TÉCNICO LISBOA

Fundamentals of Programming
School year 2021-22 First project
October 15th, 2021

Buggy Data Base (BDB)

In this first Fundamentals of Programming project, students will develop functions that allow them to solve five independent tasks to identify and correct the problems of a database (Buggy Data Base, BDB) that has been corrupted by unknown causes. The BDB contains the authentication information of the users of a system and is wrongly refusing access to some of the registered users. The tasks will consist of: 1) Correcting the documentation; 2) Discovering the database PIN; 3) Verifying the consistency of the data; 4) Decrypting the database; and
5) Password debugging.

1 Correction of documentation

1.1 Description of the problem

The first step to start solving the BDB's problems is to study the documentation provided. Unfortunately, the documentation is also corrupted, but luckily some patterns of changes to the text have been detected that can be reversed. The first pattern of changes affects the words in the text individually: a *burst of letters* formed by pairs of characters of the same case letter were inserted into the words repeatedly. To recover each word, you need to correct this outburst. To do this, when a ridiculous pair of the same letter are adjacent, they *react by* disappearing. For example:

- In 'aA', 'a' and 'A' react, leaving the string empty.
- In 'aBbA', 'Bb' self-destructs, leaving 'aA' which, as above, also reacts by leaving nothing.
- In 'abAB', no two adjacent letters are compatible, so nothing happens.
- In 'aabAAB', although 'aa' and 'AA' represent a pair of the same letter, they are both of the same case and therefore nothing happens.
- In 'cCdatabasacCADde', the following sequence of *reactions is* produced (indicated in square brackets) until the word 'database' is obtained:

  '[cC]databasacCADde' → 'databasa[cC]ADde' → 'databas[aA]Dde' →
  → 'databas[Dd]e' → 'database'

After the *outbreak of letters has been* corrected, there is a second pattern of errors: meaningless words appear in the text that correspond to anagrams of previous words. The solution seems simple: for each new word, we need to detect and eliminate all the anagrams (different from itself) found in the rest of the text.

1.2 Work to be done

The aim of this task is to write a program in Python that allows the BDB documentation to be corrected as described above. To do this, it is necessary to define the set of functions requested, as well as some additional auxiliary functions, if necessary. Only the functions for which checking the correctness of the arguments is explicitly requested should check the validity of the arguments, for the others it is assumed that they are correct.

1.2.1 correct_word: cad. carateres → cad. carateres (1 value)

This function receives a string of characters that represents a word (potentially modified by a *burst of letters*) and returns the string of characters that corresponds to applying the *reduction* sequence as described to obtain the corrected word.

1.2.2 eh_anagram: cad. characters × cad. characters → boolean (0,5 val- res)

This function receives two strings of characters corresponding to two words and returns True if and only if one is an anagram of the other, i.e. if the words are made up of the same letters, ignoring case differences and the order between characters.

1.2.3 correct_doc: cad. characters → cad. characters (1.5 values)

This function receives a string of characters representing the text with errors from the BDB documentation and returns the filtered string of characters with the corrected words and the anagrams removed, leaving only their first occurrence. The anagrams are evaluated after the words have been corrected and only anagrams that correspond to different words (sequences of characters different from the previous words, ignoring case differences) are removed. This function should check the validity of your argument, generating a ValueError with the message 'correct_doc: invalid argument' if your argument is not valid. For this purpose, consider that the words can only be separated by a single space, that the text is made up of one or more words, and that each word is made up of at least one letter (lower case).

1.3 Example

```
>>> correct_word('abBAx') 'x'
>>> correct_word('cCdatabasacCADde') 'database'
>>> eh_anagram('case', 'SaCo')
True
>>> eh_anagram('case', 'cases') False
> > >  correct_doc('??')
ValueError: correct_doc: invalid argument
>>> doc = 'BuAaXOoxiIKoOkggyrFfhHXxR duJjUTtaCcmMtaAGga \
eEMmtxXOjUuJQqQHhqoada JlLjbaoOsuUeYy cChgGvValLCwMmWBbclLsNn \
LyYlMmwmMrRrongTtoOkyYcCK daRfFKkLlhHrtZKqQkkvVKza'
>>> correct_doc(doc)
'Buggy data base has wrong data'
```

2 PIN discovery

2.1 Description of the problem

When you *double-click* to open the file containing the BDB, a window opens with a digit panel that looks like this:

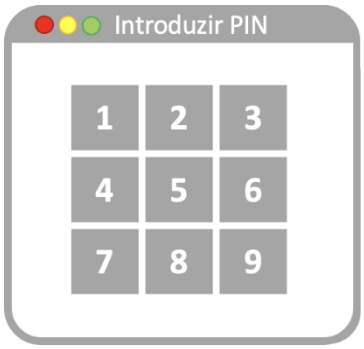

Figure 1: Panel with buttons for entering the BDB PIN.

Apparently, in order to access the data of the database, an unknown PIN is required. However, among the documentation provided is the following text:

> *"To improve security, database codes are no longer written down. Instead, memorize and follow the procedure below."*

The documentation goes on to explain that each button to be pressed can be found by starting at the previous button and moving to adjacent buttons on the keyboard: 'C' moves up, 'B' moves down, 'E' moves left and 'D' moves right. Each line of instructions corresponds to a button, starting at the previous button (or, for the first line, button '5'); press whichever button is at the end of each line. If a movement does not lead to a button, ignore that movement. For example, suppose your instructions are encoded in a string tuple like the following:

('CEE', 'DDBBB', 'ECDBE', 'CCCCB')

Starting at '5', move up ('2'), left ('1') and left (you can't and you stay at '1'), then the first button is '1'. Starting with the previous button ('1'), move to the right twice ('3') and then down three times (stopping at '9' after two moves and ignoring the third), ending with '9'. Continuing from '9', move left, up, right, down and left, ending with '8'. Finally, move up four times (stopping at '2'), then down once, ending with '5'. So, in this example, the database code is 1985.

2.2 Work to be done

The aim of this task is to write a program in Python that allows you to find the BDB PIN as described above. To do this, you must define the set of functions requested, as well as some additional auxiliary functions, if you consider it necessary. Only those functions for which checking the correctness of arguments is explicitly requested should check the validity of arguments.

2.2.1 get position: cad. characters × integer → integer (0.5 values)

This function receives a string containing only one character representing the direction of a single movement ('C', 'B', 'E' or 'D') and an integer representing the current position (1, 2, 3, 4, 5, 6, 7, 8 or 9); and returns the integer corresponding to the new position after the movement.

2.2.2 get digit: cad. characters × integer → integer (0.5 values)

This function receives a string of characters containing a sequence of one or more moves and an integer representing the starting position; and returns the integer that corresponds to the digit to be marked after all the moves have been completed.

2.2.3 get pin: tuple → tuple (1 value)

This function receives a tuple containing between 4 and 10 sequences of movements and returns the tuple of integers that contains the pin encoded according to the tuple of movements. This function must check the validity of its argument as described in this section

(i.e., a tuple of between 4 and 10 sequences of moves, where each move is a string with 1 or more characters 'C', B', 'E' or 'D'), generating a ValueError with the message 'get pin: invalid argument' if its argument is not valid.

2.3 Example

```
>>> get_position('C', 5)
2
>>> get_digit('EEC', 5)
1
>>> get_pin(())
ValueError: get_pin: invalid argument
>>> t = ('EEC',        'DDBBB', 'ECDBE', 'CCCCB')
>>> get_pin(t) (1,
9, 8, 5)
```

3 Data verification

3.1 Description of the problem

You can access the data in the BDB, but of course, the entries are encrypted and the decryption process is computationally heavy. Fortunately, there is a control sequence for each entry that allows us to detect wrong entries and thus reduce the amount of data that needs to be decrypted. Detecting a wrong entry doesn't guarantee that the corresponding password is wrong (because the change can affect other user information in the BDB), but it does reduce the list of suspect cases.

Each BDB entry is represented as a tuple with 3 fields: a *cipher* string containing one or more encrypted words separated by dashes, another *checksum* string containing a control sequence (5 lowercase letters between straight parentheses); and finally a tuple with two or more security positive integers (not yet used at this stage). The words that make up the *cipher* can only consist of lowercase letters (minimum size 1). A BDB entry is only correct if the control sequence is made up of the five most common letters in the encrypted sequence, in reverse order of occurrence, with ties decided alphabetically. For example:

- ('aaaa-bbb-zx-yz-xy', '[abxyz]', (950,300)) is correct because the most common letters are 'a' (5 occurrences), 'b' (3 occurrences) and a tie between 'x', 'y' and 'z', which are listed in alphabetical order.
- ('a-b-c-d-e-f-g', '[abcde]', (124,325,7)) is correct because, although the letters are all tied (1 of each), the first five are listed in alphabetical order.
- ('entry-very-late', '[abcde]', (50,404)) is wrong.

3.2 Work to be done

The aim of this task is to write a program in Python that allows you to validate each of the BDB entries as described above. To do this, you will have to define the set of functions requested, as well as some additional auxiliary functions, if you consider it necessary. Only those functions for which checking the correctness of arguments is explicitly requested should check the validity of arguments.

3.2.1 eh_input: universal → boolean (1,5 values)[1]

This function receives an argument of any type and returns True if and only if its argument corresponds to a (potentially corrupt) BDB entry as described, i.e. a tuple with 3 fields: a cipher, a control sequence and a subsequence.

3.2.2 validate cipher: cad. characters × cad. characters → boolean (2 values)

This function receives a string containing a cipher and another string containing a control sequence, and returns True if and only if the control sequence is consistent with the cipher as described.

3.2.3 filter bdb: list → list (1 value)

This function receives a list containing one or more entries from the BDB and returns only the list containing the entries where the checksum is not consistent with the corresponding cipher, in the same order as the original list. This function must check the validity of its argument, generating a ValueError with the message 'filter bdb: invalid argument' if its argument is not valid.

3.3 Example

```
>>> eh_entry(('a-b-c-d-e-f-g-h', '[abcd]', (950,300))) False
>>> eh_entry(('a-b-c-d-e-f-g-h-2', '[abcde]', (950,300)))
False
>>> eh_entry(('a-b-c-d-e-f-g-h', '[xxxxx]', (950,300))) true
>>> validate_cipher('a-b-c-d-e-f-g-h', '[xxxxx]')
False
>>> validate_cifra('a-b-c-d-e-f-g-h', '[abcde]')
True
>>> filter_bdb([])
ValueError: filter_bdb: invalid argument
```

[1] **ATTENTION!!** This function is the same as the one in Section 4.2.1 and only needs to be defined once.

```
>>> bdb = [('aaaaa-bbb-zx-yz-xy', '[abxyz]', (950,300)),
           ('a-b-c-d-e-f-g-h', '[abcde]', (124,325,7)),
          ('entry-most-erased', '[abcde]', (50,404))]
>>> filter_bdb(bdb)
[('entry-most-erased', '[abcde]', (50,404))]
```

4 Data decryption

4.1 Description of the problem

With the BDB filtered and all the wrong entries detected, it's time to unencrypt them to try and find out which users have corrupt passwords.
The information in the BDB is encrypted by an exchange cipher. In order to decrypt the data of each entry, the security number must first be determined. The security number corresponds to the smallest difference between all the numbers contained in the security sequence (last position of the tuple representing each BDB entry). Next, each letter must be changed by moving forward in the alphabet a number of times equal to the security number of each entry plus one for the even positions, and minus one for the odd positions of the text: with security number equal to 2 the letter 'a' becomes 'd' if it is in an even position of the string or 'b' if it is in an 'odd' position; with the same security number the letter 'z' becomes 'c' if it is in an even position or 'a' if it is in an 'odd' position. The dashes become spaces. To determine the even or odd position of a character, the complete string with the first character in the even position ('index 0) is considered. For example, for the following entry (with the wrong control code):

('qgfo-qutdo-s-egoes-wzegsnfmjqz', '[abcde]', (2223,424,1316,99))

the security number is 325 (424 - 99) and the deciphered text 'this cipher is almost unbreakable'.

4.2 Work to be done

The aim of this task is to write a program in Python that allows the BDB entries to be decrypted as described above. To do this, it is necessary to define
the set of functions requested, as well as some additional auxiliary functions, if necessary. Only those functions for which checking the correctness of arguments is explicitly requested should check the validity of arguments.

4.2.1 eh_input: universal → boolean (1,5 values)[2]

This function receives an argument of any type and returns True if and only if its argument matches a (potentially corrupt) BDB entry according to

[2] **ATTENTION!!** This function is the same as the one in Section 3.2.1 and only needs to be defined once.

described, i.e. a tuple with 3 fields: a cipher, a control sequence and a security sequence.

4.2.2 get a security: tuple → integer (1 values)

This function receives a triple of positive integers and returns the security number as described, i.e. the smallest positive difference between any pair of numbers.

4.2.3 decipher text: cad. characters × integer → cad. characters (1 values)

This function receives a string containing a cipher and a security number, and returns the decrypted text as described.

4.2.4 decrypt_bdb: list → list (1 values)

This function receives a list containing one or more entries from the BDB and returns a list of equal size, containing the text of the entries decrypted in the same order. This function must check the validity of its argument, generating a ValueError with the message 'decrypt bdb: invalid argument' if its argument is not valid.

4.3 Example

```
>>> eh_entrada(('qgfo-qutdo-s-egoes-wzegsnfmjqz', '[abcde]', \
                (2223,424,1316,99)))
True
>>> get_security_num((2223,424,1316,99)) 325
>>> decrypt_text('qgfo-qutdo-s-egoes-wzegsnfmjqz', 325) 'this
cipher is almost unbreakable'
> > >  decrypt_bdb([('nothing')])
ValueError: decrypt_bdb: invalid argument
>>> bdb = [('qgfo-qutdo-s-egoes-wzegsnfmjqz', '[abcde]',
              (2223,424,1316,99)), ('lctlgukvzwy-ji-xxwmzgugkgw',
              '[abxyz]', (2388, 367, 5999)), ('nyccjoj-vfrex-ncalml',
              '[xxxxx]', (50, 404))]
>>> decrypt_bdb(bdb)
['this cipher is almost unbreakable', 'programming fundamentals', 'very
  wrong entry']
```

5 Password debugging

5.1 Description of the problem

After detecting the wrong BDB entries and decrypting the with the algorithm developed, it is possible to obtain a list of dictionaries with the username (hame' key), the potentially corrupted password ('pass' key) and the *individual* rule from when that password was set ('rule' key). The rest of the user information contained in the BDB (also potentially corrupted) is not relevant. For example:

```
[ {'name':'john.doe', 'pass':'aabcde', 'rule':{'vals':(1,3), 'char':'a'}},
  {'name':'jane.doe', 'pass':'cdefgh', 'rule':{'vals':(1,3), 'char':'b'}},
{'name':'jack.doe', 'pass':'cccccc', 'rule':{'vals':(2,9), 'char':'c'}} ]
```

According to the documentation provided, there are two sets of rules that must be complied with when defining passwords: the *general rules* and the *individual rules*.
The *general* rules apply to all passwords and are as follows:

- Passwords must contain at least three lowercase vowels ('aeiou'),
- Passwords must contain at least one character that appears twice consecutively.

The *individual* rules are encoded by the value of the 'rule' key of each entry as dictionaries with the keys 'vals' and 'char'. The value of 'valsis a triple of two positive integers corresponding to the smallest (first pos) and largest (second pos) number of times a given small letter (value of the 'char' key) must appear for the password to be valid. For example, the rule *{*'vals': (1,3), 'char':'a'*}* means that the password must contain the letter 'a' at least 1 time and a maximum of 3 times. The value representing the maximum number of times the letter appears is always greater than or equal to the minimum.
In the example above, only the first password is valid. The middle password, 'cdefgh' is not validbecause it doesnt comply with any of the general rules (it doesn't contain 3 vowels, nor two consecutive letters that are the same), nor the individual rule (it doesn't contain instances of 'b', and it needs at least 1). The third password complies with the individual rule (it contains 6 'c's), but only one of the general rules (it doesnt contain 3 vowels). The first password is valid: it contains 2 'a' (within the limits of its individual rule), 3 vowels (2 'a' and 1 'e') and a sequence of two equal characters ('aa').

5.2 Work to be done

The aim of this task is to write a program in Python that allows you to find users with passwords that dont comply with the rules described above. To do this, you should de- finition the set of functions requested, as well as some additional auxiliary functions, if you consider it necessary. Only functions for which checking the correctness of arguments is explicitly requested should check the validity of arguments.

5.2.1 eh_user: universal → boolean (1 value)

This function takes an argument of any type and returns True if and only if its argument matches a dictionary containing the relevant user information from the BDB as described, i.e. name, password and individual rule. Consider for this purpose that names and passwords must have a minimum length of 1 and can contain any character.

5.2.2 eh_valid password: cad. characters × dictionary → boolean (1,5 values)

This function receives a string corresponding to a password and a dictionary containing the individual password creation rule, and returns True if and only if the password complies with all the definition rules (general and individual) as described.

5.2.3 filter passwords: list → list (1 value)

This function receives a list containing one or more dictionaries corresponding to the BDB entries as described above, and returns the list sorted alphabetically with the names of the users with the wrong passwords. This function must check the validity of its argument, generating a ValueError with the message 'filter passwords: invalid argument' if its argument is not valid.

5.3 Example

```
>>> eh_user({'name':'john.doe', 'pass':'aabcde',
                    'rule':{'vals': (1,3), 'char':'a'}})
True
>>> eh_user({'name':'john.doe', 'pass':'aabcde',
                    'rule':{'vals': 1, 'char':'a'}})
False
>>> eh_valid_password('aabcde', {'vals': (1,3), 'char':'a'})
True
>>> eh_valid_password('cdefgh', {'vals': (1,3), 'char':'b'})
False
>>> filter_passwords([])
ValueError: filter_passwords: invalid argument
>>> bdb = [ {'name':'john.doe', 'pass':'aabcde', 'rule':{'vals':(1,3),
              'char':'a'}}, {'name':'jane.doe', 'pass':'cdefgh',
              'rule':{'vals':(1,3), 'char':'b'}}, {'name':'jack.doe',
              'pass':'cccccc', 'rule':{'vals':(2,9), 'char':'c'}} ]
>>> filter_passwords(bdb)
['jack.doe', 'jane.doe']
```

6 Implementation Conditions and Deadlines

- The 1st° project will be submitted exclusively electronically. You must submit your project via the Mooshak system by **17:00 on November 5, 2021**. Projects will not be accepted after this time under any circumstances.
- You must submit a single file with a *.py* extension containing all the code for your project.
- The submission system assumes that the file is encoded in UTF-8. Some editors may use a different encoding, or the use of some stranger characters (notably in the comments) not representable in UTF-8 may lead to another encoding. If all the tests fail, it could be a problem with the encoding used. In this case, you should specify the file's encoding on the first line of the file.[3].
- Submissions that do not pass any of the automatic tests because of minor syntax or coding errors may be corrected by the faculty, incurring a penalty of three points.
- It is not allowed to use any module or function that is not available built-in Python 3.
- There may or may not be an oral discussion of the work and/or a demonstration of how the program works (this will be decided on a case-by-case basis).
- Remember that at Tecnico, academic fraud is taken very seriously and that cheating on an exam (including projects) leads to failure in the subject and possibly disciplinary action. The projects will be submitted to an automatic system of copy detection[4] The chair's faculty will consider The only judge of what whether or not to copy in a project.

7 Submission

The submission and evaluation of the execution of the FP project is done using the Mooshak system.[5]. To obtain the necessary access credentials and be able to use the system you must:

- Obtain the password to access the system, following the instructions on the page: http://acm.tecnico.ulisboa.pt/~fpshak/cgi-bin/getpass-fp21. The password will be sent to the email address you have configured in Fenix. The password may not arrive immediately, please wait.

[3] https://www.python.org/dev/peps/pep-0263/ [4] https://theory.stanford.edu/~aiken/moss

[5] The version of Python used in the automatic tests is Python 3.7.3.

- Once you have received your password by email, you must log in to the system via the page: http://acm.tecnico.ulisboa.pt/~fpshak/. Fill in the fields with the information provided in the email.

- Using the *"Browse..."* button, select the file with the *.py* extension containing all the code for your project. Your *.py* file should contain the implementation of the functions requested in the statement. Then click on the *"Submit"* button to make the submission.
  Wait (20-30 sec) for the system to process your submission!

- When the submission has been processed, you will be able to see the corresponding result in the table. You will receive an execution report in your email with the details of the automatic evaluation of your project and you will be able to see the number of passed/failed tests.

- To exit the system, use the *"Logout"* button.

Submit your project on time, as the following restrictions may not allow you to do so at the last moment:

- You can only make a new submission 5 minutes after the previous one.

- The system only allows 10 simultaneous submissions, so a submission may be rejected if this limit is exceeded. [6].

- There can be no duplicate submissions, i.e. the system can refuse a sub-mission if it is the same as one of the previous ones.

- The **last** submission will be considered for evaluation (even if it has a lower score than previous submissions). You should therefore carefully check that the last submission you made corresponds to the version of the project you want to be evaluated. There are no exceptions.

- Each student is entitled to **15 submissions without penalty** on Mooshak. For each additional submission, 0.1 points will be deducted from the automatic evaluation component.

8 Classification

The project grade will be based on the following aspects:

[6]Note that the limit of 10 simultaneous submissions in the Mooshak system means that if there are a large number of submission attempts before the deadline, some students may not be able to submit by then and therefore be unable to submit the code by the deadline.

1. **Automatic evaluation (80%).** The correct execution will be evaluated using the Mooshak system. The execution time of each test is limited, as is the memory used. There are 132 test cases set up in the system: 28 public tests (available on the course page) worth 0 points each and 104 private tests (not available). As the automatic evaluation is worth 80% (equivalent to 16 points) of the grade, a submission obtains the maximum grade of 1600 points.
   The fact that a project successfully completes the public tests provided does not im- This does not mean that this project is completely correct, as they are not exhaustive. E
   It is the responsibility of each student to ensure that the code produced is in accordance with the specifications of the statement, in order to successfully complete the private tests.

2. **Manual evaluation (20%).** Program style and ease of reading[7]. In particular, the following components will be considered:

   - Good practices (1.5 points): the clarity of the code, the integration of knowledge acquired during the course and the creativity of the proposed solutions will be taken into account.
   - Comments (1 value): should include the signature of the defined functions, comments for the user (*docstring*) and comments for the programmer.
   - Size of functions, code duplication and procedural abstraction (1 value).
   - Choice of names (0.5 points).

9 Recommendations and things to avoid

The following recommendations and aspects are suggestions for avoiding bad work habits (and, consequently, bad grades on the project):

- Read the whole statement, trying to understand the purpose of the various functions asked for. If you have any doubt about your interpretation, use the question time to clarify your questions.

- In the process of developing the project, start by implementing the various functions within each task in the order presented in the statement, following the methodologies studied in the course. As the tasks are independent of each other (with the exception of the common function in Sections 3.2.1 and 4.2.1), they can be solved in any order and independently of each other.

- To check the functionality of your functions, use the examples provided as test cases. Take care to faithfully reproduce the error messages and other *output*, as illustrated in the various examples.

[7] You can find suggestions for good practice at `https://gist.github.com/ruimaranhao/4e18cbe3dad6f68040c32ed6709090a3`

- Don't think that the project can be done in the last few days. If you only start your work in this period you will feel Murphy's Law at work (all problems are harder than they seem; everything takes longer than we think; and if something can go wrong, it will go wrong at the worst possible time).
- Don't duplicate code. If two functions are very similar, it's natural that they can be merged into one, possibly with more arguments.
- Don't forget that excessively large functions are penalized in terms of programming style.
- The "I'm going to run the program now anyway and then worry about style" attitude is totally wrong.
- When the program generates an error, worry about finding out what caused it. Hammering the code has the effect of distorting it more and more.