Midterm Review Session

CS106AP

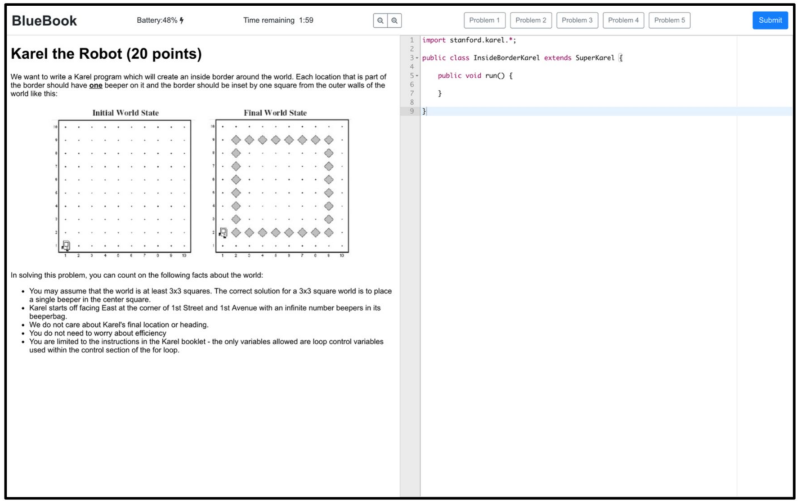
Based on Spring 2019 Midterm Review Slides by Brahm Capoor

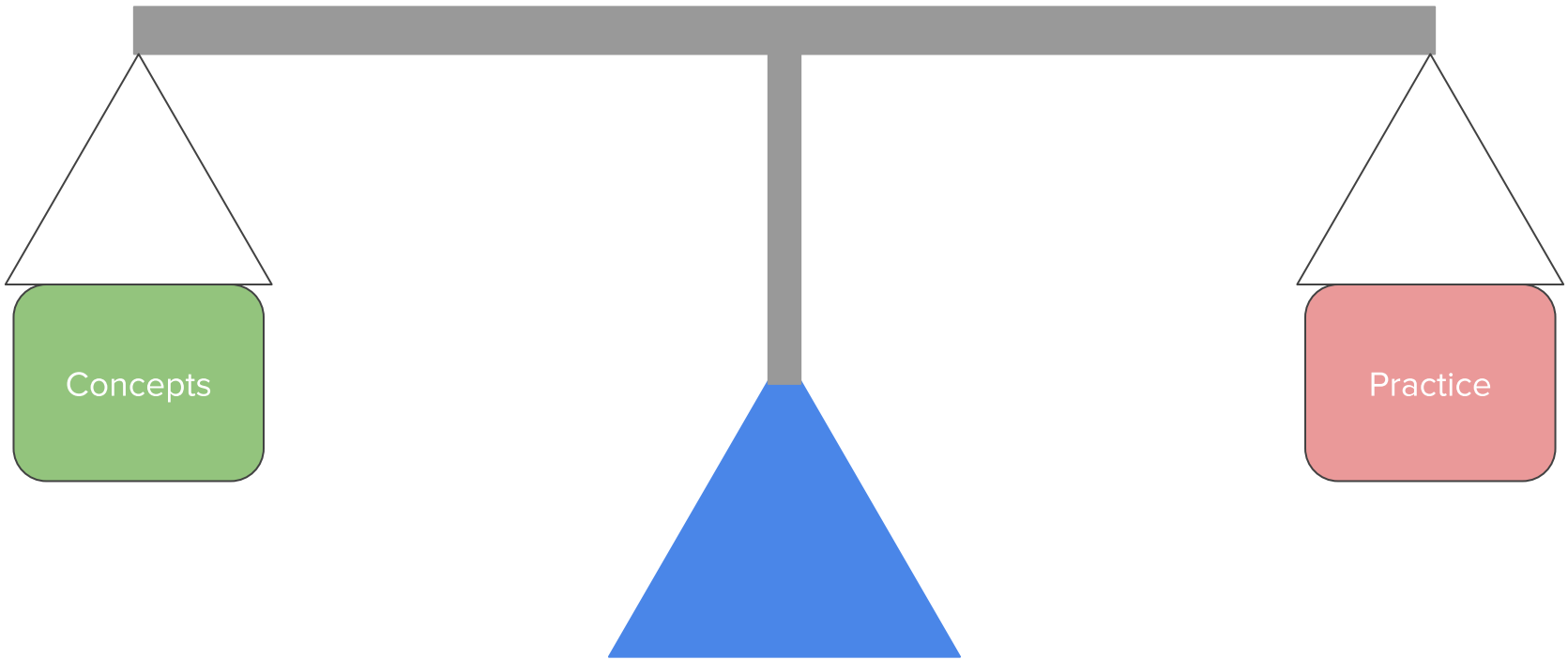
Logistics

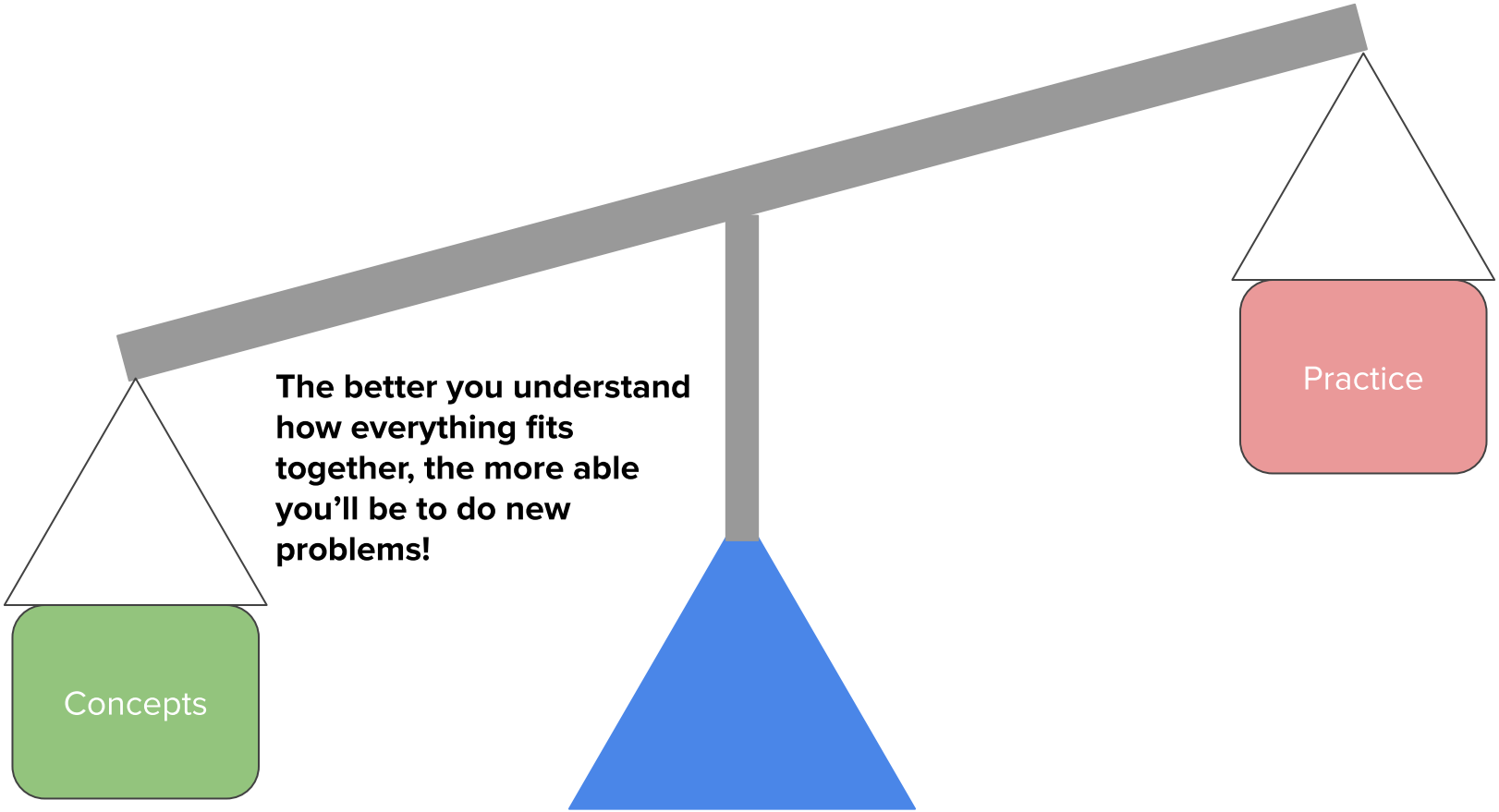
- Monday July 22, 7 to 9pm
- Hewlett 200
- Come a little early!

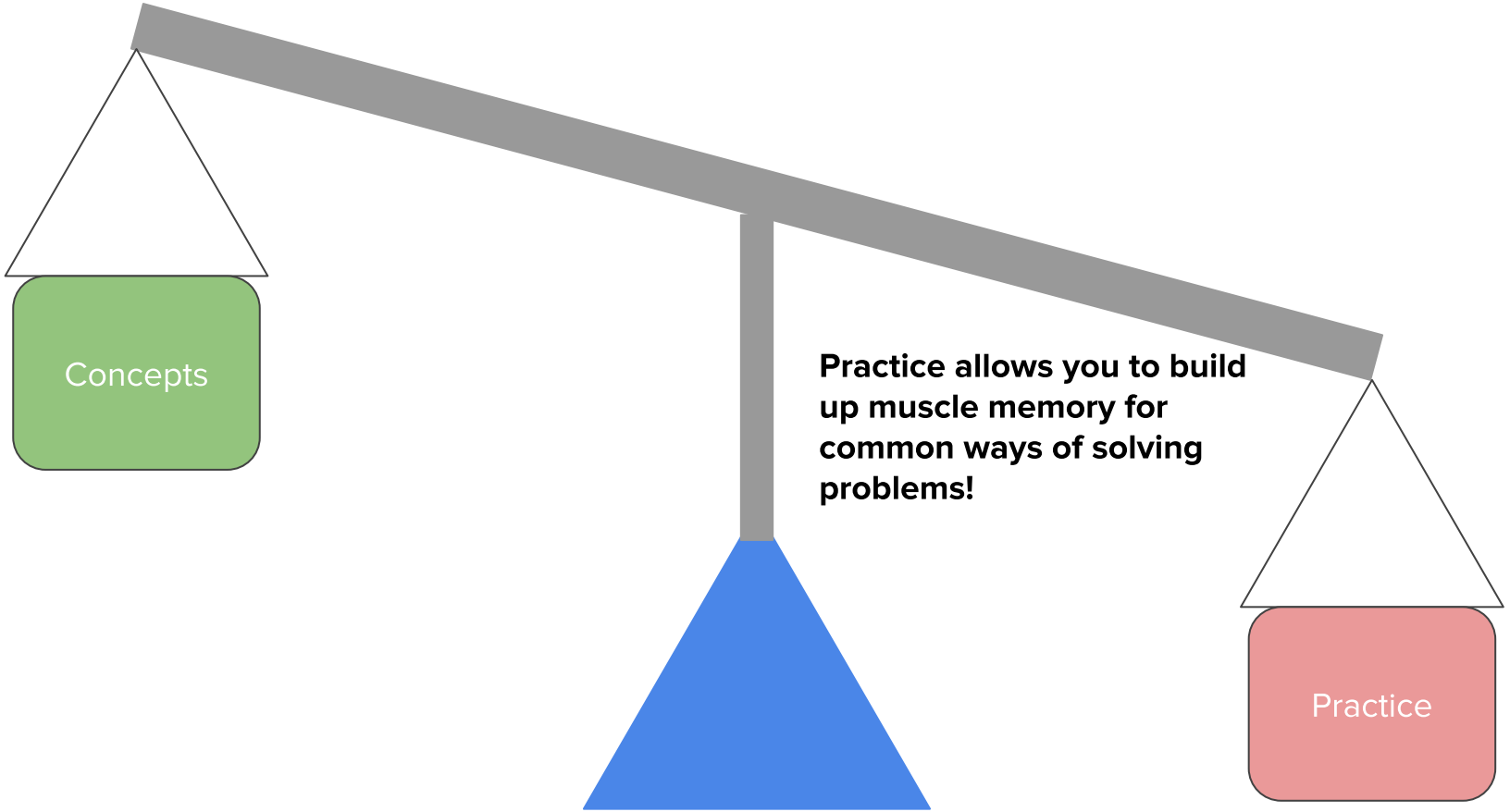
BlueBook

- Download for Mac here
- Download for Windows here
- Handout here
- Make sure to have it installed and set up before the exam




Concepts
Practice


Concepts
The better you understand how everything fits together, the more able you'll be to do new problems!
Practice


Concepts
Practice allows you to build up muscle memory for common ways of solving problems!
Practice

Where to find practice problems

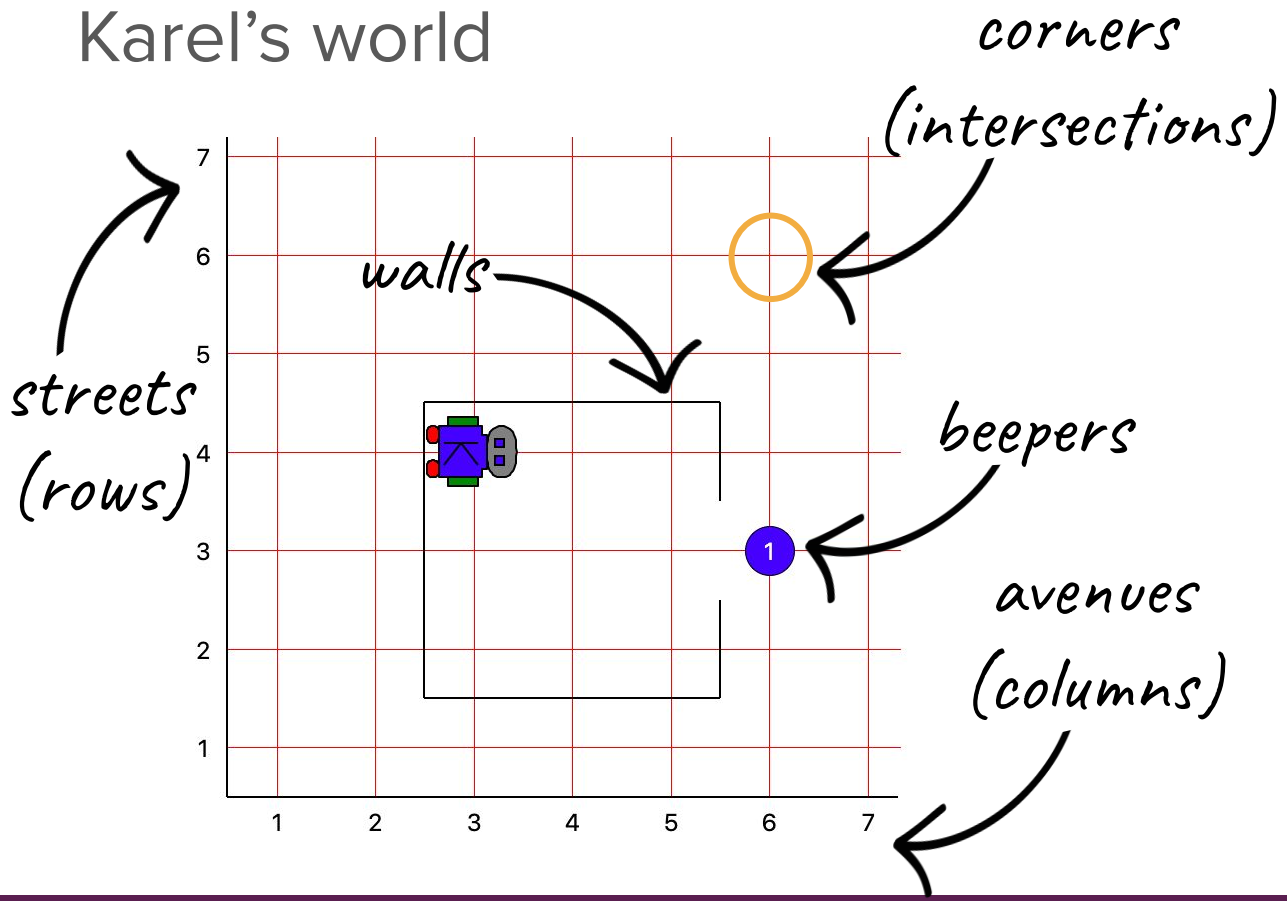
- Practice Midterm + Additional Practice Problems
- Section Handouts
- Scattered throughout these slides
- Lecture slides and homework

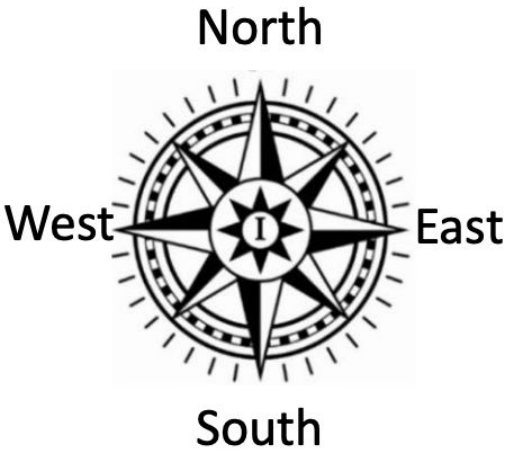
Today's Game Plan

- Karel
- Variables and Control Flow
- Functions
- Strings *
- Console Programs
- Images *
- Lists *
- Files
- Parsing *
- Dictionaries

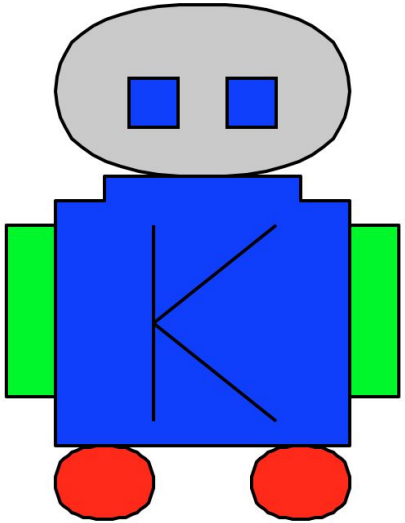
Karel

Karel's world





Karel's Commands



```
move()

turn_left()

put_beeper()

pick_beeper()
```

Questions Karel Can Ask

`front_is_clear()`	Is there no wall in front of Karel?
`left_is_clear()`	Is there no wall to Karel's left?
`right_is_clear()`	Is there no wall to Karel's right?
`on_beeper()`	Is there a beeper on the corner where Karel is standing?
`facing_north()`	Is Karel facing north?
`facing_south()`	Is Karel facing south?
`facing_east()`	Is Karel facing east?
`facing_west()`	Is Karel facing west?

Top-down decomposition

- “Divide-and-conquer”
  - Break the problem down into smaller parts
  - Ask: What are the steps that make up the larger problem? What tasks are repeated and might make good functions?

Top-down decomposition

- “Divide-and-conquer”

- Plan out your milestones (functions) first before writing them:
  - What are the pre-conditions and post-conditions for each function?
  - Which functions will call which?

Top-down decomposition

- “Divide-and-conquer”
- Plan out your milestones (functions) first before writing them
- Use the “blackbox model” for functions you’re not working on
  - When working on a particular function, assume that the others exist and already do what you want

Top-down decomposition

- “Divide-and-conquer”
- Plan out your milestones (functions) first before writing them
- Use the “blackbox model” for functions you’re not working on

Important note for Karel on the exam

- Just like in assignment 1, for a Karel problem you cannot use for loops, variables, continue, break, and other Python features that were taught after Karel. **You will only be allowed to use concepts from Lectures 1 to 4.**

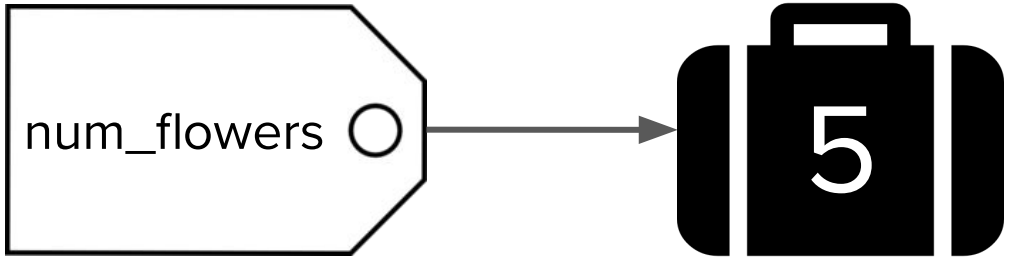
Variables and Control Flow

Variables: how we store information in our code

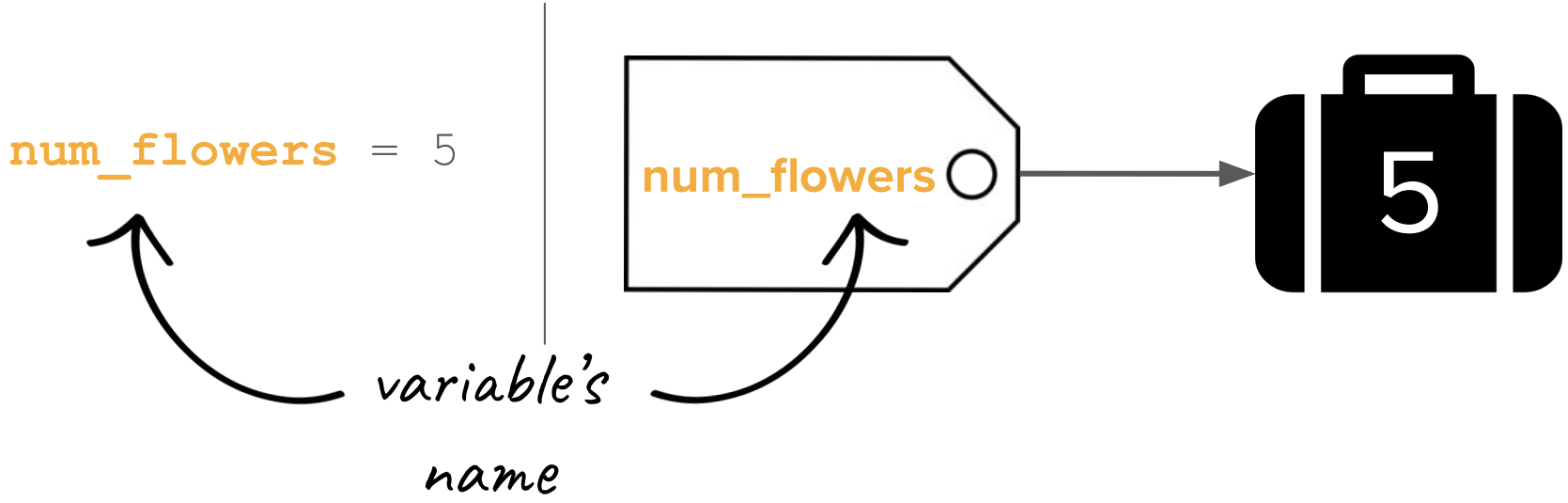
```
num_flowers = 5
```

Variables: how we store information in our code

```
num_flowers = 5
```

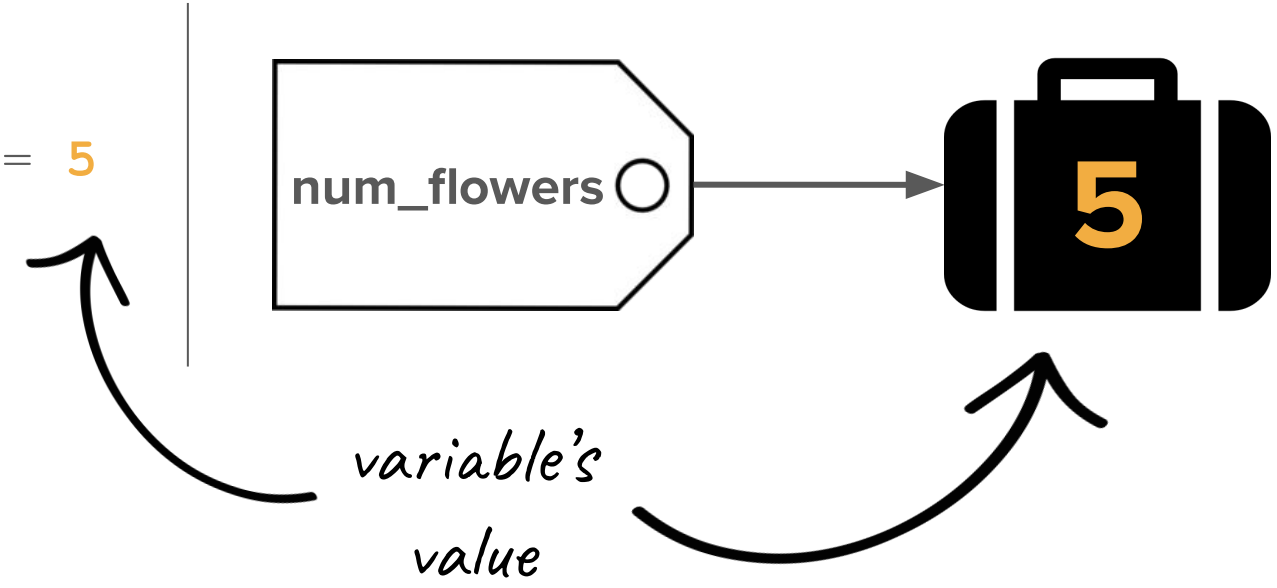


Variables: how we store information in our code



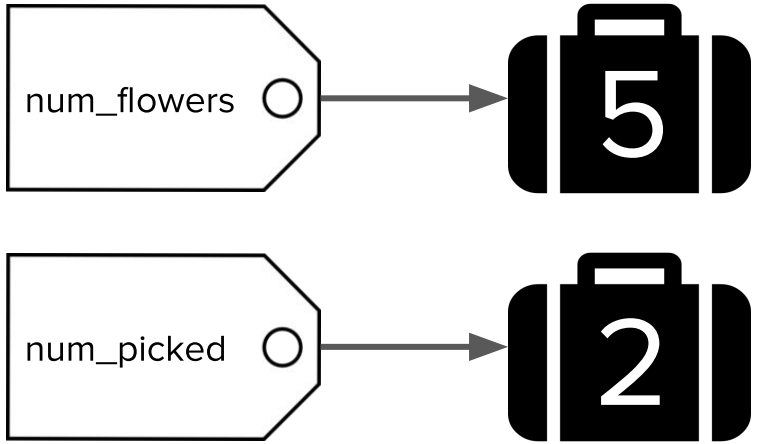
Variables: how we store information in our code

```
num_flowers = 5
```



Variables: how we store information in our code

```
num_flowers = 5
num_picked = 2
```

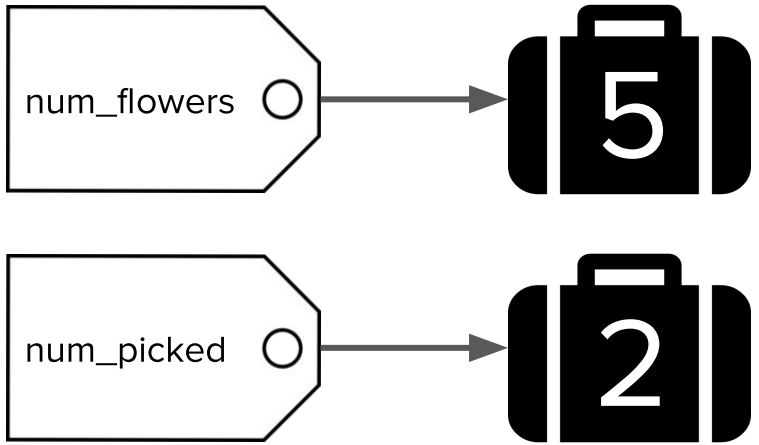


Variables: how we store information in our code

```
num_flowers = 5
num_picked = 2
num_flowers = num_flowers - num_picked
```

Variables: how we store information in our code

```
num_flowers = 5
num_picked = 2
num_flowers = num_flowers - num_picked
```

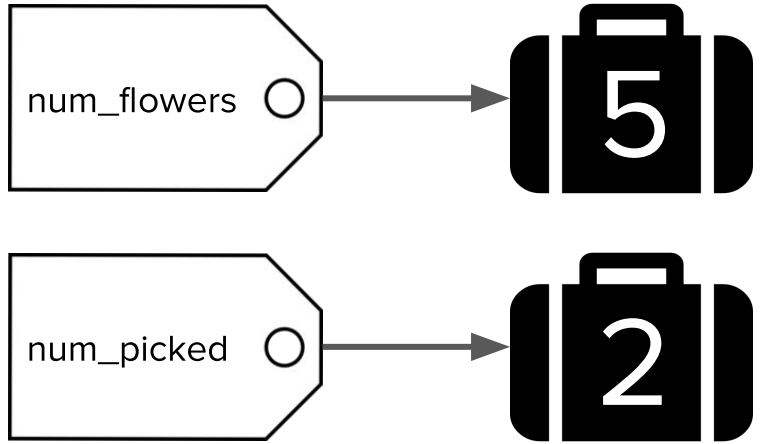


*The right side of the equals sign **always** gets evaluated first.*

Variables: how we store information in our code

```
num_flowers = 5
num_picked = 2
num_flowers = num_flowers - num_picked
```

5 2 3

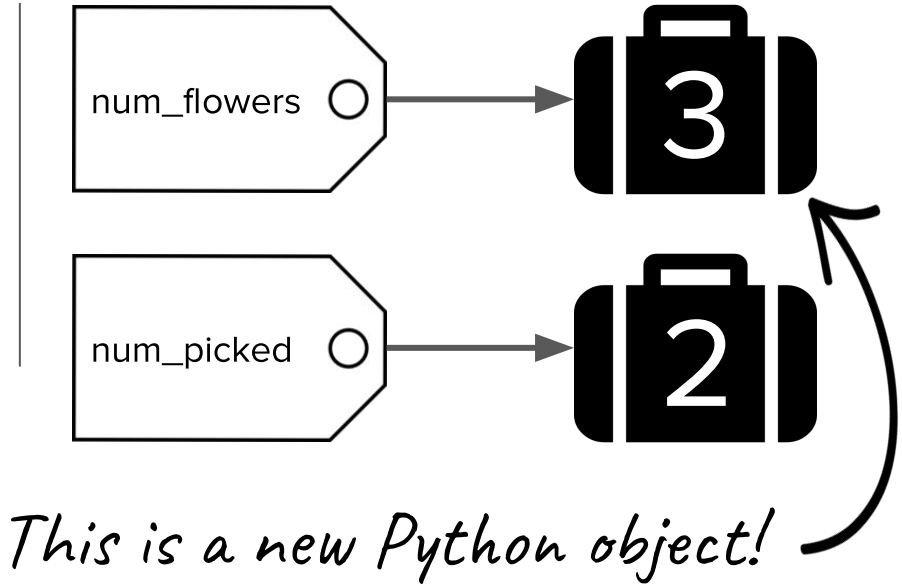


We get the values using variable retrieval (i.e. checking what suitcase is attached).

Variables: how we store information in our code

```
num_flowers = 5
num_picked = 2
num_flowers =
```

3



Variables summary

- Variables have a **name** and are associated with a **value**
- Variable **assignment** is the process of associating a value with the name (use the equals sign =)
- **Retrieval** is the process of getting the value associated with the name (use the variable's name)

All Python objects have a type!

- Python automatically figures out the type based on the value
  - Variables are “**dynamically-typed**”: you don’t specify the type of the Python object they point to

Expressions

- In Karel, we saw “boolean expressions” that evaluate to true/false
- More generally, expressions can evaluate to any type!
- The computer **evaluates** expressions to a single value.
- We use **operators** to combine literals and/or variables into **expressions**

Literals are Python objects written directly in code, e.g. the 5 in `num_flowers = 5`

Arithmetic operators

*	Multiplication
/	Division
//	Integer division
%	Modulus (remainder)
+	Addition
-	Subtraction

Operator	Precedence
`()`	1
`*, /, //, %`	2
`+, -`	3

This is your "order of operations" for Python!

Boolean comparison operators

<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Operator	Precedence
`()`	1
`*, /, //, %`	2
`+, -`	3
`<, <=, >, >=, ==, !=`	4

not, **and**, & **or**

- Python uses the words **not**, **and**, & **or** to combine boolean values
  - **not** `boolean_expression`
    Inverts True/False

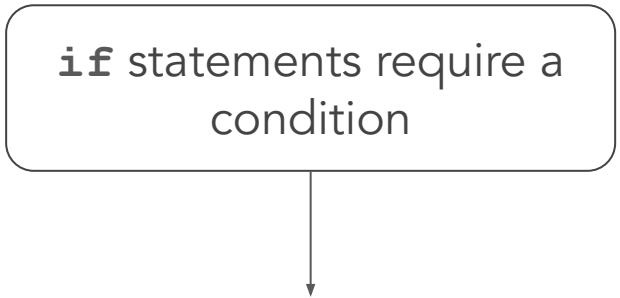
  - `boolean_expression_a` **and** `boolean_expression_b`
    If both sides are true, the entire condition is true.

  - `boolean_expression_a` **or** `boolean_expression_b`
    If either side is true, the entire condition is true.

Control Flow: the order in which our programs execute

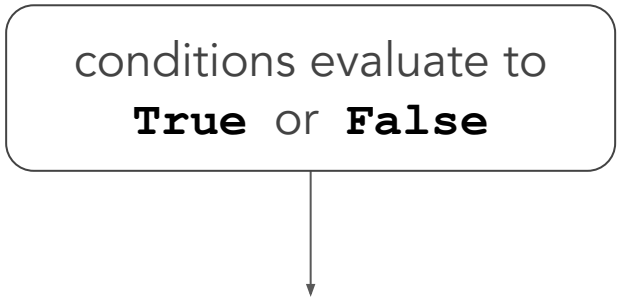
```
if front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute




```
if front_is_clear():
    # do some cool stuff
```

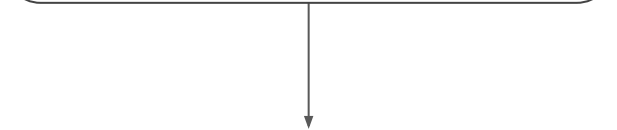
Control Flow: the order in which our programs execute




```
if front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

An if statement will only execute if the condition evaluates to **`True`**



```
if front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

If the condition is True, the code inside the if statement will happen exactly ***once***

```
if front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

Once the code inside the if statement has completed, the program moves on, *even if the condition is still* **`True`**

```
if front_is_clear():
    # do some cool stuff
# do some more cool things
```

Control Flow: the order in which our programs execute

```
while front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

`while` loops also require a condition, which behaves in exactly the same way

```
while front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

The difference is that the **`while`** loop repeats so long as the condition is **`True`**

```
while front_is_clear():
    # do some cool stuff
```

Control Flow: the order in which our programs execute

We only move on when the condition evaluates to **`False`**

```
while front_is_clear():
    # do some cool stuff
# do something different
```

Control Flow: the order in which our programs execute

But remember – you only re-evaluate the condition ***after all of the lines*** **inside the loop finish**

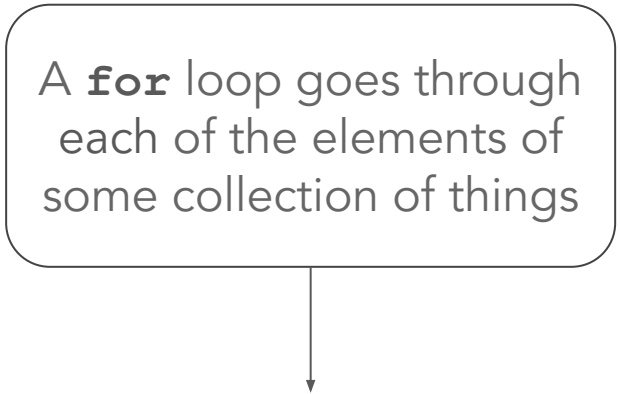
```
while front_is_clear():
    # do some cool stuff
# do something different
```

Control Flow: the order in which our programs execute

```
for i in range(42):
    # do some cool stuff
```

Control Flow: the order in which our programs execute

A **`for`** loop goes through each of the elements of some collection of things



```
for i in range(42):
    # do some cool stuff
```

Control Flow: the order in which our programs execute

The **range** function gives us an ordered collection of all the non-negative integers below a particular number

```
for i in range(42):
    # do some cool stuff
```

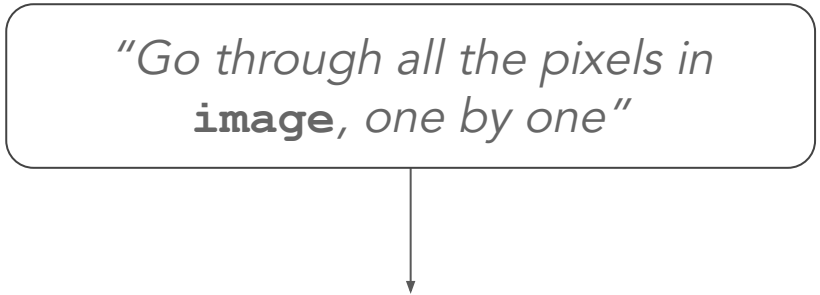
Control Flow: the order in which our programs execute

"Go through all the numbers until 42, one by one"

```
for i in range(42):
    # do some cool stuff
```

Control Flow: the order in which our programs execute

"Go through all the pixels in **`image`***, one by one"*



```
for pixel in image:
    # do some cool stuff
```

Control Flow: the order in which our programs execute

The `for` loop ends when we've gone through all the things in the collection

```
for pixel in image:
    # do some cool stuff
# more cool code here
```

For vs. while

```
for i in range(3):
    move()
```

Use for loops if you know how many times you want to repeat something!

```
while front_is_clear():
    move()
```

Use while loops if you don't know how many times you want to repeat!

Other useful things to know about control flow

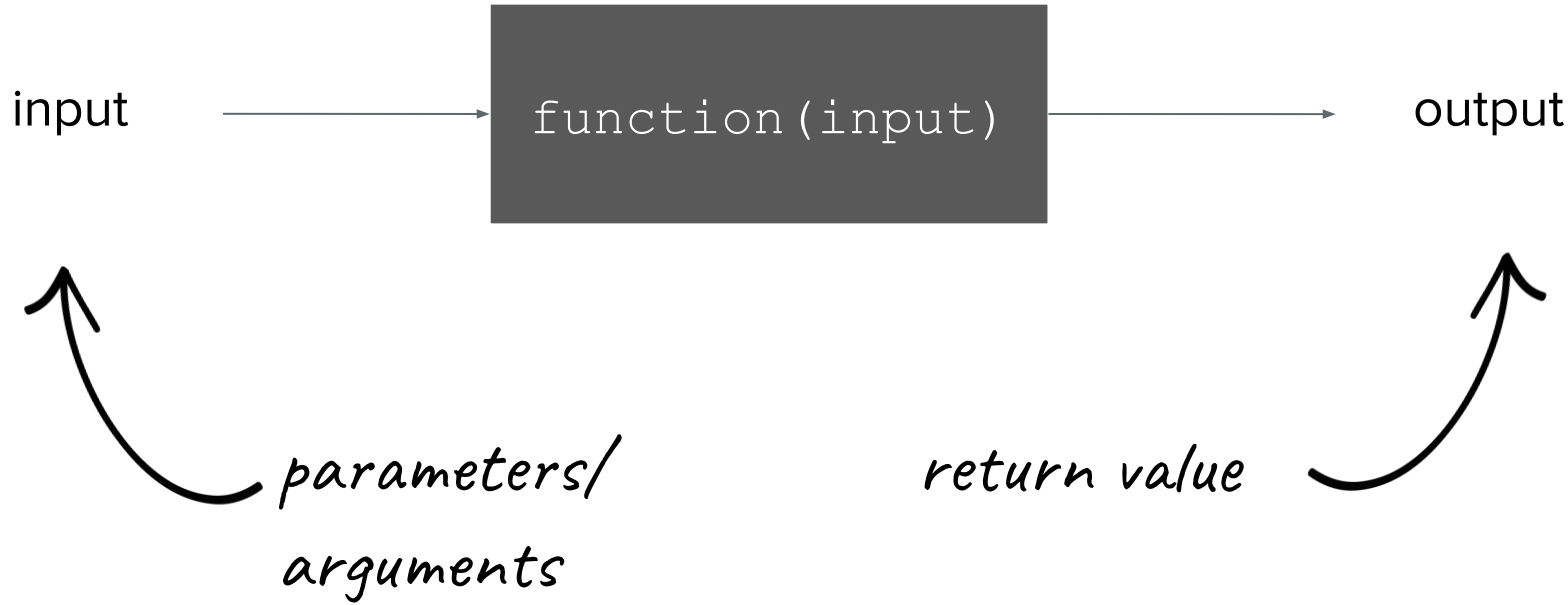
`range(10, 42)` - all the numbers between 10 (inclusive) and 42 (exclusive)

`range(10, 42, 2)` - all the numbers between 10 (inclusive) and 42 (exclusive), going up by 2 each time

Collections you can for-each loop over: strings, images, lists, files, and dictionaries

Functions

Anatomy of a function



Anatomy of a function

```
def main():
    mid = average(10.6, 7.2)
    print(mid)
```

caller

(calling function)

```
def average(a, b):
    sum = a + b
    return sum / 2
```

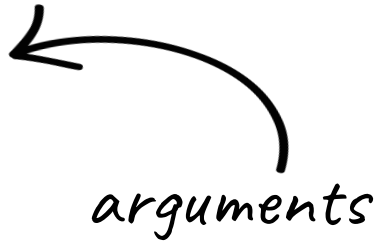
callee

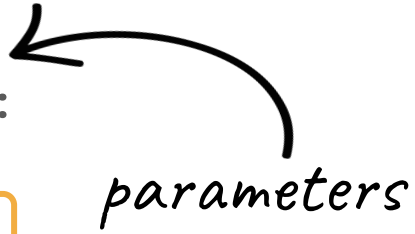
(called function)

Anatomy of a function

```
def main():
    mid = average(10.6, 7.2)
    print(mid)


def average(a, b):
    sum = a + b
    return sum / 2
```





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

```
def caller():
    x = 42
    callee(x)
    print(x)     # this still prints 42

def callee(x):
    x = x + 10   # we're only changing callee's
                 # version of x
```

Trace Programs

Very likely to show up on the midterm!

Lots of practice in section handouts and the practice midterm!

A great strategy for tracing programs: draw variables as baggage tags/suitcases, and go line-by-line through the program

Trace Programs

Very likely to show up on the midterm!

Lots of practice in section handouts and the practice midterm!

A great strategy for tracing programs: draw variables as baggage tags/suitcases, and go **line-by-line** through the program

Print vs. return

- If you want the calling function to have access to your function's output, you have to use **`return`**, not **`print()`**.
  - Make sure to store return values inside the calling function!

- Hitting a **`return`** line will cause you to immediately exit from a function.
  - This is not true with **`print()`**!

- Only use **`print()`** for displaying information to the user via the text output area.

Strings

String fundamentals

- String literals are any string of characters enclosed in single (‘’) or double quotes (“”)
- Each character in the string is associated with an **index**
- Strings can be combined with the + operator in a process called **concatenation**
- Strings are **immutable**

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() exists, returns true if all lower
'so long, and thanks for all the fish'  # returns a string
>>> s.upper()        # isupper() exists, returns true if all upper
'SO LONG, AND THANKS FOR ALL THE FISH'
```

Type conversion

- **Important note**: ‘123’ is a **`string`** and 123 is an **`int`**

- In order to convert between data types, we can use built-in Python functions: **`str()`**, **`int()`**, **`float()`**

Common string patterns

- Common pattern: processing all characters in a string

```
for i in range(len(s)):
    current_char = s[i]
    # Use current_char
```

- Common pattern: building up a new string

```
new_string = ''
for i in range(len(s)):
    if ___________:
        new_string += s[i]
```

Select only certain characters - think of this as a filter!

A problem: `remove_doubled_letters`

Implement the following function:

```
def remove_doubled_letters(s)
```

that takes in a String as a parameter and returns a new string with all doubled letters in the string replaced by a single letter. For example, calling

```
remove_doubled_letters('tresidder')
```

would return the string `'tresider'`. Similarly, if you call

```
remove_doubled_letters('bookkeeper')
```

the function would return the string `'bokeper'`.

Questions I’d ask myself

What do I do with each character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one.

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one.

Is there anything else I need to think about?

Questions I’d ask myself

What do I do with each character?

If it isn’t the same as the last character, I add it to the result string

How do I get the last character?

I go to the index before my current one.

Is there anything else I need to think about?

The character at index 0 doesn’t have a character before it but need to go into the string.

The solution

```
def removed_doubled_letters(s):
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]


    return result
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]
        if ch != s[i - 1]:
            result += ch
    return result
```

The solution

```
def removed_doubled_letters(s):
    result = ''
    for i in range(len(s)):
        ch = s[i]
        if i == 0 or ch != s[i - 1]:
            result += ch
    return result
```

Console Programs

The console and the `input()` function

- The console is just another name for the text-output area that we have already seen when using the `print()` function. In addition to displaying text, the console can also solicit text from a user.
- The `input()` function takes in a single parameter, which is a prompt to show to the user.
- The function will then wait while the user types in text into the interactive terminal (console).
- After the user submits their answer by hitting the “Enter/Return” key, the function returns the value that the user typed into the console.

Console program summary

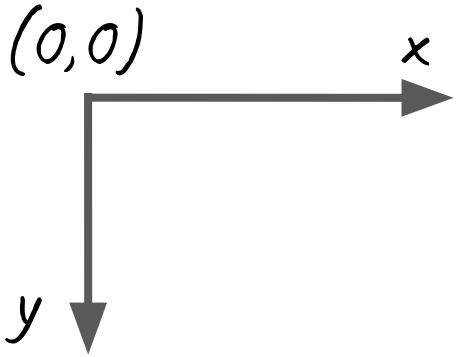
- Use **`input(prompt)`** to read in information from the user.
  - Make sure to convert the data to the correct type (from the **`string`** data type)!

- Use **`print()`** to display information for the user.
  - Make sure to convert the data to the correct type (from the **`int/float`** data type)

- Use a **`while`** loop to enable multiple runs of your program.

Images

What is an image?



- An image is made up of square **pixels**

- Each pixel has x and y **coordinates** depending on its location in the image

- Each pixel has a single color, encoded as three **RGB** numbers
  - R = red; G = green; B = blue
  - Each value represents a brightness for that color (red, green, or blue)
  - You can use these three colors to make ***any*** color!

What is an image?

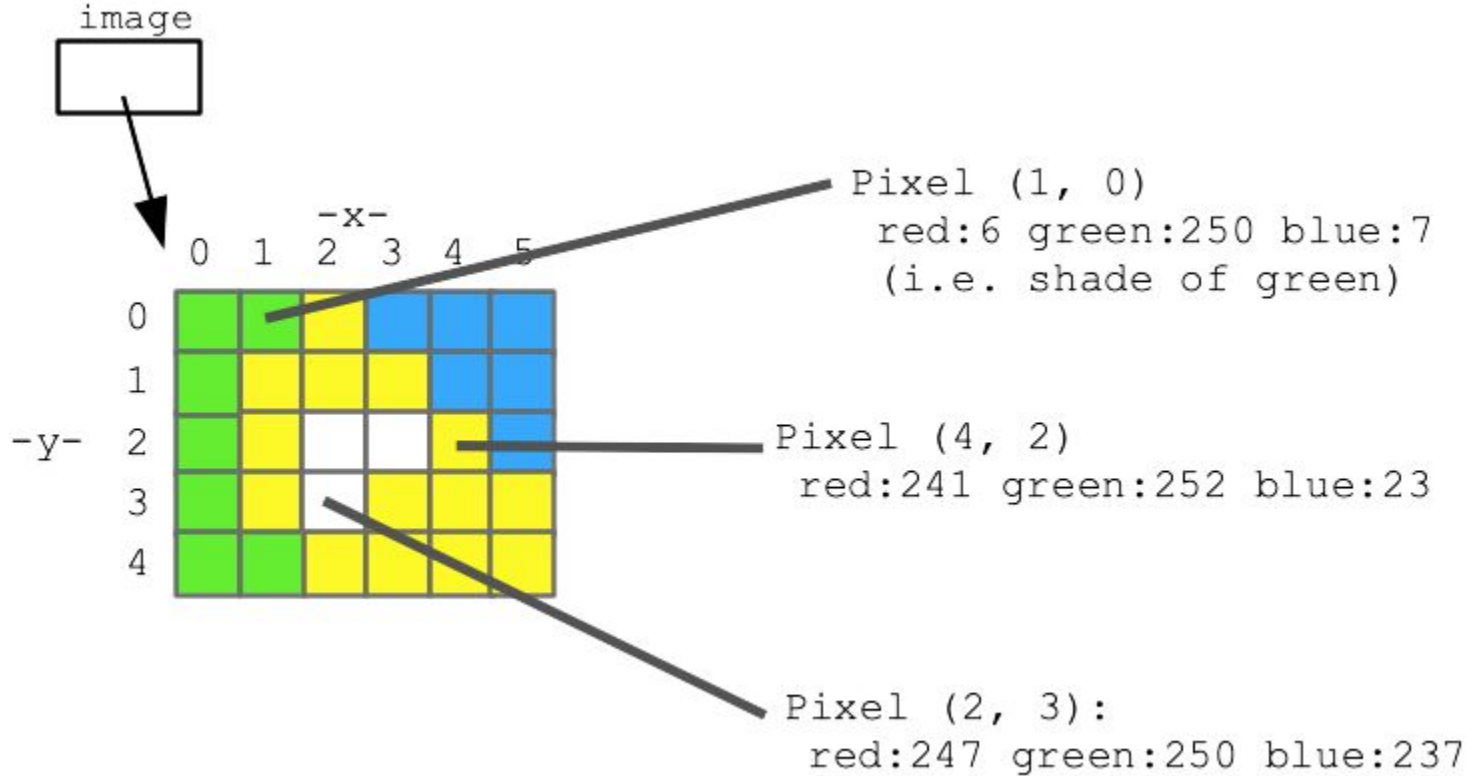


Image courtesy of Nick Parlante

SimpleImage

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

Image credits to Brahm Capoor

SimpleImage

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

height = img.height   # 10

width = image.width   # 6
```

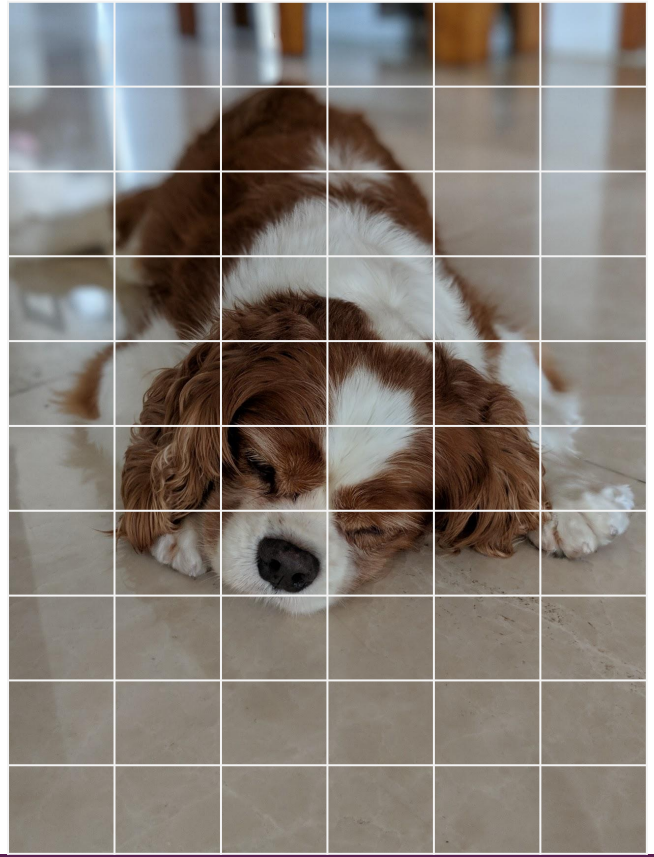


Image credits to Brahm Capoor

SimpleImage

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

height = img.height  # 10

width = image.width  # 6

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

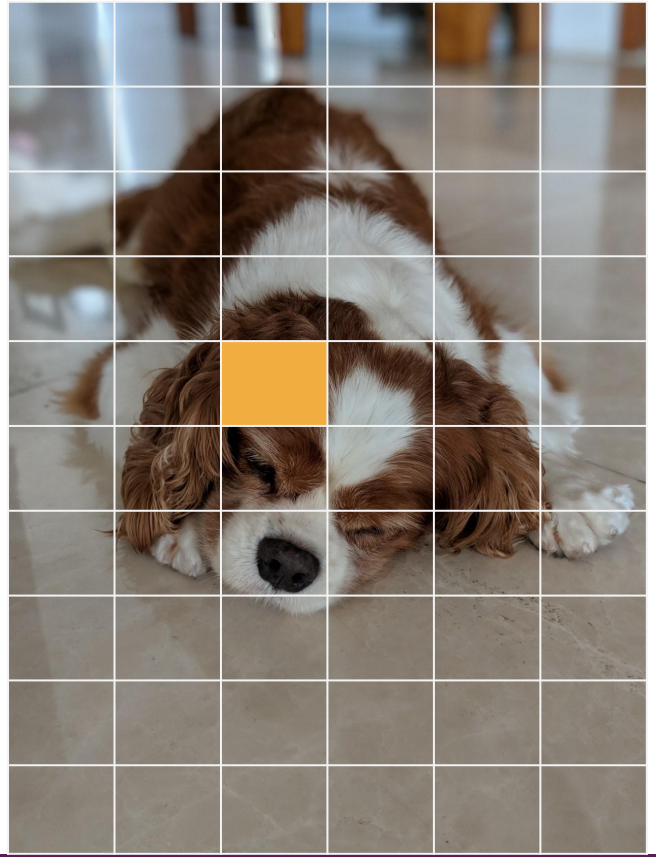


Image credits to Brahm Capoor

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 = pixel.red

green = pixel.green

blue = pixel.blue
```

Common Image Code Patterns

```
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
```

Summary

- Two different ways of accessing all pixels in an image

- Edit a pixel by updating its **properties:**
  - **`pixel.x`**, **`pixel.y`** → coordinates
  - **`pixel.red`**, **`pixel.green`**, **`pixel.blue`** → RGB values
    - A higher R, G, or B value means a greater amount of that color

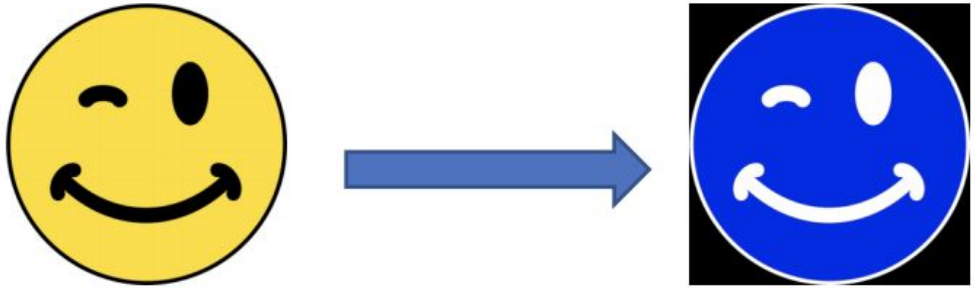
- Each SimpleImage also has properties:
  - **`image.width`** → maximum x value
  - **`image.height`** → maximum y value

A problem: make_negative

Implement the following function:

```
def make_negative(image)
```

that takes in a **`SimpleImage`** as a parameter and returns the *negative* of the image. The *negative* of an image is an image whose pixel's color components are set to their inverse. The inverse of a color component *c* is 255 - *c*.



Solution

Solving problems is about strategic procrastination: what's easy that we can do quickly?

```
def make_negative(image):
```

Solution

We definitely need to return the (modified) image, so let's do that first.

```
def make_negative(image):


    return image
```

Solution

Now, we definitely need to loop over the pixels of the image. Do we need their coordinates?

```
def make_negative(image):


    return image
```

Solution

...probably not! Let's just use a single for loop.

```
def make_negative(image):
    for pixel in image:



    return image
```

Solution

Now, we've simplified the problem for ourselves: what do we do with a single pixel in order to solve the problem?

```
def make_negative(image):
    for pixel in image:



    return image
```

Solution

Invert it, just like the problem suggests!

```
def make_negative(image):
    for pixel in image:
        pixel.red = 255 - pixel.red
        pixel.green = 255 - pixel.green
        pixel.blue = 255 - pixel.blue
    return image
```

Solution

```
def make_negative(image):
    for pixel in image:
        pixel.red = 255 - pixel.red
        pixel.green = 255 - pixel.green
        pixel.blue = 255 - pixel.blue
    return image
```

Another problem:

Implement the following function:

```
def transform(image)
```

which takes in a **`SimpleImage`** as a parameter and returns an upside-down version of the image whose red pixels (those whose red components are more than 2 times the average color component of that pixel) have been replaced by their grayscale equivalents.

Some useful hints

Whenever you're moving pixels around, it's usually easier to make a new image and copy pixels into it, like so:

```
out_image = SimpleImage.blank(image.width, image.height)
for out_y in range(out_image.height):
    for out_x in range(out_image.width):
        out_pixel = out_image.get_pixel(out_x, out_y)
        in_pixel =in_image.get_pixel(<calculate coordinates here>)
        out_pixel.red = # some value, possibly based on in_pixel
        out_pixel.green = # some value, possibly based on in_pixel
        out_pixel.blue = # some value, possibly based on in_pixel
```

Some useful hints

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

(Your diagram doesn't need to be neat, it just needs help you calculate coordinates)

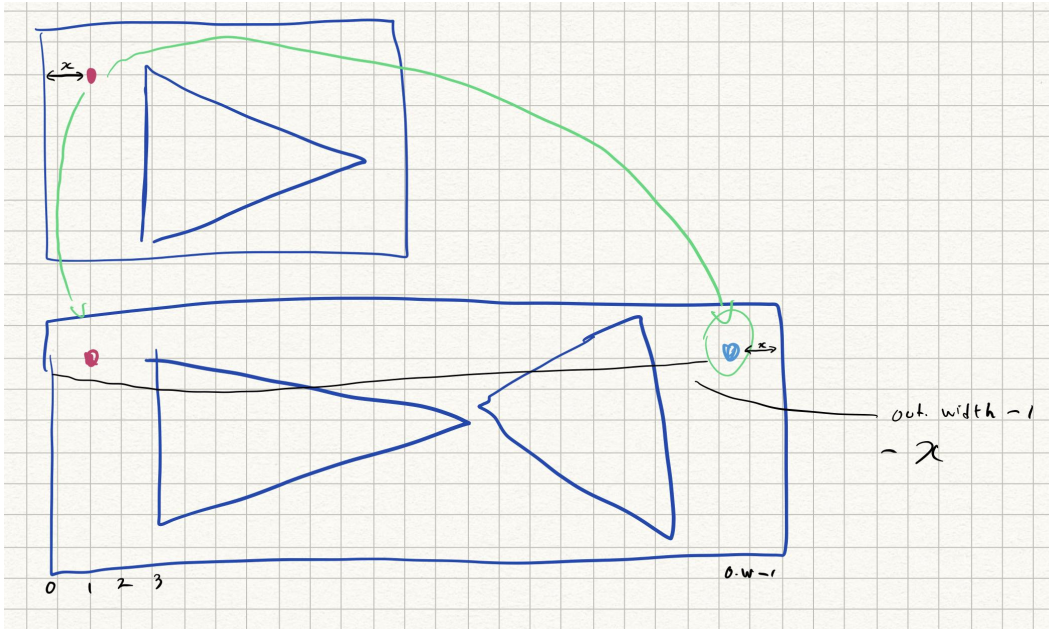
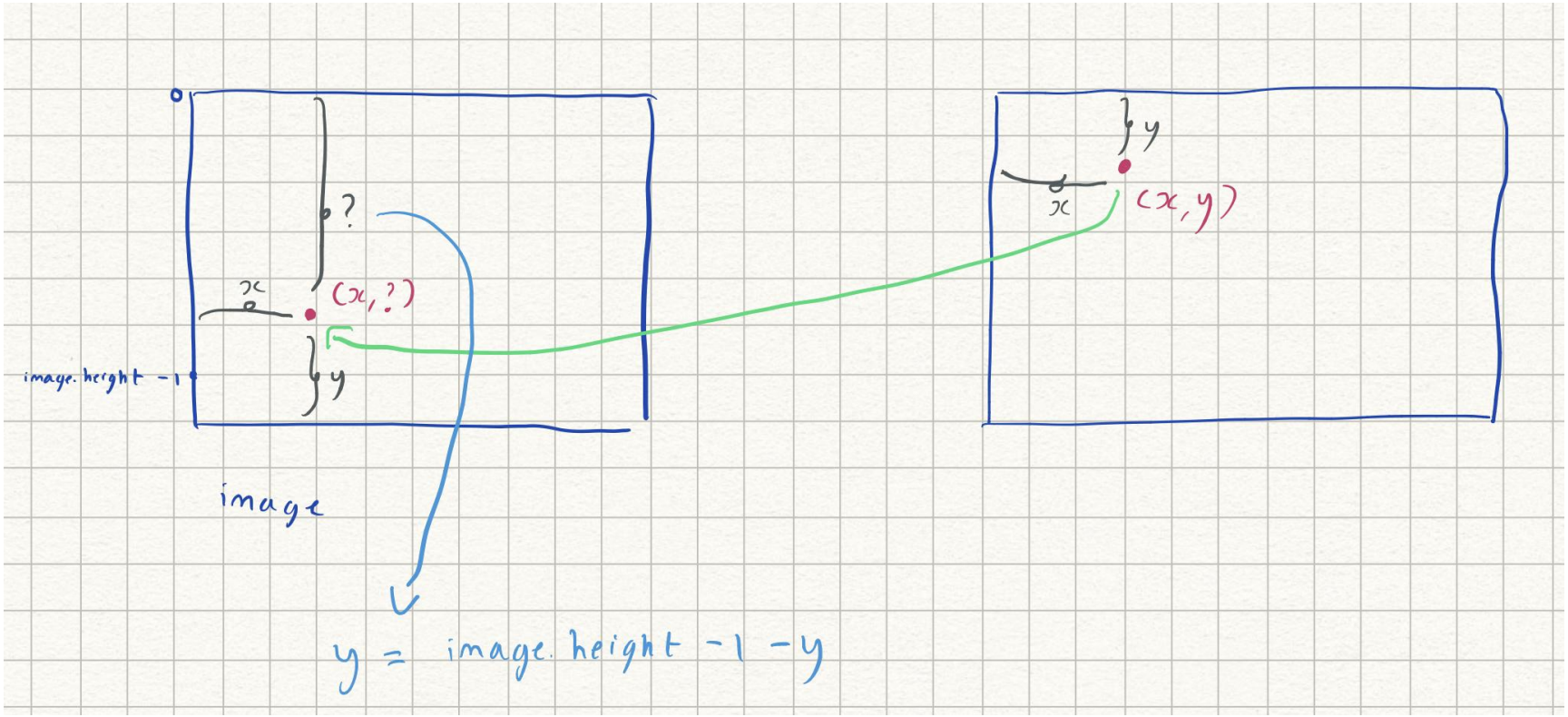


Image credits to Brahm Capoor

Solution

```
def transform(image):
    out_image = SimpleImage.blank(image.width, image.height)
    for out_y in range(out_image.height):
        for out_x in range(out_image.width):
            out_pixel = out_image.get_pixel(out_x, out_y)
            upside_down_y = image.height - 1 - out_y # see next slide
            in_pixel = image.get_pixel(out_x, upside_down_y)
            sum_color = in_pixel.red + in_pixel.green + in_pixel.blue
            average_color = sum_color // 3
            if in_pixel.red > 2 * average_color:
                out_pixel.red = average_color
                out_pixel.green = average_color
                out_pixel.blue = average_color
            else:
                out_pixel.red = in_pixel.red
                out_pixel.green = in_pixel.green
                out_pixel.blue = in_pixel.blue
    return out_image
```

Useful picture for this problem



Lists

List fundamentals

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

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

They work in many similar ways that strings do:

```
>>> 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 (equivalent to +=):

```
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. This is because unlike strings, lists are **mutable**.

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
```

A problem: `filter_starts_with_vowel`

Implement the following function

```
def filter_starts_with_vowel(s)
```

that takes in as a parameter a string **`s`** representing a sentence, and returns a list of all the words in the sentence that begin with a vowel. Your function should be case-insensitive (meaning that words that start with both capitalized and lower-case vowels should be included in the final output).

```
>>> filter_starts_with_vowel('I enjoy eating my favorite snack: Oreos')
```

```
['I', 'enjoy', 'eating', 'Oreos']
```

The solution

```
def filter_starts_with_vowel(s):

    vowel_start = []
    for word in words:



    return vowel_start
```

We'll start with a skeleton that looks mostly like our string problem structure: a result list, a **`for`** loop, and a **`return`**.

The solution

```
def filter_starts_with_vowel(s):
    words = s.split()
    vowel_start = []
    for word in words:



    return vowel_start
```

Now we're going to need to be able to access the words in our string. The .split() function is a handy way to do so!

The solution

```
def filter_starts_with_vowel(s):
    words = s.split()
    vowel_start = []
    for word in words:
        first_letter = word[0].lower()


    return vowel_start
```

Now that we are looping over all the words, we want to access each first character.

The solution

```
def filter_starts_with_vowel(s):
    words = s.split()
    vowel_start = []
    for word in words:
        first_letter = word[0].lower()
        if first_letter in 'aeiou':
            vowel_start.append(word)
    return vowel_start
```

Finally, once we find a word that satisfies our criteria, we can append it onto our growing list of acceptable words.

Files

Files in one slide

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

Files in ~~one~~ two slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

We open the file, specifying that we want to read through it, and give it a nickname of **f**

Files in ~~one~~ ~~two~~ three slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

f is a collection of lines of text, so we can use a for loop to go through each of those lines

Files in ~~one~~ ~~two~~ ~~three~~ four slides

```
def print_out_file(filename):
    with open(filename, 'r') as f:
        for line in f:
            print(line, end='')
```

Since the line in the file already has a **newline character** at the end, we shouldn't add another one when we print

```
Roses are red,\n
Violets are blue,\n
Poetry is hard,\n
So I give up.\n
```

Files in ~~one~~ ~~two~~ ~~three~~ ~~four~~ *don't worry about it* slides

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

This is a general pattern you can use whenever you need to read through a file

Files in ~~one~~ ~~two~~ ~~three~~ ~~four~~ *don't worry about it* slides

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

You can just pretty much stick this right in there without thinking! Just be careful to adjust the filename if needed.

Parsing

Components of Parsing

- File Reading
- String Manipulation
- Advanced Control Flow
- Container Data Types

Everything that we've seen so far put together into one task!

Helpful Functions for Parsing

- **`isalpha(), isdigit(), isspace()`**
- **`startswith(), endswith()`**
- **`strip(), split()`**
- **`find()`**
- **`in`** and concatenation

A problem: `extract_quote`

Implement the following function:

```
def extract_quote(s)
```

that takes in a String as a parameter and returns the text of the first quote in the string, or a blank string if there are no quotes. A quote is defined as a substring surrounded by double quotation marks (“...”).

You may assume that **`s`** only has quotation marks when it has quotes.

Key insight

Do you need to loop over the characters of the string?

We probably could, but it sort of feels like the s.find() *function is doing that for us*

Solution

```
def extract_quote(s):
    first_quote_index = s.find('"')
    if first_quote_index == -1:
        return ''

    second_quote_index = s.find('"', first_quote_index + 1)
    quote = s[first_quote_index + 1 : second_quote_index]

    return quote
```

Dictionaries

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['sonja'] = 42         # associate the key 'sonja' with value 42
>>> d['nick'] = 5           # associate the key 'nick' with value 5
>>> d['nick'] = 8           # change the value for 'nick' to be 8
                            # since keys need to be unique
>>> d['sonja']              # get the value associated with 'sonja'
42
>>>  'python' in d          # check whether a key is in the map
False
```

The dictionary ‘counting’ 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!

Words of Wisdom

*We are not trying to trick you on this exam. You **know** how to do all these problems.*

For many of you, this is your first programming exam. We know it's a weird thing to do, and we are doing our utmost to ensure that if you understand this material, you will do well.

Stay calm, focus on how to apply what you know to the problems, and keep making them easier for yourself, as we did in each of the problems today.

We are not trying to trick you on this exam. You ***know*** *how to do all these problems.*

For many of you, this is your first programming exam. We know it's a weird thing to do, and we are doing our utmost to ensure that if you understand this material, you will do well.

Stay calm, focus on how to apply what you know to the problems, and keep making them easier for yourself, as we did in each of the problems tonight.

Good Luck!