Lab 9

S2 2017

Lab 9

Note: If you do not have time to finish all exercises during the lab time, you should continue working on them later. You can do this at home (if you have a computer with python set up), in the CSIT lab rooms outside teaching hours, or on one of the computers available in the university libraries or other teaching spaces.

If you have any questions about or difficulties with any of the material covered in the course so far, ask your tutor for help during the lab.

Objectives

The purpose of this week's lab is to:

- Explore python's exceptions and exception handling mechanism.
- Define abstract data types and options for concrete data structures implementing them.

Exceptions

Exceptions are python's runtime error mechanism. The `assert` and `raise` statements allow the programmer to generate exceptions when error conditions occur, while the `try` statement allows us to implement an *exception handler* which will be activated if an exception occurs.

When a runtime error occurs, an exception (of a suitable type) is *raised.* If code in which the exception was raised is within the suite of a `try` statement that has an `except` clause for that type of exception, the suite of the `except` clause (that is, the exception handler) is executed; we say the exception has been *caught*, and execution continues with the next statement after the `try` statement. If this is not the case, and the exception was raised in the execution of a funciton, the exception is returned to point of the function call, and the same procedure repeated with the call treated as the origin of the exception. In this way, an exception travels up a call chain until it is either caught by an appropriate handler, or it reaches the top level, in which case the python interpreter stops execution and prints an error message.

Exercise 0

Copy the following two function definitions to a file and run them:

```
def funA(a, i):
    try:
        return a[i] + i
    except TypeError:
        print("Caught TypeError in funA")
    except IndexError:
        print("Caught IndexError in funA")
    return 0

def funB(a, i):
    try:
    i = funA(a, i)
        return a[i]
    except TypeError:
        print("Caught TypeError in funB")
    except IndexError:
        print("Caught IndexError in funB")
    return 0
```

- What happens if you call `funB([3,2,1], 3)`?
- What happens if you call `funB([3,2,1], 0)`?
- Can you find arguments to `funB` that cause the `TypeError` handler in `funA` to be executed?
- Can you find arguments to `funB` that cause the `TypeError` handler in `funB` to be executed?
- Can you find arguments to `funB` that cause an exception to be raised that is not caught by any of the exception handlers?

Exercise 1

(a) Consider the following function (from the lecture on algorithm complexity):

```
def sorted_find(x, slist):
    if slist[-1] <= x:
        return slist[-1]
    lower = 0
    upper = len(slist) - 1
    while (upper - lower) > 1:
        middle = (lower + upper) // 2
        if slist[middle] <= x:
            lower = middle
        else:
            upper = middle
    return slist[lower]
```

The intent of the function is to return the greatest element that is less than or equal to `x`; it assumes that `slist` is a sorted sequence.

- What are all the runtime errors that may be raised in this function?
- Are there any further, unstated assumptions? That is, are there any cases in which the function will not raise an exception, but return an incorrect answer? If so, is there assertion that can be added to the function to ensure it raises an exception instead of failing silently?

(b) Pick a function that you have written as a solution to any exercise from any of the previous labs, and find the answers to the two questions above for that function.

Exercise 2

Recall that python defines ordering of sequences so that a sequence `S1 < S2` is True iff there is an index `i` such that `S1[i] < S2[i]` and `S1[j] == S2[j]` for indices `j` from 0 to `i`. This definition of ordering does not work for sets, because sets are not ordered, and not indexable (that is, we cannot take the i:th element of a set).

Let's define an ordering on sets as follows: set `A` is less than set `B` if for every element `a` in `A` there exists some element `b` in `B` such that `a < b` is True. Recall that values of different types are not always comparable; if `a` and `b` are two values that cannot be compared (for example, if `a` is a string and `b` a number), `a < b` is considered to be False.

Write a function `set_less_than(A, B)`, that takes two sets `A` and `B` and returns True if `A` is less than `B` according to the above definition, and False otherwise.

Examples

- `{ 0.5, 1.5, 2.5 }` is less than `{ 1, 2, 3 }`
- `{ 'a', 'b' }` is less than `{ 'ab', 'bb' }`
- `{ 0, 'a', (0, 'a') }` is less than `{ 1, 'c', (2, -3) }`

Hint: The easiest way to determine if two elements are comparable is to try comparing them. If comparison is not possible, it will raise a `TypeError`.

Excercise 3

The programming problem "Reading CSV files" from Lab 7 asked you to write a program that generates geography trivia questions (about the names of countries, their capital cities, and other things). If you already solved this problem, now is the time to revisit your solution and consider if the design of your program can be improved.

This problem involves representing at least two different types of information: records of information about a country, and questions. A country record must store the relevant pieces of information read from the CSV file, and allow these to be retrieved. A question has three parts: The question text, the list of alternatives (possible answers) and which one of them is correct. To generate one type of question, a procedure like the following can be used:

```
# assuming 'countries' is a list of records of country information,
# the following selects three of them at random:
selected3 = random.sample(countries, 3)
# pick one of them to the right answer
right_answer = random.randint(0, 2)
# create the parts of the question:
question_text = "What is the captial of " + get_country_name(selected3[right_answer]) + "?"
alternatives = [ get_captial_name(country) for country in sel_three ]
return make_question(question_text, alternatives, right_answer)
```

With a small modification, the same approach can be used to generate other types of questions based on the country data.

- Implement the functions necessary to create country records and to access the information stored in them. Write (or modify) the part of the program that reads the CSV file and creates a list of country information records so that it uses only the interface function to create a record.
- Implement a function that takes as argument a question, presents the question to the user, gets an answer, and either confirms it as correct or shows the correct answer. This function should work for any kind of question that has the same three parts (question text, list of alternatives, and right answer).

  The function should also handle any possible errors in the interaction with the user. For example, if the user's answer is not (the number of) one of the alternatives, she or he should be offered the chance to enter an answer again, instead of having it counted as an incorrect answer. The user must also be given an option to quit the trivia program.

Excercise 4

In two previous lectures, we looked at a recursive algorithm for solving the Sudoku puzzle, and some parts of the implementation of an abstract data type "grid" representing the state of such a puzzle.

Here is the skeleton code that was used in the lecture.

(a) Complete the implementation of the grid abstract data type, so that the solving algorithm works.

(b) In the lecture, we sketched an implementation using a list-of-lists to store the contents of the grid. Can you think of any other data structure that could be used to implement the grid abstract data type? Can you change your implementation to use a different data structure, without changing any part of the code that uses its interface?