YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

CS106AP Midterm Exam

Summer 2019

Your exam number:

Instructions (PLEASE READ CAREFULLY)

This printed exam has been provided to you as space for scratch work and as a reference for you as you work through the problems on your laptop. **We will not grade or look at any of the work written here.** The Exam Reference Sheet is printed at the end of the exam.

However, since we have make-up exams scheduled on different days, you must turn in this paper midterm at the end of the exam period. **Do not throw it away or take it with you.** In BlueBook, the first problem asks you for the number printed above ("Your exam number") so that we can confirm everyone has returned their hard copy. It also asks you for an electronic signature agreeing that you abided by the Stanford Honor Code. **Any exam that does not provide a complete answer to this problem (exam number + signature) will automatically receive a 0.**

There are five total problems on the exam, and some have multiple parts. Some general tips for tackling problems:

- We recommend looking over the entire midterm before getting started so that you can budget your time well.
- Pay attention to which functions ask you to print a value versus return a value.
- Please type up your work and write down your ideas, even if you get stuck on a problem. We will do our best to give you partial credit, as long as we can understand what you wrote.
- You do not need to comment your functions (unless you find it helpful as you work). We will not be grading on style.

Good luck!

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

Welcome to Karel's Klinic and HosPytal (K&H)! You've recently been hired to write some programs for K&H, and luckily, the knowledge you've gained from CS106AP can help them solve some tricky medical problems and speed up some of their processes. Let's get to work!

1. Trace: The mystery bill

At the end of every month, K&H send customers their bills for all services a given patient received, either from the K&H clinic or from the K&H hospital. The three steps for calculating a patient's medical bill are as follows:

- Summing together the costs for the clinical services the patient received
- Summing together the costs for the hospital services the patient received
- Subtracting the percentage cost covered by the patient's insurance company

However, some patients have been complaining that their totals aren't coming out correctly! The billing program was written by K&H's last programmer, who unfortunately didn't take CS106AP (so they didn't really know much about good style) and has now moved on to pursue their true passion of being a zookeeper. Luckily, you have some experience with both tracing code and building receipt programs, so K&H have asked for your help in debugging the existing billing program.

In particular, K&H want you to answer the following questions:

1. We have a patient who received 6 clinical services and 2 hospital services and whose insurance will cover 50 percent of the total cost. Below is the function call relevant to this patient:

   ```
   zoo(6, 2, 0.50)
   ```

   What does the program output for this function call?

2. The correct output for the above function call (how much the patient should actually be paying) is 1300. What's going wrong in the program, and how can you fix it? (Please explain briefly in 1-2 sentences.)

HINT: The previous K&H programmer wrote one function for each of the steps described above, as well as a single function that calls the other three. To help you figure out which function is which, it's worth noting that hospital services cost $1000 each and that clinical services cost $100 each.

Using the input from question 1, start by tracing through the functions to figure out what each is doing. Once you've answered question 1 and have a better idea of what's going on, then try tackling question 2. **The full billing program is on the next page.**

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

```
A = 100
B = 1000

def pangolin(h):
    return A* h

def sloth(c):
    return B *c

def kakapo(x, p):
    x= x-x*p

def zoo(c,h,p):
    x=0
    x +=pangolin(c)
    x += sloth(h)
    kakapo(x,p)
    print(x)
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

2. Karel: PharmacistKarel

NOTE: Just like in Assignment 1, in this problem you cannot use for loops, variables, continue, break, parameters, return values, and other Python features that were taught after Karel.

Karel's been hired to help fill medication orders at K&H's pharmacy! They need you to program Karel so that it fills an arbitrary number of medication vials in any sized world. Figures 1 and 2 below shows one possible world of medication vials at the start and finish of your program.

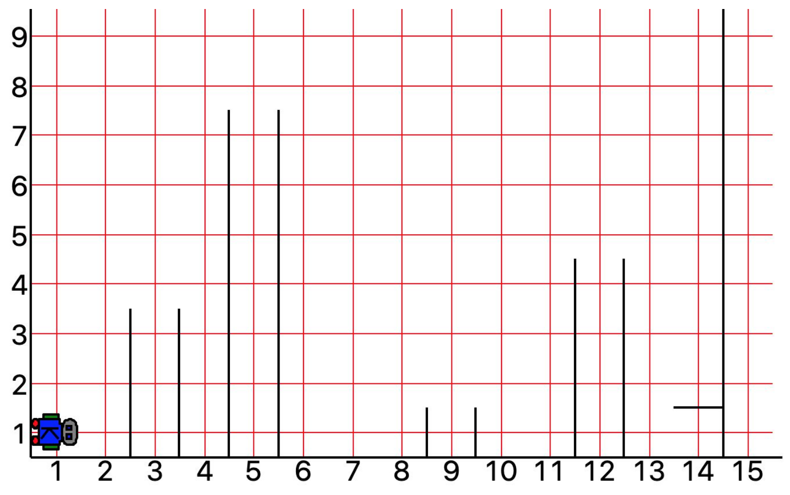


Figure 1: A possible world with medicine vials for Karel to fill.

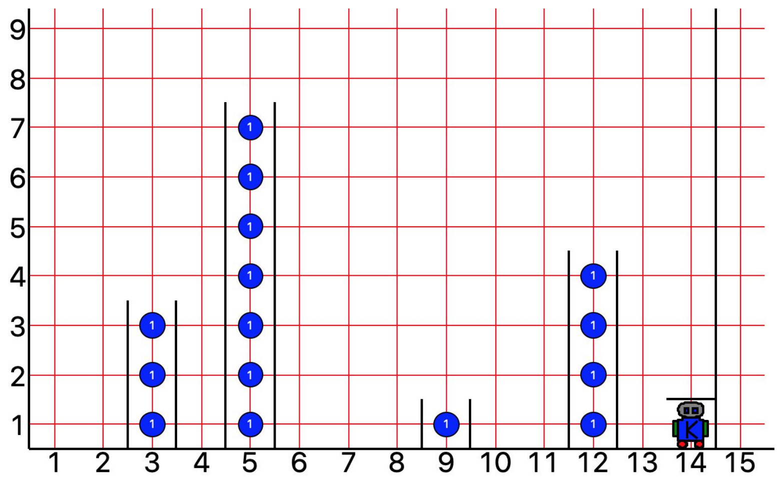


Figure 2: The end state for the world in Figure 1, once Karel has correctly filled all medicine vials.

You've been given the following constraints:

- Karel starts at the southwest corner of the world, facing east.
- Karel should end at the southeast corner of the world, facing north.

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

- Medication vials are defined as two consecutive, equally tall walls. The vials are always 1-avenue-wide.
- Medication vials will never be up against a side wall. In addition, there will always be at least two avenues between the last vile and the eastmost wall (i.e. there will always be at least one avenue between the west wall and the first vial and at least two avenues between the east wall and the last vial).
- There will never be medication vials in two consecutive avenues (i.e. medication vials will never share a wall).
- The final avenue against the eastmost wall will always have a horizontal wall above the first corner (as pictured in Figures 1 and 2).
- Karel must fill every medication vial with beepers, and vials cannot be overfilled (see Figure 2 for correctly filled vials).
- Worlds can be any height or width, and medication vials can also be any height.

The final program will call your **`main()`** function. In addition to completed **`turn_right()`** and **`turn_around()`** functions, we've provided two empty suggested functions (**`fill_vial()`** and **`move_until_blocked()`**) that you can fill in and then call in **`main()`** to help you break down the problem. While we recommend that you implement the two provided functions and further decompose the program, you're free to use a completely different decomposition strategy if an alternate method makes more sense to you. Remember strategies for top-down decomposition and pre- and post-conditions if you get stuck!

NOTE: Just like in Assignment 1, in this problem you cannot use for loops, variables, continue, break, parameters, return values, and other Python features that were taught after Karel.

```
def turn_right():
    """
    Rotates Karel 90 degrees clockwise
    """
    turn_left()
    turn_left()
    turn_left()

def turn_around():
    """
    Rotates Karel 180 degrees clockwise
    """
    turn_left()
    turn_left()

def fill_vial():
    """
    Fills a container with medicine (beepers)
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

```
    """
    # YOUR CODE HERE
    pass

def move_until_blocked():
    """
    Moves until blocked by a wall
    """
    # YOUR CODE HERE
    pass

def main():
    """
    Fills all containers and finishes in the bottom right corner of
    the world
    """
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

3. Images: Detecting cancerous growths

a) K&H's radiologist has asked for your help identifying cancerous growths in the images taken by her equipment. In particular, very dark pixels are a sign of an unhealthy area, but the known brightness threshold is difficult to distinguish with just the human eye. Your job is to write a function to help the radiologist better identify potentially cancerous growths by highlighting them green.

Write a function that takes in an image filename, loops over the image's pixels, and highlights potentially cancerous pixels green. A pixel is considered potentially cancerous if the average of its RGB values is below a provided **`CANCEROUS_THRESHOLD`**. If a pixel is not potentially cancerous, you should grayscale it instead.

Your function should return the modified image.

```
# Provided constant
CANCEROUS_THRESHOLD = 50

def highlight_cancerous_growths(filename):
    """
    filename: the name of an image file to be processed (string)
    """
    pass
```

b) Your program worked great, and the radiologist has used it to produce tons of grayscale images with green highlighting on the potentially cancerous areas! But it turns out that although K&H's medical imaging equipment is some of the best in the area, it still isn't perfect. Some of the images contain false positives and false negatives – in other words, because of small errors made by the imaging equipment, sometimes non-cancerous pixels get highlighted green, and other times cancerous pixels aren't properly detected (highlighted green).

Luckily, to help account for these errors, the equipment takes many pictures of a given area. So in order to get a fully "correct" highlighted image, we can look at all of the highlighted images taken of a specific area and consider only pixels that are identified as cancerous by the majority of the images. In other words, we can create a "best image" in which we only highlight green the pixels that the majority of images also highlight green.

Your job is to write a function **`improve_detection_accuracy()`** that takes in two parameters:

- `best_image`: a SimpleImage object that will be modified to create our "best image"
- `images`: a list of images (SimpleImage objects) that have already been processed (i.e. their "potentially cancerous" pixels have already been highlighted green). All of the photos in `images` are of the same unhealthy area, but each may contain some false positive pixels and/or false negative pixels.

We've given you code to loop over `best_image` and return it at the end of your function, but you need to fill in the code that will modify its pixels based on the photos in `images` that have already been highlighted! If the majority of images in `images` highlight a given pixel green, then you should also color that pixel green in `best_image` (R = 0, G= 255, B = 0). Otherwise, you should leave the pixel untouched.

Note that the `images` list passed into your function is a list of SimpleImage objects. **You do not need to create new SimpleImage objects in your code**; you can just call SimpleImage functions directly on the items in the `images` list and on `best_image`.

```
def improve_detection_accuracy(best_image, images):
    """
    best_image: an unprocessed SimpleImage object of the potentially
    affected area
    images: a list of SimpleImage objects
    NOTE: You do not need to create any new SimpleImage objects
    in this problem. The items in the images list are SimpleImage
    objects, not filenames.
    """
    for y in range(best_image.height):
        for x in range(best_image.width):
            pixel = best_image.get_pixel(x,y)
            # YOUR CODE HERE

    return best_image
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

4. Console program: Online appointments

K&H want to expedite their appointment scheduling process by allowing patients to schedule appointments online. In this problem, you'll be writing a console program to help them do the job! We'll break the process down into three steps (functions):

a) A parsing function for converting string times to integers
b) A file-reading function for checking what times are already taken by existing appointments
c) A console-program function for scheduling a user's appointment at the earliest available time

Each part is further described in the corresponding sections below. **Note that at each step, you can assume the functions you wrote in the previous parts of the problem work completely and are bug-free.** For example, if you have a bug in part a's `convert_to_24_hour_time()` and aren't sure how to solve it, you can still call `convert_to_24_hour_time()` in parts b and c as if it works and without penalty.

a) Write a function `convert_to_24_hour_time()` that takes in a string representing a time with 'AM' or 'PM' and converts it to a 24-hour-clock time. Note that AM corresponds to morning hours (from midnight to 11 in the morning) and PM corresponds to afternoon and evening hours (from noon to 11 in the evening).

Your function will take in a parameter `time` that has the format '[X]PM' or '[X]AM' where [X] is some number between 1 and 12 (there won't be any minutes in the times). Your function should return the converted time as an integer between 0 and 23. Below are all possible time strings and the corresponding integers for each time.

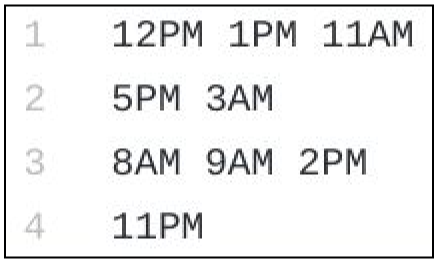
- '12AM' → 0
- '1AM' → 1
- '2AM' → 2
- '3AM' → 3
- '4AM' → 4
- '5AM' → 5
- '6AM' → 6
- '7AM' → 7
- '8AM' → 8
- '9AM' → 9
- '10AM' → 10
- '11AM' → 11

- '12PM' → 12
- '1PM' → 13
- '2PM' → 14
- '3PM' → 15
- '4PM' → 16
- '5PM' → 17
- '6PM' → 18
- '7PM' → 19
- '8PM' → 20
- '9PM' → 21
- '10PM' → 22
- '11PM' → 23

```
def convert_to_24_hour_time(time):
    """
    Takes in a string representing a time in the form:
        '[X]AM' or '[X]PM'
    where [X] is a number between 1-12

    Returns an int between 0-23 corresponding to the 24-hour time.
    """
    # YOUR CODE HERE
    pass
```

b) Write a function that takes in the name of a file containing times at which appointments have already been scheduled. In particular, you want to read the times from the file into a list (which will be used in part c to help determine when users can book an appointment, since they can't book a time that has already been scheduled). Each line of the file contains an arbitrary number of times, and times on a given line are separated by spaces. Figure 3 shows a sample input file that indicates that appointments have already been scheduled for 12PM, 1PM, 11AM, 5PM, 3AM, 8AM, 9AM, 2PM, and 11PM.



```
1    12PM 1PM 11AM
2    5PM 3AM
3    8AM 9AM 2PM
4    11PM
```

Figure 3: A sample input text file of already scheduled appointment times. The gray numbers in the left column represent line numbers and are not part of the file's text itself (e.g. line 1 is '12PM 1PM 11AM').

The file shown in Figure 3 would return the following list of integers (the order of the integers does not matter):

```
[12, 13, 11, 17, 3, 8, 9, 14, 23]
```

For this function, you can assume that your `convert_to_24_hour_time()` function works and can be called from within `get_appointments()`.

```
def get_appointments(filename):
    """
    Takes in a filename for the file containing already scheduled
    appointments.
    Returns a list of integers representing the times of the
    existing appointments.
```

```
"""
# YOUR CODE HERE
pass
```

c) Write a console program (function) that determines the best possible time for a user's appointment, based on their availability and on already scheduled appointments. A user's appointment cannot be scheduled during one of the already scheduled appointments. (Note that one scheduled appointment takes up exactly one timeslot – i.e. if someone has already booked 2PM, someone else could book 3PM as long as that time is not also listed as taken.)

We've already written code for you that calls your function from part b in order to get a list of already scheduled appointments. Your job is to write code that prompts the user for times when they are available until they enter the sentinel value "DONE." After the user inputs "DONE," you should print the **earliest available time** that they can have their appointment. The time you print should be converted to a 24-hour clock.

For example, if the user's appointment is scheduled for 2PM, you should print "Your appointment is scheduled for 14 o'clock." Figure 4 shows a sample run for your program (assuming the same input file as Figure 3 and with the user input in bold). Your prompts do not have to match ours, but you may find it easiest to use them directly so you don't need to come up with your own.

```
Please input a time when you're available or DONE when finished: 2PM
Please input a time when you're available or DONE when finished: 10PM
Please input a time when you're available or DONE when finished: 1PM
Please input a time when you're available or DONE when finished: 10AM
Please input a time when you're available or DONE when finished: 8AM
Please input a time when you're available or DONE when finished: 9AM
Please input a time when you're available or DONE when finished: 7PM
Please input a time when you're available or DONE when finished: DONE
Your appointment is scheduled for 10 o'clock
```

Figure 4: A sample run for your console program, given the input text file from Figure 3.

You can assume the users will always provide valid times. In other words, the user behaves in the following way:

- They will only input times on the hour (e.g. 1PM, 2PM, 8AM, etc.)
- They will provide correctly formatted time strings: "[X]AM" or "[X]PM" where [X] is a number between 1-12.
- They will always capitalize "AM" and "PM."
- They will not add any extraneous spaces within or around the string.
- They will input at least one time that does not conflict with an existing appointment.

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

Note, however, that the user is not required to input times in chronological order.

For this part of the problem, you can assume that your **`get_appointments()`** function and **`convert_to_24_hour_time()`** function work completely and can be called from within **`schedule_appointment()`**.

```
def schedule_appointment(filename):
    """
    Takes in a filename for the file containing already scheduled
    appointments (we call your get_appointments() function for you).
    Prompts the user for their available times until they input
    'DONE' and then prints the earliest possible time that the
    user can have their appointment
    """
    existing_appointments = get_appointments(filename)
    # YOUR CODE HERE
    pass
```

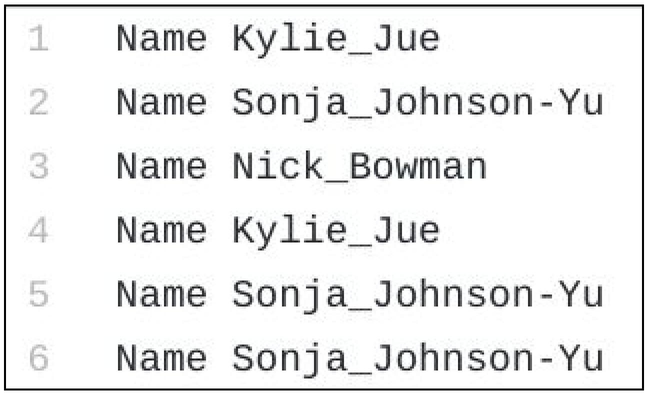
YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

5. Dictionaries: Patient visit count

At the end of the year, insurance companies always ask K&H how many times each patient visited the clinic. In particular, they want a well-structured dictionary of patient names to the number of times a given patient visited K&H. Your job is to write a program that takes in a text file that contains lines with patient names. An example input file is shown in Figure 5.

You can assume the following about the file:

- In a given file, each time that a patient's name appears represents one visit to K&H.
- Each line will have only one patient's name on it.
- Each line begins with "Name," followed by the patient's name (with the first and last name connected by an underscore).
- There are no excess leading or trailing spaces on any of the lines.



```
Name Kylie_Jue
Name Sonja_Johnson-Yu
Name Nick_Bowman
Name Kylie_Jue
Name Sonja_Johnson-Yu
Name Sonja_Johnson-Yu
```

Figure 5: A sample inputted text file of patients who have visited.

The resulting dictionary from the text file pictured in Figure 5 would be:

```
{'Kylie_Jue': 2, 'Sonja_Johnson-Yu': 3, 'Nick_Bowman': 1}
```

Write a function that processes the text file indicated by **`filename`** and returns a dictionary of patient names to how many times they visited.

```
def get_patient_visits_dict(filename):
    # YOUR CODE HERE
    pass
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

Exam Reference Sheet

Note: THIS DOCUMENT WILL BE PROVIDED TO YOU AT THE TIME OF THE MIDTERM. You do not need to print it out yourself, and it **will not count** towards your 10-pages of notes.

General Python

Built-in functions

`len()`	Returns the length of a collection (e.g. string or list).
`int()/float()/str()`	Converts a Python object to an integer/float/string type.
`range()`	Generates a range of integer values.
`input()`	Presents a string prompt to the user and returns the string that they input.
`print()`	Display output to the text output area (console).

Keywords used in boolean expressions

`not`	Negates (flips) the boolean value that follows.
`in`	Indicates if an element is part of a collection of objects.
`and`	Returns True if both values are True, False otherwise.
`or`	Returns False if both values are False, True otherwise.

Strings

Remember that string functions are called using the `noun.verb()` convention. For example, `str.isalpha()` where `str` is the string literal or variable storing the string.

Functions that return booleans

`isalpha()/isdigit()`	Returns a boolean indicating if a string is composed of all letters/digits.
`isupper()/islower()`	Returns a boolean indicating if a string is composed of all uppercase/lowercase characters.

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

Functions that return strings

`upper()/lower()`	Returns a string with all characters uppercase/lowercase.
`strip()`	Returns a string with leading and trailing whitespace removed.

Functions that return ints

`find()`	Returns the index of the first occurrence of a specified character in the string.

Functions that return lists

`split()`	Returns a list containing all substrings separated by the indicated delimiter.

Lists

To create an empty list, you use square brackets `[]`.

Remember that list functions are called using the **`noun.verb()`** convention. For example, **`lst.append()`** where **`lst`** is the variable storing the list.

`append()`	Add an element to the end of the list.
`extend()`	Add all elements in the specified list to the end of the target list.
`pop()`	Remove an element from the list and return it.
`insert()`	Insert an element into the specified index of a list.

Slicing

Recall that you can use slicing on both strings and lists (collections). You can get a particular slice of a collection using **`collection[start_index:end_index:step]`**, where the slice starts at **`start_index`** and stops right before **`end_index`** (not inclusive). The step is optional and defaults to a value of 1.

Dictionaries

- To create an empty dictionary, you use curly brackets `{}`.
- You can put or re-assign a key-value pair in your dictionary **`d`** using **`d[key] = value`**.
- To check if a particular key exists inside your dictionary **`d`**, you can use **`key in d`**.

Remember that list functions are called using the **`noun.verb()`** convention. For example, **`d.append()`** where **`dict`** is the variable storing the dict.

`keys()`	Returns an iterable over all of the keys in the dictionary.

`values()`	Returns an iterable over all of the values in the dictionary.
`items()`	Returns an iterable over the key, value pairs in the dictionary.

Recall that you can use iterables in for-each loops. To convert them to lists, you use `list(iterable)`. For example, you would need to use `list(d.keys())` to create a list of all the keys in a dictionary `d`.

Images

SimpleImage Code Patterns

```
# create image from filename
image = SimpleImage(filename)
# create blank image
image = SimpleImage.blank(width, height)

# foreach loop
for pixel in image:
    # do something with pixel

# range/y/x loop
for y in range(image.height):
    for x in range(image.width):
        pixel = image.get_pixel(x, y)

# pixel attributes
pixel.red, pixel.green, pixel.blue
pixel.x, pixel.y
```

File reading

Reading lines from a file

```
with open(filename, 'r') as f:
    for line in f:
        # do something with line
```

YOU MUST RETURN THIS HARD COPY OF THE MIDTERM AT THE END OF THE EXAM.

Karel

Basic Karel Commands

`move()`	Karel moves forward one square in the direction it is facing.
`turn_left()`	Karel rotates 90 degrees to the left (counterclockwise).
`put_beeper()`	Karel places a beeper on the corner where it is currently standing.
`pick_beeper()`	Karel picks up a beeper from the corner where it is currently standing.

Karel Conditional Functions

These Karel functions return `True` if the answer to the corresponding question is "yes" and `False` if the answer is "no."

`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?