Vul hier je naam en studentnummer in vóór je begint, en leg je collegekaart klaar:	

What to do

In the questions below, we ask you to implement the `TODO`s that are in the code. Note that you should not use pseudocode, but **real Python**. In particular, we need to see (from the code that you write) that you understand how to:

- ☐ retrieve items from a dictionary
- ☐ store or retrieve items within an instance using `self`
- ☐ write a short loop to process information
- ☐ store and update information in a class from within another class
- ☐ print a summary of all the data in an object
- ☐ use return correctly

It is no problem if you forget a colon (`:`) somewhere, etc., as long as your code is clear and not ambiguous.

You should do all of the problems on the next pages! Even if you are certain you have met the requirements, a simple mistake may make it harder to be certain that you understand what you're doing.

Hands

Let's implement a small part of the student "hands" queue, as it is called by teachers and assistants. We'll create a class `Question` to represent one student's question, as well as a class `Queue` for the queue itself. This system helps teachers and assistants to find the next student who has a question, while maintaining order: students who have been in line longest should be helped first.

Let's start out with the `Question` class.

```
TRACK_NAMES = { 1: "Parttime", 2: "Fulltime" }

class Question:
    def __init__(self, question, student_name, track):
        # TODO
    def description(self):
        # TODO
```

As you can see a small dictionary is available which lists types of students. Here's a brief example of instantiating a question and printing its description:

```
>>> q1 = Question("Hulp met tentamenvragen maken", "David Malan", 1)
>>> print(q1.description())
David Malan (Parttime) asked: Hulp met tentamenvragen maken
```

Then there is the `Queue` class, which can *contain* a number of questions and has a few methods to manage the queue. Note that each instance has a list of `Questions`. We have added some testing code to show how the class should work.

```
class Queue:

    def __init__(self):
        self.questions = []
    def size(self):
        return len(self.questions)
    def add(self, question):
        # TODO
    def is_empty(self):
        # TODO
    def peek(self):
        # TODO
    def claim(self):
        return self.questions.pop(0) # removes and returns next question from front of queue
    def find(self, track):
        # TODO
    def print_list_in_order(self):
        # TODO

queue = Queue()
q1 = Question("Hulp met tentamenvragen maken", "David Malan", 1)
queue.add(q1)
q2 = Question("danjkwnnqojnad", "Colton Ogden", 2)
queue.add(q2)
queue.print_list_in_order()
print(queue.peek().description())   # David Malan (Parttime) asked: Hulp met tentamenvragen maken
print(queue.claim().description())  # David Malan (Parttime) asked: Hulp met tentamenvragen maken
q3 = Question("Where's my video crew?", "Zamyla Chan", 1)
queue.add(q3)
print(queue.claim().description())  # Colton Ogden (Fulltime) asked: danjkwnnqojnad
queue.print_list_in_order()         # prints multiple lines of descriptions
print(queue.size())                 # 1
print(queue.is_empty())             # False
queue.claim()
print(queue.size())                 # 0
```

Vul ook hier je naam in vóór je verder gaat:	

Question 1.

Implement the `__init__` method for `Question`. The parameters `question` and `student_name` are strings, and the parameter `track` is an integer that should correspond to one of the types of students in the `TRACK_NAMES` dictionary. Make sure you save each parameter into the object.

```
def __init__(self, question, student_name, track):
```

Question 2.

Implement the `description` method for `Question`. It should return a string describing the question like in the examples. Remember to use the `TRACK_NAMES` dictionary.

```
def description(self):
```

Question 3.

Implement the `add` method for `Queue`. It should add a given question to the end of the queue **and** return the number of items in the queue after adding (so we can show our student at what position they are!). We recommend using the `append()` method for lists.

```
def add(self, question):
```

Question 4.
Implement the `is_empty` method for `Queue`. It should return `True` if the queue is empty and `False` otherwise.

```
def is_empty(self):
```

Question 5.
Implement the `peek` method for `Queue`. It should return the element at the front of the queue but `not` remove it, for example to check what question was asked.

```
def peek(self):
```

Question 6.
Implement the `find` method for `Queue`. It should return a list containing all questions for a particular track (extracted from all questions that are currently in the queue). The track is given as a number.

```
def find(self):
```

Question 7.
Implement the `print_list_in_order` method for `Queue`. It should print on each line: a number 1, 2, 3, etc - followed by one question's description.

```
def print_list_in_order(self):
```

Debuggen

Bucketsort is een algoritme om een lijst te sorteren. Daarbij worden waardes uit de lijst in verschillende buckets gezet. Deze buckets worden daarna gesorteerd of worden gesorteerd gehouden tijdens het vullen. Hieronder staat een implementatie:

bucket_sort.py
```python
def bucket_sort(numbers):
    small_bucket = []
    medium_bucket = []
    big_bucket = []

    for number in numbers:
        if number < 50:
            insert(small_bucket, number)
        elif number < 100:
            insert(medium_bucket, number)
        else:
            insert(big_bucket, number)

    return small_bucket + medium_bucket + big_bucket

def insert(bucket, number):
    for i in range(len(bucket)):
        if bucket[i] > number:
            bucket.insert(i, number)
            break

print(bucket_sort([3,79,81,6]))
```

Een test van de code krijgen geeft de volgende uitkomst:

```
$ python bucket_sort.py
[]
```

De uitkomst van `bucket_sort()` is een lege lijst, terwijl er [3,6,79,81] verwacht wordt.

Question 8.
Leg uit wat er fout gaat **en** hoe je dit kan oplossen.

Voor een lokale supermarkt worden de gedane boodschappen van de dag bewaard in een dictionary. De keys zijn hier de gekochte producten (`bread`, `cheese milk`) en de values lijsten van klanten die de producten hebben gekocht. Nu wil dezelfde supermarkt weten wat verschillende klanten kopen, daarom is onderstaand programma geschreven. Dit programma zou de bestaande dictionary moeten omdraaien, zodat de keys de klanten zijn en de values een lijst van gedane boodschappen. Het werkt alleen net niet helemaal.

```
groceries.py
1)  def flip(groceries):
2)      customers = {}
3)
4)      for item in groceries:
5)          for customer in groceries[item]:
6)              customers[customer].append(item)
7)
8)      return customers
9)
10) groceries = {"bread": ["alice", "bob", "charlie"],
11)              "cheese":["alice"],
12)              "milk":["alice", "bob"]}
13)
14) print(flip(groceries))
```

Bij een test van de code gebeurt het volgende:

```
$ python groceries.py
Traceback (most recent call last):
  File "groceries.py", line 14, in <module>
    print(flip(groceries))
  File "groceries.py", line 6, in flip
    customers[customer].append(item)
KeyError: 'alice'
```

Question 9.
Leg uit wat er fout gaat **en** hoe je dit kan oplossen.