blue = red_component // 2
```

Since the midterm: you can also call `img.get_pix` and `img.set_pix` if you'd like to operate on rgb tuples rather than Pixel objects.

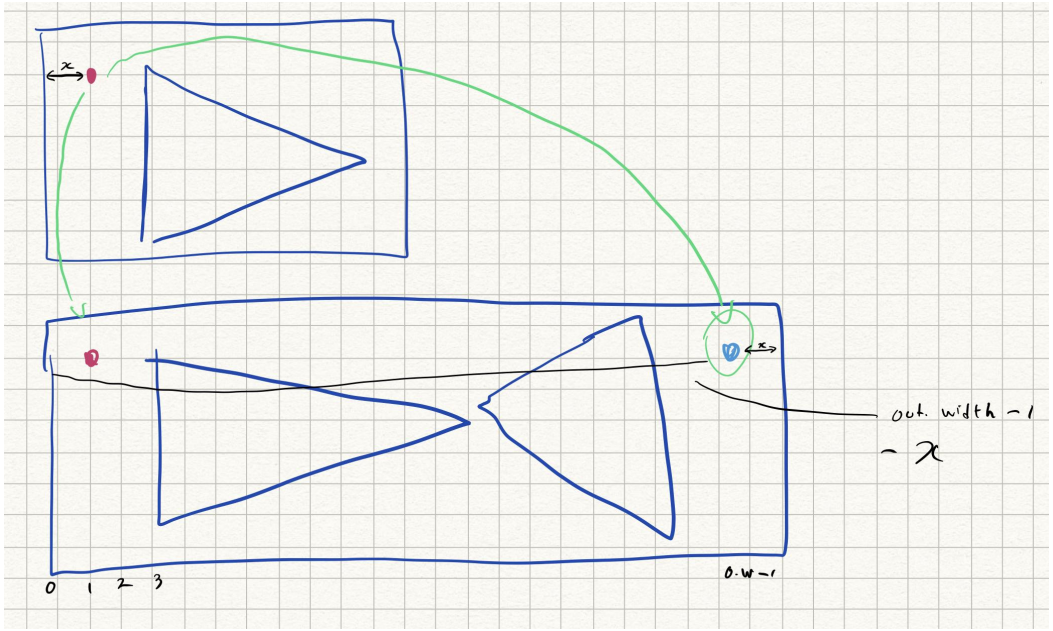
Two common code patterns for images

```
for pixel in image:
    # we don't have access to the coordinates of pixel

for y in range(image.height):
    for x in range(image.width):
        pixel = image.get_pixel(x, y)
        # now we do have access to the coordinates of pixel
```

Both of these loops go over the pixels in the same order

Whenever you're calculating coordinates, drawing diagrams is your best course of action, like so:



(This is an actual diagram I drew in office hours to explain the mirror problem! Your diagram doesn't need to be neat, it just needs help you calculate coordinates)

A string is a variable type representing some arbitrary text:

```
s = 'We demand rigidly defined areas of doubt and uncertainty!'
```

and consists of a sequence of characters, which are indexed starting at 0.

```
eighth_char = s[7]  # 'n'
```

We can slice out arbitrary substrings by specifying the start and end indices:

```
string_part = s[10:20]  # 'rigidly de'
```

The start index is inclusive, and can be omitted if you want to start at the beginning of `s`

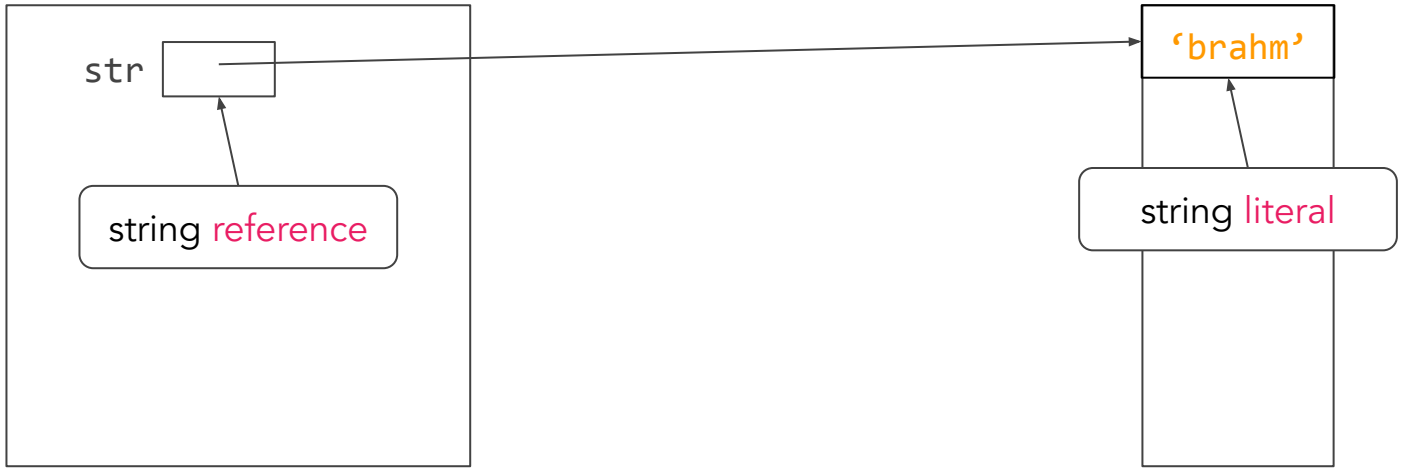
The end index is exclusive, and can be omitted if you want to go until the end of `s`

Useful String functions

```
>>> s = ‘So long, and thanks for all the fish’
>>> len(s)
36
>>> s.find(‘,’)     # returns the first index of the character
7
>>> s.find(‘z’)
-1                  # returns -1 if character isn’t present
>>> s.find(‘n’, 6)  # start searching from index 6
10
>>> s.lower()       # islower() also exists, returns true if all lowercase
‘so long, and thanks for all the fish’  # returns a string, doesn’t modify
>>> s.upper()       # isupper() also exists, returns true if all uppercase
‘SO LONG, AND THANKS FOR ALL THE FISH’
```

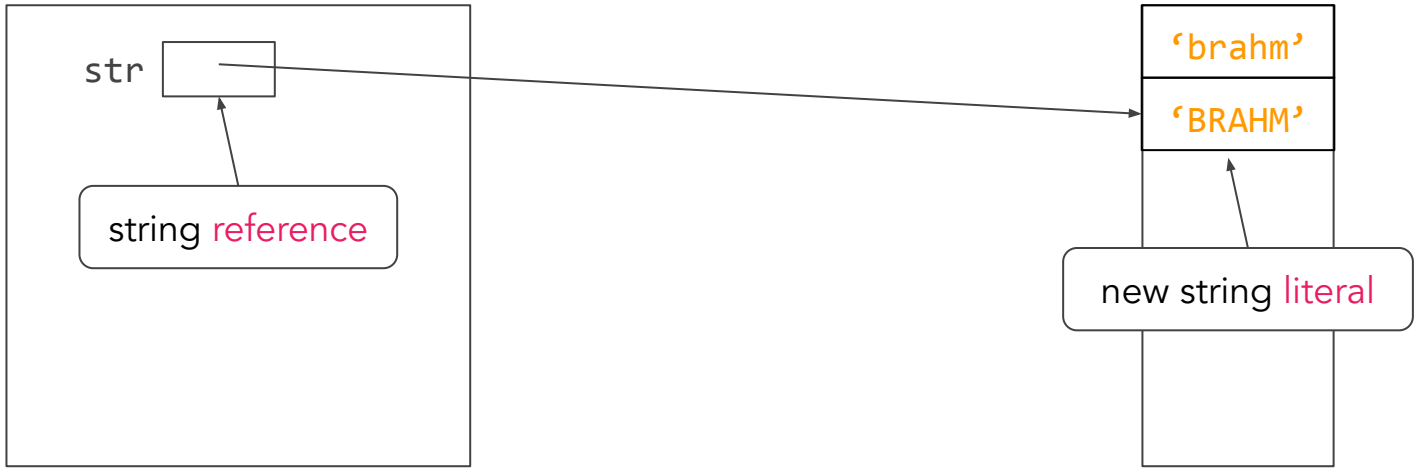
An important nuance: string literals are immutable

```
str = 'brahm'
```



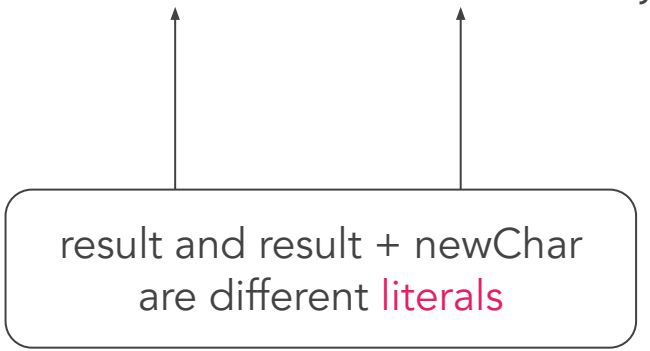
...but references aren't!

```
str = str.upper()
```



This leads to a common pattern for String problems

```
s = ‘banter’
result = ‘’
for i in range(len(s)):
     ch = s[i]
     newChar = # process ch
     result = result + newChar;
```



Reading Files

```
def process_file_line_by_line(filename):
    with open(filename, ‘r’) as f:
        for line in f:
            # process the line
```

Reading Files

```
def process_file_line_by_line(filename):
    with open(filename, 'r') as f:
        for line in f:
            # process the line
```

```
def process_file_with_list(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    # we can now index into and slice out
    # lines
    first_line = lines[0]

    for line in lines[1:]:
        print(line)
```

```
def process_whole_text(filename):
    with open(filename, 'r') as f:
        text = f.read()

    words = text.split()
    # sick code here
```

Reading Files

```
def process_file_line_by_line(filename):
    with open(filename, 'r') as f:
        for line in f:
            # process the line


def process_file_with_list(filename):
    with open(filename, 'r') as f:
        lines = f.readlines()
    # we can now index into and slice out
    # lines
    first_line = lines[0]

    for line in lines[1:]:
        print(line)
```

```
def process_whole_text(filename):
    with open(filename, 'r') as f:
        text = f.read()

    words = text.split()
    # sick code here
```

Advice you didn’t expect to hear tonight: consider `stripping`

Advice you didn't expect to hear tonight: consider `stripping`

(3 different people advised me against that joke, but this way, you'll never forget your `\n`s)

A list is a variable type representing a ~~list~~ linear collection of elements of any type:

```
num_list = [4, 2, 1, 3]
str_list = [‘ax’, ‘bc’]
```

They work pretty much exactly the same way as strings:

```
>>> len(num_list)
4
>>> str_list[1]
‘bc’
>>> num_list[1 : 3]
[2, 1]
```

You can make an empty list using square brackets

```
lst = []
```

And then stick stuff in it using the `append` method:

```
for i in range(5):
    lst.append(i)
# lst now is [0, 1, 2, 3, 4]
```

You can also stick another list at the end using the `extend` method:

```
second_lst = [8, 9, 10]
lst.extend(second_list)  # lst is now [0, 1, 2, 3, 4, 8, 9, 10]
```

Note that each of these functions modifies the list, rather than returning a new one.

You can also sort a list:

```
nums = [0, 9, 4, 5]
nums = sorted(nums)
```

and print it out:

```
print(nums)  # prints [0, 9, 4, 5]
```

and check whether it contains an element:

```
>>> 3 in nums
False
```

You can also put lists inside other lists!

```
>>> lst = []
>>> lst.append([1, 2, 3])
>>> lst.append([42, 100])
>>> print(lst)
[[1, 2, 3], [42, 100])
```



Parsing

Unfortunately, there's no one-slide summary of parsing that can do it justice: check out slides 142-164 in the midterm review session to go through a detailed walkthrough!

Dictionaries and Counting

Dictionaries allow us to build one-way associations between one kind of data (which we call the key) and another (which we call the value). A common metaphor for them is a phone book, whose keys are people's names and whose values are their phone numbers. It's super easy to look someone's number up if you know their name, but harder to do it the other way around.

```
>>> d = { }                 # make an empty dictionary
>>> d['brahm'] = 42         # associate the key 'brahm' with the value 42
>>> d['nick'] = 5           # associate the key 'nick' with the value 5
>>> d['nick'] = 8           # change the value for 'nick' to be 8
                            # since keys need to be unique
>>> d['brahm']              # get the value associated with 'brahm'
42
>>>  'python' in d          # check whether a particular key is in the map
False
```

The dict-count algorithm

One of the most important uses of dictionaries is using them to count the occurrences of other things, since they allow us to directly associate things with their frequencies.

It's so important that there's a pretty generic piece of code we can use when solving a problem like this. Let's make sure we understand how it works.

The algorithm

```
def count_the_things(things_to_count):
```

The general problem setup is thus: we have a collection of things we want to count (this could be a file, or a list, or a string), and want to print out the frequency of each thing.

The algorithm

```
def count_the_things(things_to_count):
    counts = {}
```

First, we set up a `counts` dictionary, which will associate each thing (as a key) with the number of times it occurs (as a value)

The algorithm

```
def count_the_things(things_to_count):
    counts = {}

    for thing in things_to_count:
```

Then, we just loop through each thing in the collection. This looks a little different based on what the collection actually is, and we'll assume that it's a list here.

The algorithm

```
def count_the_things(things_to_count):
    counts = {}

    for thing in things_to_count:
        if thing not in counts:
            counts[thing] = 0
```

If we haven't seen this particular thing before, we need to make sure that it's a key in the map, so we stick it in there and associate it with a 0.

The algorithm

```
def count_the_things(things_to_count):
    counts = {}

    for thing in things_to_count:
        if thing not in counts:
            counts[thing] = 0
        counts[thing] += 1
```

Now, because we've seen the thing, we need to increment the count in our `counts` dictionary.

The algorithm

```
def count_the_things(things_to_count):
    counts = {}

    for thing in things_to_count:
        if thing not in counts:
            counts[thing] = 0
        counts[thing] += 1

    for thing in sorted(counts.keys()):
```

Once we've gone through all the things, we're going to print their frequencies in sorted order (which would be alphabetical for string keys and numerical for int keys). Let's loop through the sorted keys of `counts`.

The algorithm

```
def count_the_things(things_to_count):
    counts = {}

    for thing in things_to_count:
        if thing not in counts:
            counts[thing] = 0
        counts[thing] += 1

    for thing in sorted(counts.keys()):
        print(thing, counts[thing])
```

Then, we just print the thing and how often it occurs!

Tuples

A tuple is a data structure that's kind of like a list, with a few key differences:

- It's immutable, so once we create it, we can't change its elements
- It's fixed size (so we can't add or remove elements)

We create it by surround its elements with parentheses and separating them with commas, like so:

```
my_favorite_tuple = (42, 100)
tuple_with_two_types = (22, 'banter', 35)
```

All of the normal list indexing and slicing operations work on tuples as well:

```
>>> my_favorite_tuple[0]
42
>>> tuple_with_two_types[1:]  # it's unlikely you'd need to do this
('banter', 35)
```

We'll talk more about nesting data structures later, but you can make lists of tuples and sort them, like so:

```
>>> lst = [(5, 6), (2, 3), (1, 2), (2, 4)]
>>> sorted(lst)
[(1, 2), (2, 3), (2, 4), (5, 6)]
```

Tuples are sorted by looking at their first elements, and ties are broken by looking at subsequent elements.

Nesting Data Structures

You might encounter many different kinds of questions involving nested data structures:

- Nested dictionaries (like in babynames, the tweets project and the facebook lecture example)
- Dictionaries with lists as values (like in mimic)
- Dictionaries with lists of tuples as values (like in homework 8)

Each of these structures is really useful and it's worth taking the time to understand exactly what each level of the nesting means.

Once you have that understanding, there are a few things you can do to make the problem easier for yourself.

Tip #1: Decomposing through variable names

The idea here is to store intermediate levels of your dictionary in variables rather than to pile [ ]s on top of one another:

Tip #1: Decomposing through variable names

The idea here is to store intermediate levels of your dictionary in variables rather than to pile []s on top of one another:

```
def add_one_to_count(counts, counted, month):
    if counted not in counts:
        counts[counted] = {}

    if month not in counts[counted]:
        counts[counted][month] = 0

    counts[counted][month] += 1

    return counts
```

This function does what it's supposed to, but it's a little unclear what each level of the dictionary represents and this could trip you up in an exam

Tip #1: Decomposing through variable names

The idea here is to store intermediate levels of your dictionary in variables rather than to pile []s on top of one another:

```
def add_one_to_count(counts, counted, month):
    if counts not in counted:
        counted[counts] = {}
    inner_counts = counts[counted]
    if month not in inner_counts:
        inner_counts[month] = 0

    inner_counts[month] += 1

    return counted
```

The introduction of this `inner_counts` variable makes it clearer to you what exactly the inside dictionary is supposed to be, which makes like a little less confusing.

Tip #1: Decomposing through variable names

The idea here is to store intermediate levels of your dictionary in variables rather than to pile []s on top of one another:

```
def add_one_to_count(counts, counted, month):
    if counts not in counted:
        counted[counts] = {}
    inner_counts = counts[counted]
    if month not in inner_counts:
        inner_counts[month] = 0

    inner_counts[month] += 1

    return counted
```

It sounds like a small thing, but I promise that it *really* helps.

Why does this work?

The key idea to remember is that these collections don't return back copies of their elements, but the elements themselves (Python never copies unless you explicitly tell it to!)

```
>>> d = {'a' : [1, 2, 3], 'b': [7, 8, 9], 'x': [42, 100, 105]}
>> lst = d['x']
>>> lst.append(64)
>>> d
{'a' : [1, 2, 3], 'b': [7, 8, 9], 'x': [42, 100, 105, 64]}
```

And now: a demo in the terminal

if you're looking at the slides afterwards, check out the review session recording for the demo!

Applying it to a practice problem

```
def make_county(words):
    """
    Given a list of non-empty words, produce 'county' dict
    where each first-char-of-word seen is a key, and its value
    is a count dict of words starting with that char.
    So ['aaa', 'abb', 'aaa'] yields {'a': {'aaa':2, 'abb':1}}
    >>> make_county(['aaa', 'abb', 'aaa'])
    {'a': {'aaa':2, 'abb':1}}
    """
```

Applying it to a practice problem

```
def make_county(words):
    """
    Given a list of non-empty words, produce 'county' dict
    where each first-char-of-word seen is a key, and its value
    is a count dict of words starting with that char.
    So ['aaa', 'abb', 'aaa'] yields {'a': {'aaa':2, 'abb':1}}
    >>> make_county(['aaa', 'abb', 'aaa'])
    {'a': {'aaa':2, 'abb':1}}
    """
```

```
def make_county(words):
    counts = {}


    return counts
```

Let's start simple: we know we need to make a dictionary, so let's do that.

```
def make_county(words):
    counts = {}

    for word in words:
        # sick code here



    return counts
```

We're definitely going to need to process each word in `words`, so let's go ahead and loop through them.

```
def make_county(words):
    counts = {}

    for word in words:
        first_char = word[0]




    return counts
```

Now, we're going to get the first character from the current `word`

```
def make_county(words):
    counts = {}

    for word in words:
        first_char = word[0]


        inner_counts = counts[first_char]




    return counts
```

Let's pretend that `first_char` is definitely in counts, and get the counts dictionary associated with it, using a variable to better decompose our solution.

```
def make_county(words):
    counts = {}

    for word in words:
        first_char = word[0]
        if first_char not in counts:
            counts[first_char] = {}
        inner_counts = counts[first_char]




    return counts
```

Now, we need to guarantee that first_char will in fact be a key in counts, so let's just stick an empty dictionary in there if we haven't seen it before.

```
def make_county(words):
    counts = {}

    for word in words:
        first_char = word[0]
        if first_char not in counts:
            counts[first_char] = {}
        inner_counts = counts[first_char]


        inner_counts[word] += 1


    return counts
```

Now, let's pretend we've seen `word` before and increment its corresponding count in `inner_counts`

```
def make_county(words):
    counts = {}

    for word in words:
        first_char = word[0]
        if first_char not in counts:
            counts[first_char] = {}
        inner_counts = counts[first_char]
        if word not in inner_counts:
            inner_counts[word] = 0
        inner_counts[word] += 1


    return counts
```

Now, we'll do exactly the same thing with word to make sure that it's in inner_counts, except we'll initialize its count to 0 instead.

What data structure should I use?

What data structure should I use?

First, a disclaimer:

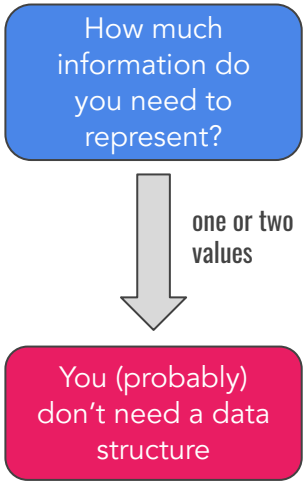
In general in an exam, it will be pretty clear what kind of data structure needs to be used where, or what is passed in, or what will be returned. You can also typically get a hint from the question itself (e.g. the sample output we provide or just a description). All these strategies are totally fine.

That said, unless we explicitly prohibit it or require you to use something else, you can always use a data structure in any problem on an exam if you feel it's appropriate. The next few slides provide a rough framework for deciding which data structure to use, and also shed light on the differences between the 3 big ones we've used thus far.

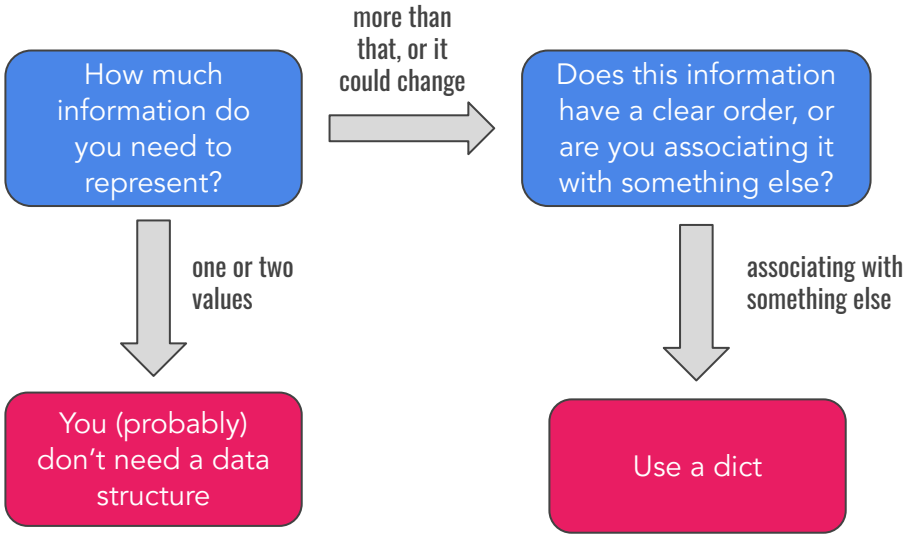
What data structure should I use?

How much information do you need to represent?

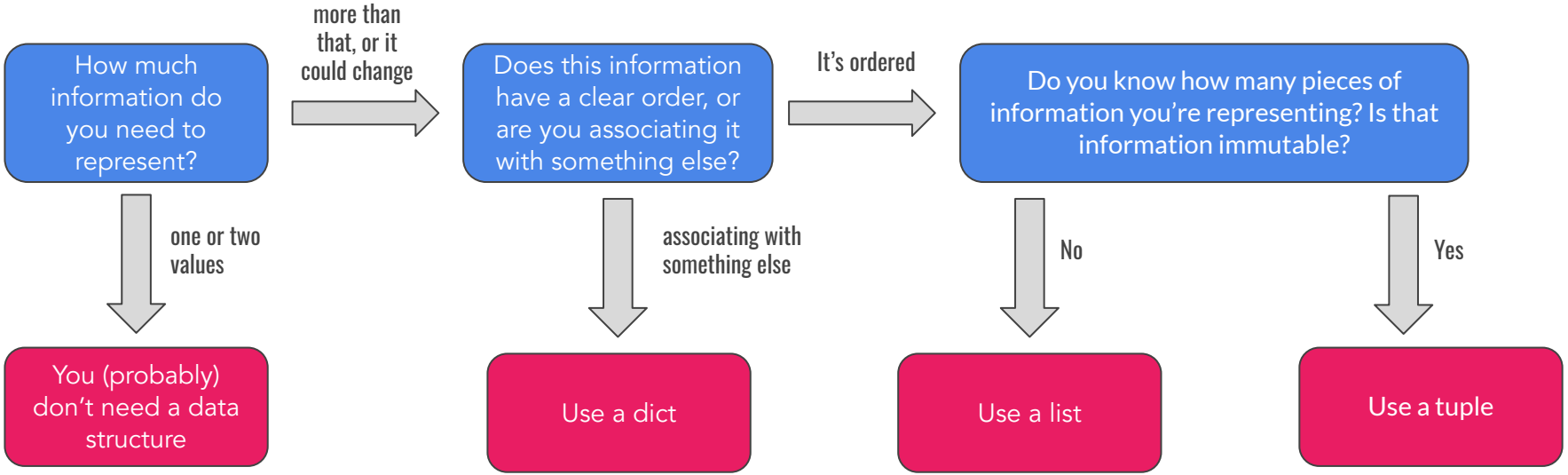
What data structure should I use?



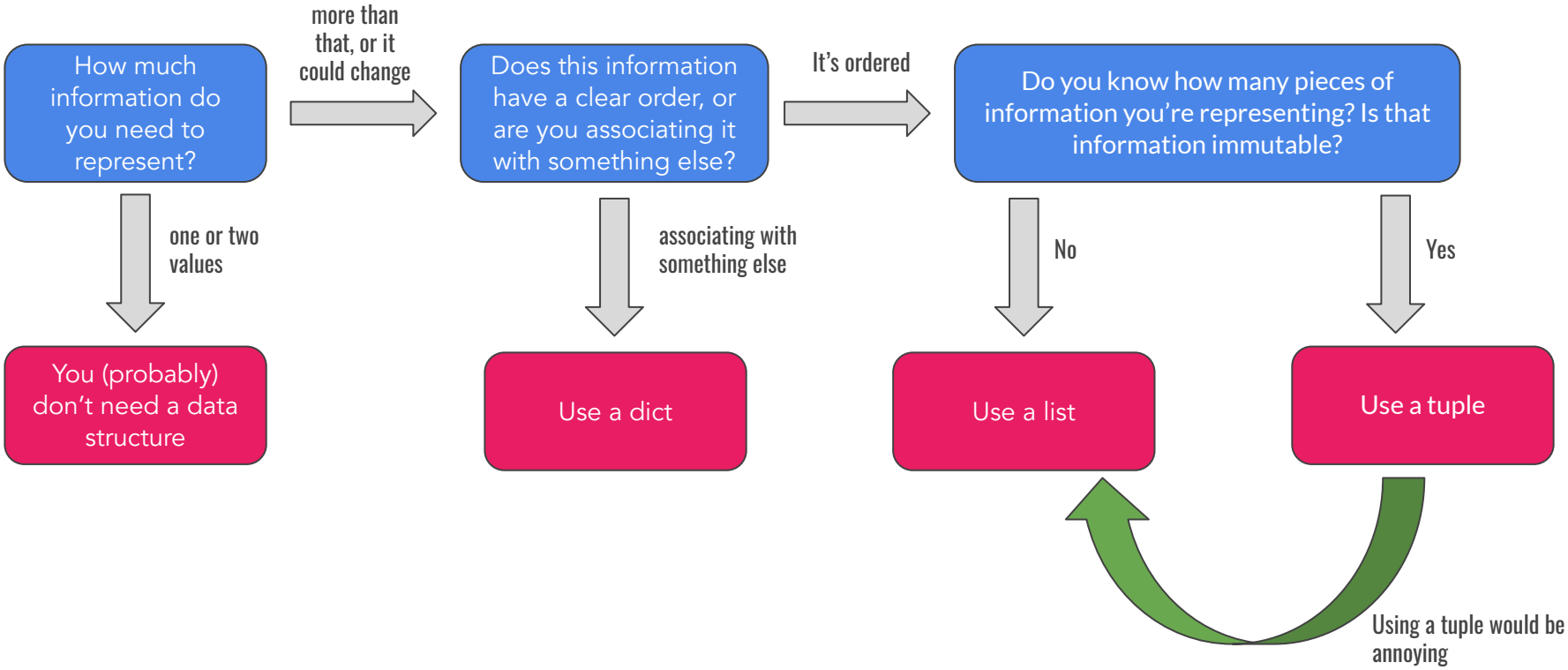
What data structure should I use?



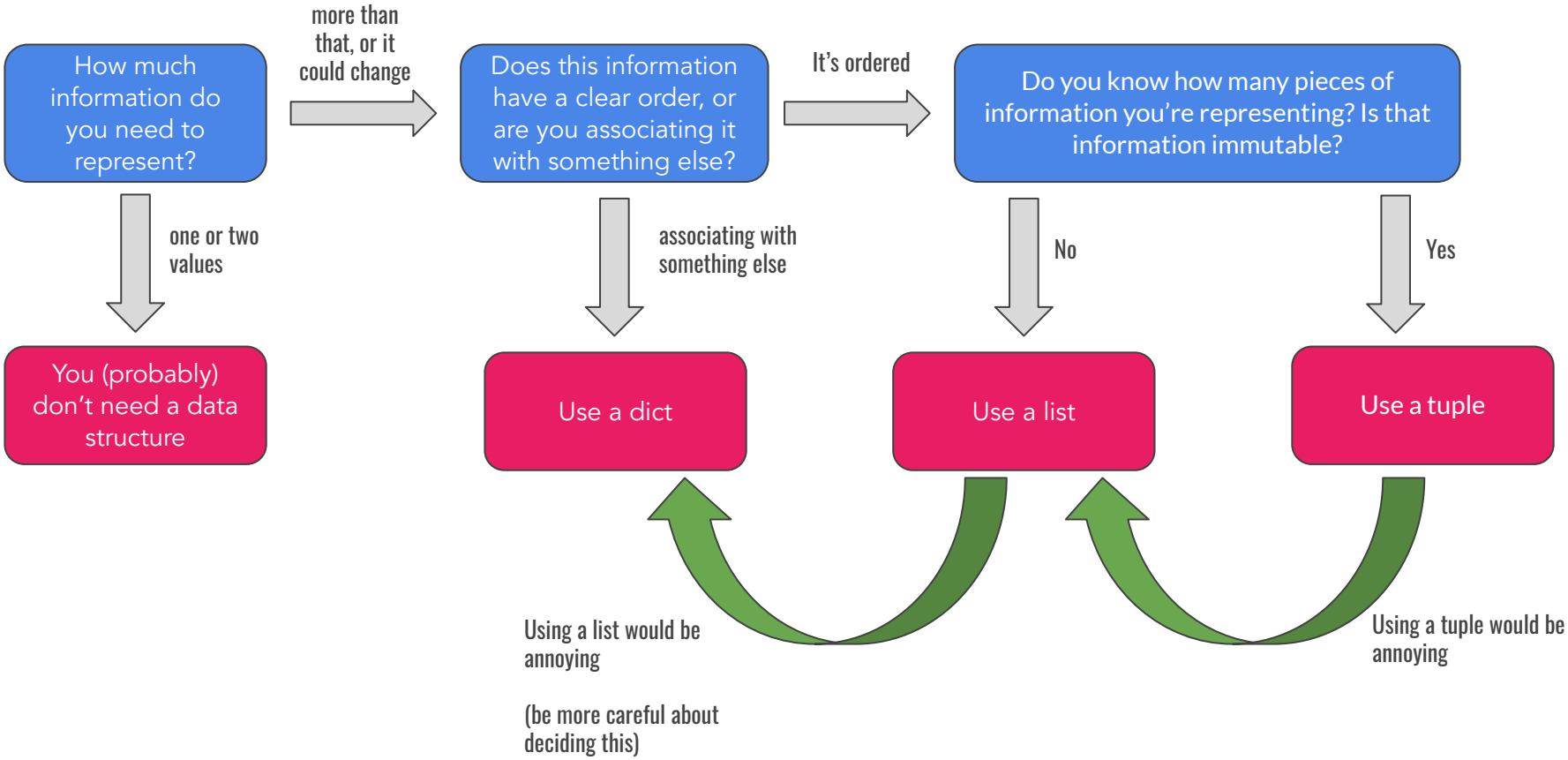
What data structure should I use?



What data structure should I use?



What data structure should I use?



What data structure should I use?

The logic in that flow chart applies recursively to structures nested within one another, as well. For example, you may realize that you need to use a dictionary, but not be certain what the values are. Use this heuristic to help you figure that out!

`tk` and Graphics

Fundamentally, any drawing problem we give you will require that you implement a function with the following signature:

```
def draw_some_stuff(canvas, width, height <other parameters>)
```

You don't need to worry about where `canvas`, `width` and `height` come from (check out the `make_gui` function in the Baby Graphics starter code if you're interested) - your job is simply to call functions on `canvas` to draw what we describe.

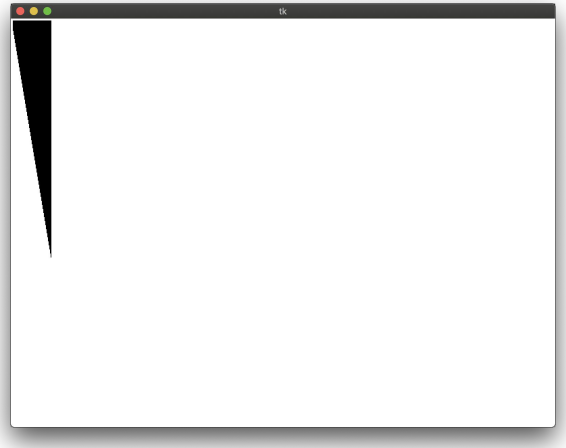
As you can imagine, there is enormous variety in the kinds of graphics problems we could give you, but we're restricting it to one kind of problem: drawing proportionate lines

A problem: triangle drawing

Implement the following function:

```
def draw_triangle(canvas, width, height, n)
```

that takes a canvas and its dimensions as well as an integer `n` as a parameter. Starting at `x=0` and moving rightwards by one pixel each time, draw `n` vertical lines. The line at `x = i` should be `i%` of the the total height of the screen.



```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
```

We know we need to draw **n** lines, so let's stick a `for` loop in here

```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
```

We know we need to draw **n** lines, so let's stick a `for` loop in here

```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
        x_coord = i
```

Let's strategically procrastinate by finding the x-coordinate of the line we're drawing. We're drawing the `i`-th line, so the x-coordinate is `i`.

```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
        x_coord = i
        percent = i / 100
        line_h = height * percent
```

Now, **i** is an integer but we'd like its percentage equivalent, so we'll find that (using float division) and thus the height of this line

```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
        x_coord = i
        percent = i / 100
        line_h = height * percent
        canvas.create_line(x_coord, 0,
                           x_coord, line_h)
```

Now, we draw the line!

```
def draw_triangle(canvas, width, height, n):
    for i in range(n):
        x_coord = i
        percent = i / 100
        line_h = height * percent
        canvas.create_line(x_coord, 0,
                           x_coord, line_h)
```

Calculating the percentage is the important part

An important nuance: coordinates are 0-indexed

An aside: when should you care about off-by-ones?

An aside: when should you care about off-by-ones?

Answer: Always. It doesn't save you that much time to include them, and is a much safer bet than omitting them. That said, when we grade, we're more concerned about cases where those 1s lead to visible differences in output, or crashes. It's not as important in graphics problems, and won't be graded down as much.

`lambdas` and how to use them

A `lambda` is a way of expressing a simple function concisely, and it takes the following form:

```
lambda <parameter_name> : <return_value>
```

Here, *<parameter_name>* is a name for the single parameter passed into this function and *<return_value>* is the expression that the function returns, which itself is likely some operation on *<parameter_name>*.

```
lambda x : 2 * x  # doubles the passed in parameter
lambda s : mixup(s) # calls a mixup function, defined elsewhere
```

You've already seen functions like `sorted`, `max`, and `min`, which process the elements of a list in some way or the other.

All of them require us to compare a pair of elements to see which is greater. We can do so because Python has a default way of comparing variables of the same type:

- Integers and Floats are compared numerically.
- Strings are compared lexicographically (i.e. in alphabetical order).
- Tuples are compared using their first elements, and ties are broken using subsequent elements.

Sometimes, this may not be the comparison we want to be making:

- We might want to order strings by how long they are rather than where they'd be in alphabetical order
- We might want to find the number with the fewest prime factors
- We might only want to sort tuples by their 2nd element

Essentially, we want to specify the property by which we compare elements of the list.

The `sorted`, `max`, and `min`, all have an optional `key` argument, into which we pass a function that extracts the property of interest from an arbitrary element in the list and operate based on it. Theoretically, we could do something like this:

```
def extract_second_element(tup):
    return tup[1]

lst = # some list of tuples
 # get tuple with smallest second element
min_element = min(lst, key=extract_second_element)
```

It's a little strange to define a separate function only to use it once, though, so we'd rather more concisely inline it using a lambda:

```
min_element = min(lst, key=lambda tup : tup[1])
```

List comprehensions

In Mathematics, there's a kind of notation called Set Builder Notation, which allows you to succinctly express all the elements in a set.

"All the square numbers below 100"
"The cube roots of the first 8 even numbers"

List comprehensions are a way of concisely constructing lists in Python that looks a lot like Set Builder notation.

We'll go over the syntax in a sec, but know that List Comprehensions are what is known as syntactic sugar: it's really nice to to use but can totally just be unrolled into a for loop if you'd prefer.

You should understand list comprehensions (and we will ask questions about them), but you should know that they're a substitute, not a unique solution.

The general structure of a list comprehension is as follows:

```
[operation<element> for element in collection if condition<element>]
```

The general structure of a list comprehension is as follows:

```
[operation<element> for element in collection if condition<element>]
```

`collection` is some collection of elements (a list, or a dict's keys, or a tuple, or a string).

The general structure of a list comprehension is as follows:

```
[operation<element> for element in collection if condition<element>]
```

`collection` is some collection of elements (a list, or a dict's keys, or a tuple, or a string).

`element` is a temporary variable used to represent each element in `collection`.

The general structure of a list comprehension is as follows:

```
[operation<element> for element in collection if condition<element>]
```

`collection` is some collection of elements (a list, or a dict's keys, or a tuple, or a string).

`element` is a temporary variable used to represent each element in `collection`.

`operation<element>` is a transformed version of element stored in the output list.

It doesn't need to be a function, but it can be.

The general structure of a list comprehension is as follows:

```
[operation<element> for element in collection if condition<element>]
```

`collection` is some collection of elements (a list, or a dict's keys, or a tuple, or a string).

`element` is a temporary variable used to represent each element in `collection`.

`operation<element>` is a transformed version of element stored in the output list.

It doesn't need to be a function, but it can be.

`condition<element>` is an optional condition we can use to only transform the `element`s in `collection` for which `condition<element>` is true.

If you want to indiscriminately transform all the `element`s in `collection`, you can omit `if condition<element>` from the comprehension.

```
[operation<element> for element in collection if condition<element>]
```

reads in English as:

> `operation<element>` *for each* `element` *in* `collection`, *so long as* `condition<element>` *is true.*

Let's work through a pretty complicated list comprehension together to understand the process: given a dictionary whose keys are integers and whose values are lists of integers, find a list of the squares of each key, provided that the sum of the elements in its corresponding value is an even number. For example, given this dictionary:

```
d = {1 : [1, 2, 4], 5: [4, 4, 8, 0], 42: [42]}
```

the comprehension should return the list `[25, 1764]`.

```
[ for k in d.keys() ]
```

Start by writing the stuff you know is going to be there: a loop through each of the elements

```
[k * k for k in d.keys() ]
```

Next, add the desired operation

```
[k * k for k in d.keys() if sum(d[k]) % 2 == 0]
```

Next, add the condition we want,
and we're done!

“What should I do in these last 4 days?”

“What should I do in these last 4 days?”

Keep practicing! Come to office hours and the LaIR! Post on Piazza! Aim to appreciate the big ideas and the algorithmic details, not the syntax!

“What should I do in these last 4 days?”

Keep practicing! Come to office hours and the LaIR! Post on Piazza! Aim to appreciate the big ideas and the algorithmic details, not the syntax! Sleep!

Parting words