INF109: Tekna Crash Course

Torstein Strømme

Saturday, November 21, 2015

Contents

1 Strings and Text

1.1 Character count

Write a function *count_characters*(s, c) which takes a string s and a character c as input, and returns an integer indicating the number of occurrences of the character c in the string. Write the code without using the built-in string method *count*()

```
>>> # Example:
>>> count_characters("You are an analog circuit", "a")
4
```

1.2 Replace all v1

Write a function *replace_all*$(text, s, r)$ which takes as input three strings *text*, s, r, and returns a single string wich is a copy of *text* except that every occurrance of the word s is replaced by the string r. You may assume that s contain no spaces, and you need not replace occurrances of s which occurs as a substring of another word in *text*. Write the code without using the built-in string method *replace*().

```
>>> # Example:
>>> replace_all("You are an analog circuit", "an", "not an")
'You are not an analog circuit'
```

1.3 Find

Write a function $find(text, s)$ which takes as input a text string *text* and a search string s, and returns the index of the first letter of the first occurrance of s in *text*. If none can be found, return -1. Write the code without using regular expressions or the built-in string methods $find()$ and *index*().

```
>>> # Example:
>>> find("You are not analog circuits, are you?", "are")
4
```

1.4 Replace all v2

Write a function *replace_all*$(text, s, r)$ which takes as input three strings and returns a single string wich is a copy of *text* except that every occurrance of the word s is replaced by the string r. You should replace instances greedily, i.e. replace first the first occurance in s, and then countinue searching from the first unreplaced character not including the replacement text. Take care to make sure you do not crash even if s is longer than *text*. Write the code without using the built-in string method *replace*().

```
>>> # Example:
>>> replace_all("bbabaa, ababazz", "aba", "xxab")
'bbxxaba, xxabbazz'
```

1.5 Palindrome (string)

Write a function *is_palindrome*(s) which takes as input a string s and returns True if s is a palindrome and False otherwise. A string is a palindrome if it is the same when the characters are reversed.

```
>>> # Example:
>>> is_palindrome("agnes i senga")
True
```

1.6 Bracket order

We are given strings containing brackets of 3 types - round ( ), square [ ] and curly { } ones. The goal is to check whether brackets are in correct sequence, i.e. any opening bracket should have closing bracket of the same type somewhere further by the string, and bracket pairs should not overlap, though they could be nested:

```
(a+[b*c] - {d/3})  # Square and curly brackets are nested in the round ones
(a+[b*c) - 17]     # Square brackets overlap with round ones which does not make sense
```

Write a function *check_brackets*(s) which takes as input a string s, and returns True if the bracket order makes sense, and False otherwise.

```
>>> # Example:
>>> check_brackets("(a+[b*c] - {d/3})")
True
>>> check_brackets("(a+[b*c) - 17]")
False
```

1.7 Textframe

a) Write a function $frame(line, patt)$ which takes as input a string *line* and a pattern string *patt*, and returns the text of *line* surrounded by the pattern as shown below. Note that the pattern on the rigth side of the returned string is mirrored. You may assume that neither *line* nor $frame$ contains any newline characters.

```
>>> # Example:
>>> text = "The cat and the mouse used to be friends."
>>> frame(text, "-*")
'-* The cat and the mouse used to be friends. *-'
```

b) Write a function $frame\_2d(text, patt)$, which lifts the ASCII frame from task a) to two dimensions. Make sure you handle strings with newlines in them, as shown in the example below. You may use the function from a) as a subroutine if you want. You may assume that *patt* does not contain any newline characters.

```
>>> # Example:
>>> text = "The cat\nand the mouse\n  used to be friends."
>>> res = frame_2d(text, "**@*")
>>> print(res)
******************************
******************************
**@@@@@@@@@@@@@@@@@@@@@@@@@@@@**
**@**************************@**
**@* The cat                *@**
**@* and the mouse          *@**
**@*   used to be friends. *@**
**@**************************@**
**@@@@@@@@@@@@@@@@@@@@@@@@@@@@**
******************************
******************************
```

1.8 Autograder

When a certain teaching assistant in INF109 is grading compulsory assignments, he is very peculiar about style. In particular, he thinks it is very important that the students follow the 80-character rule, which states that no line should contain more than 80 characters. In fact, he cares about this rule so much that it is the only criteria by which he judge submissions. Now, grading is hard work, and the TA has tasked you with automating the grading process for him.

Your task is to write a function $grade(filename)$ which takes as input a string with the name of a python file submitted by a student, and returns True if the student passed the assignment, and False if the student failed.

```
>>> # Example:
>>> grade("vag002-assignment8.py")
True
>>> # returned True, since student vag002 writes with impeccable style, and
>>> # thus his submission contains no lines above 80 characters.
```

1.9 Autoadjust

In order to get full marks on your assignments, you have noticed that it is a good idea to keep the lines of your submission below 80 characters. (In fact, you have heard rumors of this one notourious TA which directly fails anybody who submit a file with 81 characters or more on the same line.) However, since your sense of style is equivalent to that of an obnoxious monkey, your submissions sometimes contain lines with 81, 82 or even more characters. You have therefore decided to write a script which automatically formats your submission in accordance with the style guidelines.

Your task is to write a function $below\_80(infile, outfile)$ which takes as input a string with the name of an existing file $infile$, and the name of a file to be created $outfile$. The outfile should contain the same text as that of the infile, however no line should exceed 80 characters. Any line which is longer than 80 characters should be broken into as few lines as possible such that none of them exceed 80 characters. Moreover, you are not allowed to break any line in the middle of a word, you can only insert new lines between words. You may assume that no words are longer than 80 characters, and that all words are separated by spaces. (For simplicity we do not require that the logic of any code is maintained, or that the created file will even be able to run.)

```
>>> # Example:
>>> grade("sketch.py")
False
>>> # returned False, since sketch.py contained several lines exceeding 80
>>> # characters.
>>> below_80("sketch.py", "final.py")
>>> grade("final.py")
True
>>> # returned True, since no lines of final.py exceed 80 characters.
```

1.10 CSV to text

Comma separated values (CSV) is a popular text readable file format for spreadsheets in which each row is separated by a newline, and the values of the different colums in the row is separated by comma. For instance, a CSV-file representing a phone book may look something like this (phonebook.csv):

```
Last Name, First Name, City, Phone Number
Smith, Adam, Starling City, 12345678
Jones, Sarah, Gotham, 23456789
Johnsson, Petter Olav, Smallville, 34567890
```

A CSV like the one above can be a bit messy, so you have decided to make it a bit nicer by aligning the entries of each column. You have decided to make a function $nice\_csv(infile, outfile)$ which takes as input the filenames of an existing CSV-file $infile$, and a text file to be created $outfile$. The outfile should show the table described by the infile, but commas are removed and columns are aligned. There should be a minimum of two spaces between each column, but otherwise the distance between the columns should be as small as possible.

```
>>> # Example:
>>> nice_csv("phonebook.csv", "phonebook.txt")
```

This should create the file phonebook.txt:

```
Last Name  First Name   City           Phone Number
Smith      Adam         Starling City  12345678
Jones      Sarah        Gotham         23456789
Johnsson   Petter Olav  Smallville     34567890
```

1.11 Newspaper categorization

Your friend at the social sciences department is conducting a research study of local newspapers and what topics they write about compared to what natinal newspapers write about. Most newspapers are online these days, and your friend has already found a way to collect all the newspaper articles from local newspapers around the country in a big file. However, seeing that there are millions of articles, he has now hired you to automate his counting process.

Your task is to write a function $count\_topics(articlefile, topics)$ which count how many articles are written about each topic. The function should output a list of tuples where the first element in the tuple is the topic, and the second element in a tuple is the count of how many articles cover that topic. An article may cover several topics.

The first parameter of the function is a string $articlefile$ which has the name of a file containing articles. Articles are separated by newline characters, and no article is longer than one line (those pesky journalists almost always break the 80-character rule, though).

The second parameter of the function is a list *topics*, where each topic is a 3-tuple; the first element of the tuple is the topic itself. The second and third elements of the tuple are the most prominent buzz-words in that topic. If any of the three words occur in an article, that article should be counted towards that topic. (Note that a buzz-word can be a buzz-word for several different topics, and one topic may be a buzz-word for another topic.)

Example: articles.txt

```
sam had lion for dinner and invited all his friends
there was a burglary in king oscar's street monday
cat was pretending to be tiger in morewood avenue
the cat and the mouse used to be friends
```

```
>>> # Example
>>> topics = [("cat", "lion", "tiger"), ("road","street","avenue")]
>>> count_topics("articles.txt", topics)
[("cat", 3), ("road", 2)]
```

2 Lists

2.1 Most common

Write a function *most_common*(*lst*) which takes a list of integers as input and returns the value occurring most frequently. If there is tie between two or more elements, return the smallest among them.

```
>>> # Example:
>>> lst = [1, 5, 3, 6, 2, 7, 4, 2, 4, 6, 2]
>>> most_common(lst)
2
```

2.2 Palindrome (list)

Write a function *is_palindrome*(*lst*) which takes a list of elements as input and returns whether the list forms a palindrome. We say that a list is a palindrome if the first element is the same as the last element, the second element is equal to the second to last element etc. In other words, the list is equal to the reversed version of itself.

```
>>> # Example:
>>> lst = [1, 'a', 4.0, 'a', 1]
>>> is_palindrome(lst)
True
```

2.3 Evens and odds

Write a function *splitt_even_odd*(*lst*) which takes a list of elements as input and returns two lists, containing the even and odd indexed elements respectively.

```
>>> # Example:
>>> lst = [1, 5, 3, 6, 2, 7, 4, 2, 4, 6, 2]
>>> split_even_odd(lst)
([1, 3, 2, 4, 4, 2], [5, 6, 7, 2, 6])
```

2.4 Odds and evens

Write a function *merge_even_odd*(*even*, *odd*) which takes two list of elements as input and returns a single lists where the two input lists are intertwined. In case that one list is shorter than the other, the last elements of the longer list should appear back to back at the end of the returned list.

```
>>> # Example:
>>> even = ['a', 'b', 'c']
>>> odd = [1, 2, 3, 4, 5, 6]
>>> merge_even_odd(even, odd)
['a', 1, 'b', 2, 'c', 3, 4, 5, 6]
```

2.5 Reorder

Nigel always finds himself in the middle of the alphabet. This was always a source of distress for Morten, who never experienced to be called up first whenever the class was arranged alphabetically to present something. Now that he himself is a teacher, it is time to correct this misordering from the universe, so at first Nigel

wanted to pick a completely random ordering of his students. However, that turned out to be impractical. Instead, he decided he would pick the student to go *first* at random, and then go alphabetically from there.

Picking a number at random, Nigel can do. But in order to find the ordering, he asks you to write the function *reorder(students, k)* which takes as input a list *students* of names in alphabetical order of length n, and an integer k, the index of the first student to present. The function should return a list in the order they should present

```
>>> # Example:
>>> students = ['Andre', 'Sara', 'Tom']
>>> reorder(students, 1)
['Sara', 'Tom', 'Andre']
```

2.6 Unflatten

```
def flatten(list2d):
    list1d = []
    for row in list2d:
        for cell in row:
            list1d.append(cell)
    return list1d
```

The above function *flatten* takes as input a 2D list, and outputs a 1D list with the elements in row-major order. Your task is to do the reverse, i.e. write a function *unflatten(list1d)* which takes as input a 1D list which is in row-major order, and returns a 2D list which satisfy the following requirements:

- Every row is of equal length
- The difference between the row length and the number of rows (column height) is as small as possible.
- The row length can not exceed the number of rows

For example, if the input list has 20 elements, the output list should have dimensions 5x4, not 10x2 or 4x5.

```
>>> # Example:
>>> list = [0, 1, 2, 3, 4, 5]
>>> unflatten(list)
[[0, 1], [2, 3], [4, 5]]
```

3 Input and eval()

3.1 Integrate

Write a program intgrate() which prompts the user for a function and a range. Then make an estimate of the integral for the function under the given range by evaluating the function for 1000 samples evenly distributed in the range [x0, x1].

```
>>> # Example
>>> integrate()
f(x) = ##-- Here, the user inputs:   x+10 --##
x0, x1 = ##-- Here, the user inputs:   0, 10 --##
149.94999999999996  # Estimate for closed integral from x=0 to x=10 for f(x)=x+10
```

3.2 Calculator

Using the python shell as a calculator is nice, but you think there is some room for improvement seeing for instance that the prompt gives no indication that the shell is in fact a calculator. So you decide you want a "calculator mode" for your shell. You want to start calculator mode by calling a function *calc*(), and leave the calculator when the user types "exit" in the calculator prompt.

Write the function *calc*() which takes no arguments, but starts a calculator mode in the shell. Any python-formatted mathematical expression should now be possible to write, and the result should be shown. The prompt itself should also look different, to indicate that this is actually calculator mode. (Note: you should avoid that eval(str) crash when the user does not input anything)

```
>>> # Example
>>> calc()
calc> 2+3
5
calc>                # User does not input anything, new prompt is shown
calc> cos(0)
1
calc> exit
>>>
```

3.3 Avereganza deviazion

Given a collection of n different *mathematical* functions $f_1(x), f_2(x), \ldots f_n(x)$, we define the *avereganza* $f_a(x)$ as the average value over the formulas, i.e. $f_a(x) = (f_1(x) + f_2(x) + \ldots + f_n(x))/n$.

a) Write a function $f\_avg(funcs, x)$ which calculates $f_a(x)$. As input, it takes a list $funcs$ of length n containing strings representing the functions $f_1(x), f_2(x), \ldots f_n(x)$, and a real number x. The function returns a float indicating the averaganza value f_a evaluated at the point x.

```
>>> # Example:
>>> funcs = ["2*x - 1", "sin(x**2)", "0.2*(x-5)**2 + 2"]
>>> f_avg(funcs, 0)
2.0
```

Now we define the *deviazion* $d_i(x)$ of a function f_i at the point x as the absolute difference between $f_i(x)$ and $f_a(x)$, i.e. $d_i(x) = |f_a(x) - f_i(x)|$.

b) Write a function $f\_dev(funcs, i, x)$ which calculates the deviazion for f_i at the point x, i.e. which calcuates $d_i(x)$. As input, it takes a list $funcs$ of length n containing strings representing the functions $f_1(x), f_2(x), \ldots f_n(x)$, an integer i and a real number x. The function should return a float indicating the value of $d_i(x)$.

```
>>> # Example:
>>> funcs = ["2*x - 1", "sin(x**2)", "0.2*(x-5)**2 + 2"]
>>> f_dev(funcs, 1, 0) # Calculates deviazion for "sin(x**2)" at the point x = 0
2.0
```

c) Write a function $sort\_by\_dev(funcs, x)$ which sorts the functions by their deviazion evaluated at the point x. As input, it takes a list $funcs$ of length n containing strings representing the functions $f_1(x), f_2(x), \ldots f_n(x)$, and a real number x. The function should return a list of length n containing the same functions as $funcs$, but such that they are sorted by increasing deviazion value.

```
>>> # Example:
>>> funcs = ["2*x - 1", "sin(x**2)", "0.2*(x-5)**2 + 2"]
>>> sort_by_dev(funcs, 0)
["sin(x**2)", "2*x - 1", "0.2*(x-5)**2 + 2"]
```

3.4 Hacker

So you just came across a program like the one below running at some website.

```
def find_root():
    num = eval(input("Square root of which number? "))
    print(num**0.5)
```

Being a sesoned hacker who knows the capabilities of eval, you consider taking over the server where this program runs. However, on second thought you decide that a fierce warning will suffice.

Your task is to craft a message which exploits eval() to print a message of your choice to the screen, warning the author of the above program against the dangers of using eval in his code. It is OK to crash the program *after* your message is displayed, though there could be a bonus point hidden here if you do not...

```
>>> # Example:
>>> find_root()
Square root of which number? ##-- Here goes the message you craft --##
DANGER! USING EVAL() IS EVIL! IT IS ONLY INTENDED FOR USE AT THE INF109 FINAL!
0 # At this point the program can crash or print any number
```

3.5 Guessing game

Write a program *guessing_game(n)* which takes as input an integer n and plays the guessing game with the user through a text interface. The guessing game works as follows: The user gets to guess the number n. Then the program tells the user whether the guess was too low, too high, or just right. The game ends when the user guesses the correct number, or when he enters a blank line as input (no guess). If the user guesses the correct number, the function should print how many guesses he made.

```
>>> # Example:
>>> guessing game(8)
Guess my number >>> 10
Too high!
Guess my number >>> 5
```

```
Too low!
Guess my number >>> 8
Correct! You used 3 guess(es) to find the number!
>>> guessing_game(4)
Guess my number >>> -3
Too low!
Guess my number >>>                                ##-- Blank input from user --##
Aww, you did 1 guess(es), but then gave up :(
```

4 Numbers and counting

4.1 Palindrome (number)

Write a function *is_palindrome*(n) which takes as input an integer n and returns True if n is a palindrome and False otherwise. An integer is a palindrome if it is the same when the digits are reversed.

```
>>> # Example:
>>> is_palindrome(14041)
True
```

4.2 Approximation of e

The number e is an important mathematical constant that is the base of the natural logarithm. It is approximately equal to 2.71828, and is defined as the limit of $(1 + 1/n)^n$ as n approaches infinity. For some strange reason, computer scientists love e. For instance, in the IPO filing for Google in 2004, rather than a typical round-number amount of money, the company announced its intention to raise \$2,718,281,828, which is e billion dollars rounded to the nearest dollar. In another instance, the famous computer scientist Donald Knuth let the version numbers of his program Metafont approach e. The versions are 2, 2.7, 2.71, 2.718, and so forth.

Another definition of e is the sum of the infinite series below. Note that $0! = 1$.

$$e = 1 + 1 + \frac{1}{1 \cdot 2} + \frac{1}{1 \cdot 2 \cdot 3} + \cdots = \sum_{k=0}^{\infty} \frac{1}{k!}$$

Write a program *estimate_e(n)* which estimates e by computing the sum of the first n terms of the infinite series above.

```
>>> # Example:
>>> estimate_e(3)
2.5
```

4.3 Approximation of π

By the 5th century CE, π was known to about seven digits in Chinese mathematics, and to about five in Indian mathematics. Further progress was not made for nearly a millennium, until the 14th century, when Indian mathematician and astronomer Madhava of Sangamagrama, founder of the Kerala school of astronomy and mathematics, discovered the infinite series for π, now known as the Madhava-Leibniz series, and gave two methods for computing the value of π. One of these methods is to obtain a rapidly converging series by transforming the original infinite series of π. By doing so, he obtained the infinite series:

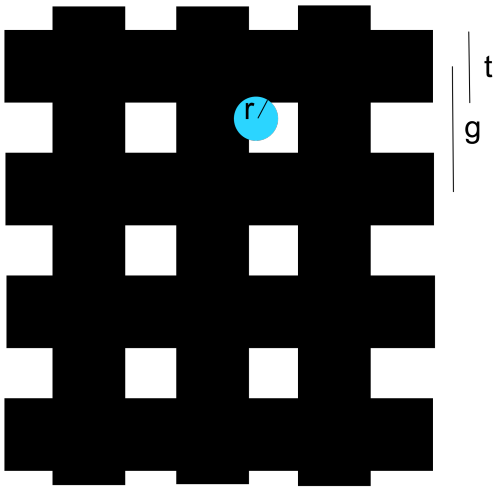
$$\pi = \sqrt{12}(1 - \frac{1}{3 \cdot 3} + \frac{1}{5 \cdot 3^2} - \frac{1}{7 \cdot 3^3} + \cdots) = \sqrt{12} \sum_{k=0}^{\infty} \frac{(-1)^k}{(2k+1) \cdot 3^k}$$

Write a function *estimate_pi(epsilon)* which estimates π to within some small constant float value *epsilon* using the series above. (Note that due to each term decreasing in absolute value whilst alternating in sign, we can know that we are no further than *epsilon* away from π when the difference between the current estimate and the previous estimate is smaller than *epsilon*.)

```
>>> # Example:
>>> estimate_pi(0.01)
3.14260474566
```

4.4 Surviving the flyswatter

What are the chances of a fly escaping a flyswatter? A flyswatter is big and intimidating grid-shaped object which sometimes appear out of the middle of nowhere, with the purpose to kill and destroy poor flies. Luckily, in order to swing the flyswatter fast enough, the humans made many holes in the flyswatter, by which they also provided a means to escape.



Write a function $survive(t, g, r)$ which calculates the probability that a fly of radius r escapes a flyswatter with line thickness t and grid spacing g. Assume the fly is a perfect circle with radius r, and that it will hit the flyswatter at a completely random point somewhere in the middle of the swatter. Return a float between 0.0 and 1.0 indicating the probability that the fly will survive. You may assume that $g > 0$.

```
>>> # Examples:
>>> # The flyswatter is solid, no room for escape
>>> survive(5.0, 10.0, 1.0)
0.0
>>> # Even with infinitely thin grids the flyswatter kills the fly at overlap
>>> survive(0.0, 5.0, 1.0)
0.36
```

4.5 Josephus' problem

In the Jewish revolt against Rome (66-70 AD), Josephus and some of his comrades were holding out against the Romans in a cave. With defeat imminent, they resolved that, like the rebels at Masada, they would rather die than be slaves to the Romans. They decided to arrange themselves in a circle. One man was designated as number one, and they proceeded clockwise killing every seventh man... Josephus was among other things an accomplished computer scientist; he wanted to live an join the Romans. Thus, he figured, if he could simulate the execution process, he would know which position would be killed last, and then surrender without sacrificing his honour.

Write a function *josephus(n)* which takes as input an integer n, the number of jewish rebles, and returns an integer indicating the position in the ring which is killed last. The positions in the ring starts at 0 and ends at $n - 1$. The first rebel to be killed is in position 0, the second rebel to be killed is in position 7, and so forth.

```
>>> # Examples:
>>> josephus(10)
```

```
2
```