COVER PAGE FOR DIGITAL ON CAMPUS EXAMINATION IN CANVAS

Name of the course: Data Structures & Algorithms
Course code: 822188-B-6
Date of the examination: 19/01/2021
Duration of the examination: 3 hours
Lecturer: Pieter Spronck **ANR**: 447823
Lecturer's telephone number available during examinations: 8118

Students are expected to conduct themselves properly during examinations and to obey any instructions given to them by examiners and invigilators. Firm action will be taken in the event that academic fraud is discovered.

All mobile devices (telephone, tablet, etc.) must be switched off in the exam room and should be stored in your jacket or bag. You are not allowed to carry these during the exam. It is also not allowed to wear a watch (both analog and digital) in the exam room.

Exam instructions

1. Only the following examination aids are permitted during the exam:
   a. Books, lecture notes (either in printed form, book form, or digitally supplied on the university computers):
      - Python documentation as supplied with Python
      - The book "The Coder's Apprentice"
      - The book "Think Python"
      - Notebooks, either local notebooks or on the JupyterLab server (jupyterlab.uvt.nl)
      - A Python cheat sheet as supplied on Canvas
      - Lecture sheets as supplied on Canvas
      - Anything else that is supplied on Canvas for this course
   b. Blank scrap paper and pens. Scrap paper and other hard copy materials may be taken home by the students at the end of the exam.
   c. Student's own calculator (any).
   d. Internet sources: https://jupyterlab.uvt.nl (this is the Jupyterlab server of the university).
   e. Computer programs: Any of the programs in the default installation, including Python, Anaconda, Spyder, Jupyter, and IDLE.
2. In case a student does not comply with instructions paragraph 1, this can be qualified as (a suspicion of) fraud. The EER, Rules and Guidelines of the Examination Board of the School and the General Guidelines apply.
3. The exam has 5 questions.
4. The expected passing score (the score to pass the exam) is 6. The score is calculated by taking 40% of the midterm score and 60% of the exam score, unless the exam score is higher than the midterm score (or the student has no midterm

score), in which case the exam score counts for 100%. Standard rounding to half points applies, except that 5.5 or higher will be rounded to 6, while 5.4 or lower will be rounded to 5.

5. Each question is worth 2 points.
6. All questions require the student to write code, or write comments in a code file. For writing code, the main requirement for getting all the points for the answer is that the code processes the tests given by the instructors correctly. In degenerative cases (such as code being unintelligible, code being exceptionally slow, code only being able to handle the test cases supplied in the template files but not other test cases, or a student replacing a loop with a long list of conditional statements) points will be subtracted. Code that crashes may not get points at all.
7. Each question has its own template code file, in which the students have to fill in a function, or add comments to functions. These code files are to be submitted as answers. Please meet the following requirements:
   a. Students should add their name and student number at the top of each file as a comment.
   b. Only the standard Python modules may be used to develop code (as discussed in the first 32 notebooks), so no numpy or pandas.
   c. Students may add extra tests in the main() function that is found in each of the code files.
   d. If a student needs to add print() statements in a function that they are developing for debugging purposes, these print() statements should be removed before submitting the final code.
   e. Functions never need to ask the user for input. All inputs for the functions are supplied via parameters.
   f. If a student wants to use the notebooks as editor, then they should copy the code from the Python files to the notebook in which they want to do the editing, and after having finished their code, copy the code back to the original file. They then submit the original file.
   g. Code files should be tested before submitting, especially if notebooks were used to develop the code (to catch copying mistakes).
   h. If a student wants to add remarks to submitted code, please do so as comments in the code files that are submitted.
   i. Five separate code files must be submitted. Make sure that they are Python (`.py`) files! Do not change the names of the files! Do not pack them in a ZIP file! Do not submit `.ipynb` files! Do not submit screenshots!
8. Not all questions are equally hard, and it tends to vary between students what they consider to be hard. If you are stuck on a question, turn to the next one, and return to the question you got stuck on later. Think before you start coding – finding a good approach tends to save a lot of time.
9. If a student has questions during the exam, they should ask those to an instructor present, or in the Discussion forum on Canvas. The instructors will keep track of those questions and answer them. Students should NOT post code in the forum! Students may see each other's questions, but should NOT answer each other's questions! Communicating on the contents of the exam with anyone but the instructors is not allowed!

10. The instructors will supply a discussion of each of the questions on Canvas after the grading has been completed. If any questions remain after you have viewed these discussions, please contact the instructors via the Canvas Inbox to arrange an exam inspection. This may be arranged via Zoom or Teams.

You can start the examination now, good luck!

Resit Data Structures & Algorithms 2021/2022

Please read the exam instructions on the cover page carefully!

Note that academic fraud amounts to handing in code that you did not write yourself, either because you got someone to help you, or because you copied someone else's answers, or because you consulted internet sites which you are not allowed to consult during the exam!

However, you may use code which you find in the books or notebooks to base your answers on.

What academic fraud is and the fact that you are not allowed to commit it, has been pointed out to you now multiple times during the class, via a lecture, via an announcement, via a video, via the cover page of this exam, and right here on this page.

When academic fraud is detected, the exam board will be notified immediately. Consequences may be severe. Denial of knowledge of academic fraud will not be accepted as an excuse.

If you need to download the exam because you received this page on paper, then please go to the Canvas course, and find the exam (it will be either under Assignments or Quizzes). You find all the files there, and you can also submit your answers there.

Make sure that you place any `.txt` file which is provided in the same folder as where you edit the code for the third exercise. If you do not, your code may not be able to see them. In particular, if you use the Jupyter server for coding, upload them too.

If you have questions during the exam, you can ask them via the Discussion forum for the Canvas course. You may read each other's questions, but you may not answer them – this would actually be a violation of the rule that you cannot communicate with others during an exam. Leave the answering to the instructors.

We wish you good luck.

Exercise 1: Armstrong numbers in range

File: `E01_armstrong_range.py`

One-line summary

In this exercise, you return all the Armstrong numbers between two integers.

Context

Armstrong numbers are numbers the digits of which, when cubed and summed, are equal to the number itself. For example, the number 153 has the digits 1, 5, and 3; $1^3 + 5^3 + 3^3 = 153$, which means that 153 is an Armstrong number. Note: 0 and 1 are also Armstrong numbers.

Exercise description

Write a function `armstrong_range()` which takes two arguments, `start` and `finish`, which specify the lower and upper bound of a range of integers. Both `start` and `finish` are 0 or greater, and `finish` is greater than or equal to `start`. The function checks whether each number in the range, starting from `start` and ending with `finish`, is an Armstrong number.

The function returns a list of all Armstrong numbers in the range provided.

Examples

`armstrong_range( 0, 1000 )` returns `[0, 1, 153, 370, 371, 407]`
`armstrong_range( 0, 100 )` returns `[0, 1]`
`armstrong_range( 100, 370 )` returns `[153, 370]`
`armstrong_range( 50, 100 )` returns `[]`
`armstrong_range( 407, 407 )` returns `[407]`

Note

Of course, your program should calculate the Armstrong numbers and not just return the values given in the examples.

Exercise 2: Count substrings in file

Files: `E02_find_substrings, wikipedia.txt, jeeves7.txt`

One-line summary

In this exercise you search the contents of a file for substrings in unique words and return how many there are.

Exercise description

The function `find_substrings_in_file()` gets two parameters: a `file` name and a `target` substring. The function returns an integer that is the number of words in the file that contain the `target`, without counting duplicate words multiple times.

For example, if the target is “Blitz” and there are three instances of the word “Blitzkrieg” and one instance of “Blitzkreeg” in the file, then the integer returned should be 2 (assuming that no other words in the file contain “Blitz”).

You may assume that words only consist of letters of the alphabet, and that all other characters in the file are punctuation. I.e., “didn’t” is considered to be two words, “didn” and “t”. Handle the words in the file case-sensitively.

If the `target` is an empty string, since an empty string is a substring of any word, the function will return the total number of unique words in the file.

If the file does not exist, return -1. You do not need to handle any other file errors.

Examples

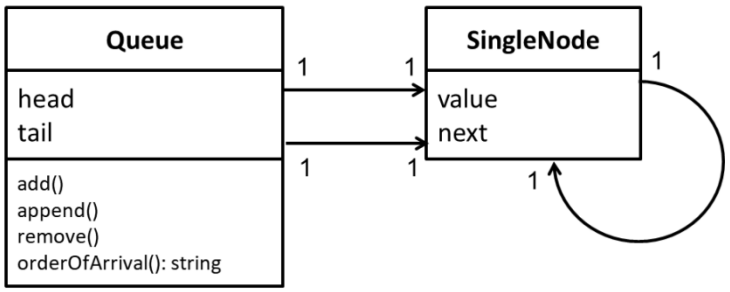
For the file “wikipedia.txt”:

If the target is “recipient”, the function returns 2. There are four words in the file which contain “recipient” as substring, namely twice the word “recipient” and twice the word “recipients”. This means that there are two different words which contain “recipient” as substring, and thus the return value is 2.

If the target is “R”, the function returns 4, since in the file the words `Rockerfeller`, `Rockefeller`, `Russion`, and `Russian` all contain “R”. The word `Russian` occurs twice, so you count it only once.

Exercise 3: Keeping order in a queue

File: `E03_queue`



One-line summary

In this exercise you will be keeping track of the order in which people arrive to a queue.

Exercise description

The code given for this exercise contains two classes: `SingleNode` and `Queue`. Queue is actually a `SingleLinkedList`, as introduced in the course. The code for this class (which is taken directly from the notebooks) contains methods `__init__()`, `add()`, `append()`, and `remove()`. `add()` adds a new `SingleNode` at the head of the `Queue`, `append()` appends a new `SingleNode` at the tail of the `Queue`, and `remove()` removes the `SingleNode` which is at the head of the `Queue` (if the `Queue` is not empty, otherwise it will generate an exception).

The `Queue` will be used to keep track of a line of people. New arrivals will either go to the tail of the queue, or will be placed at the head of the queue (depending on whether `add()` or `append()` is used). Someone may get removed from the queue; if so, this is done with the `remove()` method, so they will be removed from the head of the queue.

In this exercise you have to implement the method `orderOfArrival()` for the `Queue`, which returns a string that tells us the order in which people have arrived to join the queue (separating the names of the people with the word “then”), regardless of whether they were added to its head or its tail.

When creating your solution, you are allowed to change the existing methods of the class `Queue` (in fact, that is pretty much necessary). You may also make changes to the `SingleNode` class if you want. You may assume that the `remove()` method is only used when there is at least one person in the queue.

Example

```
queue1 = Queue()
queue1.append("Bobby")
queue1.add("Xerxes")

print( queue1.orderOfArrival() )
```

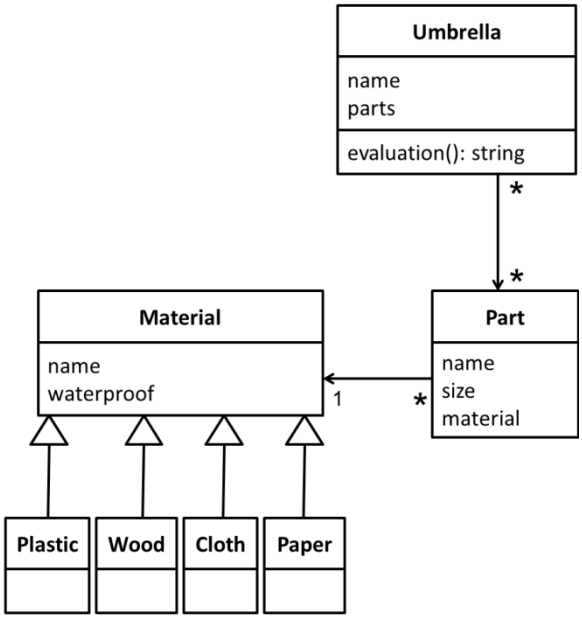
Should return: `Bobby then Xerxes`

Exercise 4: Umbrella

File: `E04_umbrella.py`

One-line summary

In this exercise, you will build umbrellas from parts, and evaluate their quality.



Context

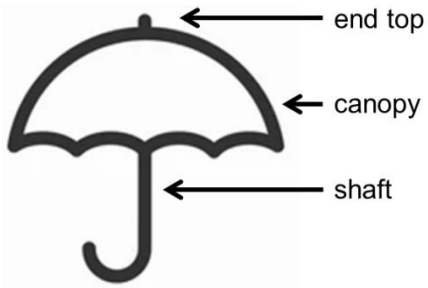
Parts that fit together to build a new thing are not always optimal. They may be of the wrong material, or too heavy. In this exercise you will make sure that they are not.

Exercise description

The exercise file contains a class `Material`, which has a name, and a boolean which indicates whether the material is waterproof. From `Material` four classes are derived: `Plastic`, `Wood`, `Cloth`, and `Paper`.

Two classes in the exercise file need implementation: `Part` and `Umbrella`.

The class `Part` represents a part of an umbrella. For this exercise, the umbrella parts are the shaft, the canopy, and (optionally) the end top. A `Part` object has a name, a size, and a material from which it is constructed. These are given to the `__init__()` method of `Part`. You have to implement the `__init__()` method, and a suitable `__repr__()` method for `Part`.



The class `Umbrella` represents a model umbrella. Via the `__init__()` method it is given a name and a list of parts. You have to implement the `__init__()` method, and a suitable `__repr__()` method for `Umbrella`.

Finally, you have to implement a method `evaluation()` for `Umbrella`. This method evaluates whether the umbrella meets all the requirements that it should have. The requirements are: (1) the sum of the sizes of the parts of an umbrella should be 5 at most; and (2) all the parts of the umbrella should be waterproof. The method returns a string which is the evaluation of the model. There are four possible evaluations, namely:

"<name> meets the requirements"	(total size <=5 and all parts are waterproof)
"<name> is too big"	(total size > 5 and all parts are waterproof)
"<name> is not waterproof"	(total size <= 5 and not all parts are waterproof)
"<name> is too big and not waterproof"	(total size > 5 and not all parts are waterproof)

Example

```
plastic = Plastic()
paper = Paper()
plastic_shaft = Part( "plastic shaft", 2, plastic )
paper_canopy = Part( "paper canopy", 2, paper )
umbrella_3 = Umbrella( "Model 3", [plastic_shaft, paper_canopy] )
umbrella_3.evaluation() returns “Model 3 is not waterproof”
```

Exercise 5: Smallest plus largest

File: `E05_smallest_plus_largest.py`

One-line summary

In this exercise you have to determine the time complexity of five functions which determine the sum of the smallest and the largest number in a list of numbers.

Exercise description

Each of the functions, which are all named `smallest_plus_largest_`*x*`()` (*x* being a number), gets one parameter: `numlist` which is a list of numbers (the length of the list is at least 1). The function returns the sum of the smallest and the largest number on the list.

For each of the functions, write as a comment in the file what the time complexity is (immediately below the function), and give a brief explanation on how you determined this. If you do not add an explanation, the answer is automatically considered to be wrong. In your big-O notation of the time complexity, you may use `n` to refer to the length of the list.

Background information

You may assume the following:

- Taking a sublist from a list is `O(n)`, where `n` is the length of the sublist
- The iterator `range()` is `O(1)`
- The function `len()` is `O(1)`
- The function `min()` is `O(n)`, where `n` is the number of items given to the function
- The function `max()` is `O(n)`, where `n` is the number of items given to the function
- The list method `sort()` is `O(n*log(n))`, where `n` is the length of the list

Here are descriptions of the algorithms:

- `smallest_plus_largest_1()` uses `min()` and `max()` on the list.
- `smallest_plus_largest_2()` goes through the list, remembering the smallest and largest.
- `smallest_plus_largest_3()` does the same as the previous, but uses indices.
- `smallest_plus_largest_4()` sorts the list then takes the first and last number.
- `smallest_plus_largest_5()` uses a recursive implementation of functions that determine the smallest and largest number on a list; these functions split the list in two and compare the numbers returned by a recursive call on the sublists, to determine the smallest and largest respectively.