Laboratory 10:

Select one of these two sets of laboratory questions. Sections 104 and 105 need to take the first set.

Tuples, Sets, List Comprehension, Exceptions

(1) Write a function that creates a random set of *n* elements (*n* being the sole parameter of the function). Each element is a tuple of two coordinates, (x, y), where x is a random number between 0 and 9 and y a random number between 0 and 999.

(2) Write a function that takes a set of tuples as input. The function returns the same set, but tuples with the same first coordinate as a previous tuple in the set are excluded. (Hint: There are many possible ways of doing this, some involving even more advanced data structures. The easiest solution to my mind is to iterate through the set, collect the first coordinates of the tuples encountered in a list or set, check whether the first coordinate of a new tuple to be processed is already there, and only add the new tuple to the result set accordingly.) Since Python does not guarantees the ordering of elements in the set, the result is a bit unpredictable.

For an example, I use the function in (1) to create a set of 50 elements.

```
>>> myset = create_random_set(50)
>>> myset
{(8, 601), (6, 42), (5, 946), (2, 212), (0, 811), (9, 632), (4, 801),
(4, 0), (7, 404), (5, 451), (1, 537), (1, 588), (9, 399), (4, 677),
(8, 223), (6, 665), (8, 968), (5, 559), (5, 47), (1, 540), (9, 562),
(3, 186), (0, 296), (3, 196), (4, 897), (7, 610), (2, 378), (5, 766),
(8, 686), (4, 284), (7, 725), (5, 938), (4, 243), (8, 502), (6, 33),
(5, 845), (7, 345), (0, 458), (0, 169), (0, 246), (9, 141), (1, 662),
(9, 547), (8, 958), (5, 20), (3, 617), (1, 550), (6, 79), (3, 91), (5,
99)}
```

I then run the function from (2).

```
>>> myseta = remove(myset)
>>> myseta
{(8, 601), (7, 404), (1, 537), (6, 42), (5, 946), (2, 212), (0, 811),
(3, 186), (9, 632), (4, 801)}
```

As you can see, only 10 tuples survived the selection.

(3) Write a function that removes duplicate elements from a list by converting first the list to a set and then back to a list.

(4) You can create a set of letters in a word by using the set constructor as in `set('ahmedabad')`, which yields `{'e', 'b', 'd', 'h', 'a', 'm'}`. Write a function that checks whether the letters used in the two words given to it as an argument are the same or not. (Two check the equality of two sets, just use the comparison operator `==` .)

(5) Use list / dictionary comprehension to create the following:

(a) A list of the first 10 square numbers.

(b) A list of all numbers x between 1 and 100 such that $x = 1 \pmod 5$ and $x = 2 \pmod 6$.

(c) A dictionary that associates a square number (between 1 and 10000) and its root, i.e. {1:1, 4:2, 9:3, … }

(6) Use the time module to time the functions that you are developing in this exercise.

Use list comprehension to implement a function that takes a list of arbitrary elements and returns a list of only those elements in the first list that are integers or can be converted into an integer. Your first implementation should try a conversion using int( ), catching all exceptions. Your second implementation should check whether an item is an integer or a floating point by using `isInstance(item, float)` and `isInstance(item, int)`. To check create a list with 10000 integers, 10000 floating point numbers, and 20 strings. Use the `shuffle` from the random module to shuffle the list before testing.

The Hangman Game

Prerequisites:

Lists, Interaction with files, String processing, Loops, Conditional statements.

Overview

The hangman game is a game where a player has to guess a word with a certain number of attempts. Initially, the player is shown a template with one space per letter. In each round, the player guesses a (new) letter. If the letter is in the word, then the spaces where the letter appears are filled in. Otherwise, one of the head, torso, left arm, right arm, left leg, and right leg of a hanged stick-person is drawn hanging from a gallows. Once the last body part of the hanged man is drawn, i.e. after six unsuccessful guesses for a letter in the word, the player has lost. You can find much information for the Hangman Game in the internet, including quite a number of Python code projects, which is why it is no longer used as a class project. In this laboratory, we are going to build the hangman game step by step.

```
+-----+
|     |
|     O
|    /|\
|    / \
|
|
|
|
        You lost
```

Getting a word

Before we can start, we need to have a source for words to be guessed. On the internet, you can find a “English word list” by Lawles that you can download. Our first task is to remove some words that do not fit the hangman game. Our first task is to write a Python script that takes the words in the Lawles file and writes them into another file, called “vocabulary.txt”, unless the word contains a hyphen, a parenthesis, or is too small, i.e. has a length of less than five letters.

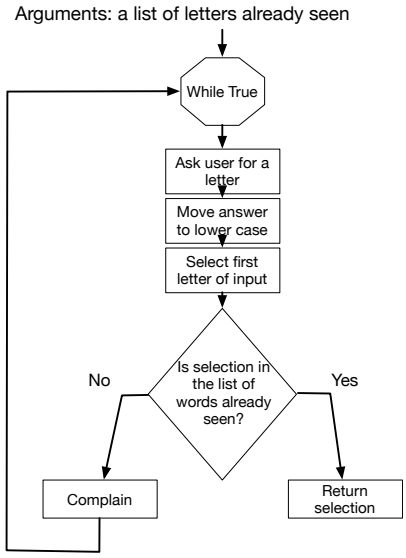
After we created our file, we write a function `def get_random_word():` that randomly selects a word from “vocabulary.txt”. The function opens the file and reads all words line by line into a list. We then use the function `random.randint(0, len(lista)-1)` to randomly

select an index of an element in the list and then print out the word in the list at this index. If we were anticipating playing Hangman incessantly, then we might want to create the list of words once per session and select the word out of the list separately for each round of game.

Drawing the stick figure

Write a function `def draw(errors):` that draws the stick figure (see above right) in dependence on the number of errors. If the number of errors is 0, then there is only the rope, if it is 1, then there is the head, … If the number of errors is larger than 5, then the full figure is displayed with an appropriate message. I create the drawing one line at a time, checking for the number of errors for each line, but you can use a different strategy such as creating different strings for each class of errors.

Getting a letter



Write a function `def get_letter(seen_list):` that uses a list of already seen letters. It uses an infinite loop. In each iteration, the program asks the user to enter a letter, takes the first letter of the input, and converts the letter to lower case. If the letter is not in the list of already seen letters, it accepts the letters and returns it (jumping out of the infinite loop by this). Otherwise, it just complains. The drawing on the left shows the control flow, but frankly, it is more complicated than the Python function.

Checking for Success

If the user enters a letter, three cases arise: First, the letter has already been entered. In this case, the previously described getting-a-letter function handles the prompting for another letter. Second, the letter is new, but not in the word. In the main function, this results in incrementing the number of errors. Third, the letter is new and it is in the word. The main function will then check whether the word has been guessed. To do this, it is easier to write a function `def success(word, seen_list)` that returns True if all of the letters in the argument `word` (containing the word to be guessed) are in the `seen_list`, containing all the letters that the user has guessed so far. The function just passes through the letters in word. If a letter is not in the `seen_list`, then it returns `False`. If however all the letters are in the `seen_list`, then it returns `True`.

Displaying a word

We display the secret word by replacing all letters not yet seen with underscores. Besides the word to be guessed, we need a list of all seen letters. For example, if the secret word is “xenophobia” and the list of guessed letters is “a”, “e”, “i”, “o”, “u”, “n”, then we display

`_eno__o_ia`.

Putting this together: the `play_hangman()` function

Like good software engineers, we have divided the task into small modules. In a bottom-up manner, we now create the main function, called `def play_hangman( )`. The function maintains a state consisting of the list of seen letters, the secret word, and the number of errors. It then enters the main loop. We display the secret word (initially with only underscores) and ask for a new letter. The new-letter function already handles instances of the letter entered by the user having been seen before. There are now two possibilities: First, the new letter is not in the word. In this case, we increment the number of errors and inform the user with a “Sorry”. We also check whether the number of errors is too large, in which case we inform the user, print out the secret word and return from the `play_hangman` function, or alternatively, draw the hangman. Second, the new letter is in the word. We check whether the user has guessed the word. If yes, we congratulate, otherwise, we display the hangman and the secret word (minus the non-guessed letters). Some of this functionality can be embedded in the test for the while loop or you can have an infinite `while True` loop.

Congratulations, you have build your first usable game.