FIT1045/53 Algorithms and programming in Python, S1-2019
Assignment 2 (value 18%).
Due: Friday 17th May, 2019, 11:55 pm

Objectives

The **objectives of this assignment** are:

- To demonstrate the ability to implement algorithms using basic data structures and operations on them.
- To gain experience in designing an algorithm for a given problem description and implementing that algorithm in Python.

Submission Procedure

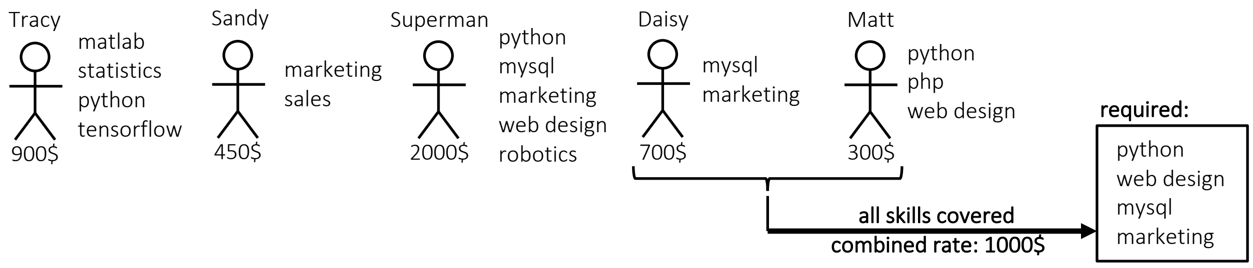
1. Put you name and student ID on each page of your solution.
2. Save your files into a zip file called yourFirstName_yourLastName.zip
3. Submit your zip file containing your solution to Moodle.
4. Your assignment will not be accepted unless it is a readable zip file.

Important Note: Please ensure that you have read and understood the university's policies on plagiarism and collusion available at `http://www.monash.edu.au/students/policies/academic-integrity.html`. You will be required to agree to these policies when you submit your assignment.

Marks: This assignment will be marked both by the correctness of your code and by an interview with your lab demonstrator, to assess your understanding. This means that although the quality of your code (commenting, decomposition, good variable names etc.) will not be marked directly, it will help to write clean code so that it is easier for you to understand and explain.
FIT1045 Students: This assignment has a total of 30 marks and contributes to 18% of your final mark.
FIT1053 Students: This assignment has a total of 35 marks and contributes to 18% of your final mark. There is an extra task at the end for you to complete.
Late submission will have 10% off the total assignment marks per day (including weekends) deducted from your assignment mark. (In the case of Assignment 1, this means that a late assignment will lose 3 marks for each day (including weekends)). Assignments submitted 7 days after the due date will normally not be accepted.
Detailed marking guides can be found at the end of each task. Marks are subtracted when you are unable to explain your code via a code walk-through in the assessment interview. **Readable code is the basis of a convincing code walk-through.**

Task 1: Talent Acquisition (15 Marks)

Background
In this task we investigate algorithmic solutions to what is called a *constrained optimisation problem*. Specifically, we look into the problem of automatically composing a team to work on a given project. We want to make sure that between the team members, we have people who are capable of doing all the work that the project requires (constraint), but we also want to keep our total costs as low as possible in terms of the combined daily rate charged by all team members (optimisation). For instance, a problem input might look like this:



More formally, the problem can be stated as: Given a set of required skills and a set of possible candidates, each of which has a set of skills they posses and a daily rate, find a set of candidates such that

1. Every skill required for the project is possessed by at least one of the candidates in the set.
2. The total daily rate of all candidates in the set is minimum, i.e., there is no set satisfying Condition 1 with a smaller total rate.

As for all computational problems, there are various strategies to tackle it. In this task we will look at two approaches: one which tries to find a reasonably cheap team quickly (relaxing Condition 2 above), and one which takes more time but is guaranteed to compute the best possible team.

Instructions
Create a Python module called `hiring.py`. Your module must contain the *six* functions described in the subtasks below, but you are allowed, and in fact encouraged, to implement additional functions as you deem appropriate. The module must not contain any imports. Throughout this task we will adhere to the following conventions:

- We will represent a skill simply by a string (e.g., `"java"`, `"lua"`, or `"marketing"`) and a set of skills will be given by a list of strings.
- A candidate will be represented by a pair `(skills, rate)` consisting of a list of skills `skills` and a positive integer `rate` representing their daily rate.
- We will assume and ensure that lists of skills do not contain duplicates and that each required skill for the input project is at least possessed by one of the available candidates.

a) Write functions `cost(candidates)`, `skills(candidates)` and `uncovered(project, skills)` for working with the basic ingredients of our constrained optimization problem as follows. The function `cost(candidates)` takes as input a list of candidates and produces as output the combined daily rates of all the given candidates. The function `skills(candidates)` takes as input a list of candidates and produces as output the list of skills possessed by at at least one of the candidates (again, the output list should not have any duplicates). The function `uncovered(project, skills)` takes as input a list of required skills `project` and a list of provided skills `skills` and produces as output a new list of skills that contains all skills in `project` not contained in `skills`.

b) Write a function `team_of_best_individuals(skills_required, candidates)` that solves our problem *approximately* by iteratively finding the best next candidate evaluated in isolation, i.e., by only considering the number of relevant skills covered per dollar daily rate—without taking into account what it will cost to complete the team around that candidate. To represent this evaluation metric, write another function `best_individual_candidate(project, candidates)` that accepts as input a list of required skills `skills` and a list of candidates `candidates` and that returns as output the index of the candidate with the maximum number of skills covered per dollar daily rate. If there is a tie, return the earliest candidate involved in the tie. Based on that evaluation metric, the function `team_of_best_individuals(project, candidates)` has

Input: A list of strings `project` representing the required skills and a list of available candidates `candidates`.

Output: A list of candidates `team=[c1, c2, c3, ..., ck]` taken from the input candidates such that

- For all the skills in `project` there is at least one candidate in `team` that has that skill.
- Candidate `c1` is the best individual candidate for the required skills, `c2` is the best individual candidate for the skills required that are not covered by candidate `c1` and so on.
- Every candidate possesses at least one skill relevant to the project that is not covered by all previous candidates in the list.

c) Write a function `best_team(project, candidates)` that solves our optimization problem optimally. That is, function `best_team(skills_required, candidates)` has

Input: A list of strings `project` representing the required skills and a list of available candidates `candidates`.

Output: A list of candidates `team` taken from the input candidates such that

- For all the skills in `project` there is at least one candidate in `team` that has that skill.
- The total daily rate `cost(team)` is less or equal to to all other possible sets of candidates from `candidates` which satisfy the first property.

Hint: Think about how you can relate the problem of finding the best team for a project to *itself*. Related to that, can you come up with a criterion to determine whether an individual candidate is part of the best team?

Examples Assume we have the following candidates and required skills for a project:

```
jess = (["php", "java"], 200)
clark = (["php", "c++", "go"], 1000)
john = (["lua"], 500)
cindy = (["php", "go", "word"], 240)

candidates = [jess, clark, john, cindy]
project = ["php", "java", "c++", "lua", "go"]
```

a) Calling `cost([john, cindy])` returns `740`, the total daily rate of `john` and `cindy`.

b) Calling `skills([clark, cindy])` returns a permutation of the list `["php", "c++", "go", "word"]` because these are the skills covered by at least one of `clark` and `cindy`.

c) Calling `uncovered(project, skills([clark]))` returns `["java", "lua"]` because these are the skills not covered by `clark`.

d) Calling `best_individual_candidate(project, candidates)` would return `0` because `jess` covers 2 required skills for a daily rate of \$200 or $1/100$ useful skills per dollar. Thus, `jess` covers more skills per dollar than `clark` ($3/1000$), `john` ($1/500$), or `cindy` ($1/120$).

e) Calling `team_of_best_individuals(project, candidates)` returns a list equal to `[jess, cindy, john, clark]` because, as we know from Example d above, `best_individual(project, candidates)==0` and then `best_individual(uncovered(project, skills([jess])), [clark, john, cindy])==2` and so on.

f) Calling `best_team(project, candidates)` returns a list `team` equal to some permutation (same elements but possibly different order) of `[jess, clark, john]` because `uncovered(project, skills(team))=[]` and `cost(team)=1700`, which is less than the cost of all other feasible teams (i.e., those that cover all skills).

Marking Guide (total 15 marks)
Marks are given for the correct behaviour of the different functions:

a) 1 mark for `cost` and 2 marks for each of `skills` and `uncovered`

b) 2 marks for `best_individual_candidate` and 2 marks for `team_of_best_individuals`

c) 6 marks for `best_team`

All functions are assessed independently to the degree possible. For instance, even if function `skills` does not always produce the correct output, function `team_of_best_individuals` can still be marked as correct.

Task 2: Calculator (15 Marks)

Background
In this task we explore a simple *parsing* problem. "Parsing" refers to the task of correctly interpreting structured information that is given in a flat unstructured form such as a string. A simple example of this is the evaluation of arithmetic expressions. Arithmetic expressions involving the operations of addition (+), subtraction (−), multiplication (∗), division (/), and exponentiation (^) are normally written in "infix" notation, i.e., with the operation symbol in-between the two operands that it is applied to. This leads to the problem that an expression could principally be interpreted in multiple ways and we have to decide which operator to give *precedence* to. For example, according to the standard arithmetic rules, we have

$$10 - 5 * 4\hat{}2 + 100/4 = -45$$

because ^ has a higher precedence than ∗ and / which in turn have a higher precedence than + and −. These standard precedence-based rules can be overridden by parentheses. For instance, we have

$$((10 - 5) * 4\hat{}2 + 100)/4 = 45$$

because expressions inside parentheses are evaluated first in a recursive manner before standard operator precedence rules are applied.
In this task, we will implement these rules in Python to create a calculator that can evaluate well-formed infix expressions given as a string that contains *non-negative* floating-point numbers (e.g, `"0.0"`, `"92"`, `"7.5"` or `"943.2543"`), operators `"+"`, `"-"`, `"*"`, `"/"`, `"^"`, parentheses `"("` and `")"`, and whitespaces `" "`. We evaluate the operators in the typical order outlined above (and in addition from left to right in case of equal operator precedence, which is relevant for the non-associative operator ^).

Instructions
Create a python module `parsing.py`. Within that module create the five functions described in the subtasks below. The only import statement in the module must be `from math import pow`.

a) Write a function `tokenization(expr)` that maps an arithmetic expression to its "tokens", i.e., the individual syntactic units it contains. This function takes as input a string representing a mathematical expression consisting of non-negative numbers and the symbols listed in the background including potentially spaces. It returns a list of **tokens** corresponding to the given expression. A **token** can either be a *string* corresponding to an operator from the set {`"+"`, `"-"`, `"*"`, `"/"`, `"^"`}, a *string* containing a single opening or closing parenthesis ({`"("`, `")"`}), or a non-negative *float*. Whitspace from the input string do not appear among the tokens.

b) Write functions `has_precedence(op1, op2)` and `simple_evaluation(tokens)` that together can evaluate simple arithmetic expression *without* parentheses.

- Function `has_precedence(op1, op2)` takes as input two operator tokens, i.e., strings from the set {`"+"`, `"-"`, `"*"`, `"/"`, `"^"`} and outputs `True` if `op1` has higher precedence than `op2`; otherwise `False`.
- Function `simple_evaluation(tokens)` takes as input a list of tokens (excluding parentheses) and returns the single floating point number corresponding to the result of the tokenized arithmetic expression.

c) Write functions `complex_evaluation(tokens)` and `evaluation(string)` that put everything together and allow to evaluate strings representing well-formed arithmetic expressions. As an intermediate step, the function `complex_evaluation(tokens)` takes as input a list of tokens (this time *including parentheses*) and returns the single floating point number corresponding to the result of the tokenized arithmetic expression. Finally, the function `evaluation(string)` has as input a string containing a well-formed arithmetic expression and as output the single float corresponding to its result.

Example:

a) Calling `tokenization("(3.1 + 6*2^2) * (2 - 1)")` would return the list `["(", 3.1, "+", 6.0, "*", 2.0, "^", 2.0, ")", "*", "(", 2.0, "-", 1.0, ")"]`. Note that the symbols are strings while the numbers are floats.

b) Calling `has_precedence("*","+")` and `has_precedence("^","+")` both return `True`. In contrast, calling `has_precedence("*", "^")` as well as `has_precedence("*","/")` both return `False`.

c) Calling `simple_evaluation([2, "+", 3, "*", 4, "^", 2, "+", 1])` would return 51.0. This is because we first evaluate '^', giving $2+3*16+1$, then we evaluate "*", giving $2+48+1$, and lastly we evaluate the two "+" left to right giving 51. Returned as a float, this is 51.0.

d) Calling `complex_evaluation(["(", 2, "-", 7, ")", "*", 4, "^", "(", 2, "+", 1, ")"]` as well as `evaluation("(2-7) * 4^(2+1)")` both return -320.0. This is because we first evaluate the terms in parentheses, giving $-5*4\hat{}3$. Then we evaluate the ^, giving $-5*64$. Evaluating the '*' gives -320.0.

Marking Guide (total 15 marks)
Marks are given for the correct behaviour of the different functions:

a) 3 marks for `tokenization`

b) 5 marks for `simple_evaluation` and 1 mark for `has_precedence()`

c) 5 marks for `complex_evaluation` and 1 mark for `evaluation`

All functions are assessed independently to the degree possible. For instance, even if function `simple_evaluation` does not always produce the correct output, function `complex_evaluation` can still be marked as correct.

Task 3: FIT1053 Students Only (5 Marks)

In addition to the above work, you are required to complete one of the following two tasks:

1. Argue the correctness of the function `simple_evaluation` from Task 2. To do this, annotate loop invariants as comments in your code and provide a complete argument in a block comment at the beginning of your function. Your complete argument should refer to invariants you identify in your code.

   Marking Guide (total 5 marks)

   a) 2.5 marks for correctly identifying appropriate invariants within code

   b) 2.5 marks for using invariants to formulate a proof of correctness

   or

2. The aim of this task is to test your `best_team` function from Task 1 and your `complex_evaluation` function from Task 2. To start, create a third module `test_modules.py`. In this module, import your `best_team` function from your hiring.py module and your `complex_evaluation` function from your `parsing.py` module. If you have followed the correct naming requirements, this can be done by having all three of your modules in the same folder and placing the following two lines of code at the start of your `test_modules.py` module:

   ```
   from hiring import best_team
   from parsing import complex_evaluations
   ```

   You will now be able to use these functions from within your new module. You now need to write the function `test(func, input, output)` to be used to test an input function, `func`. Here, `input` is a valid problem input for the function being tested and `output` is the output we would expect from the function. If a test fails, appropriate information should be displayed to the user, such as what function was called, what the input was, what output the function produced, and the output that was expected.

   Provide at least 4 test cases (input and expected output) for each of your `best_team` and `complex_evaluation` functions. It is expected that these test cases cover a board range of problems of various difficulty. For example, the test cases for your `complex_evaluation` function could start by testing each of the operations separately then build up to testing expressions that include a mixture of operations and parentheses.

   Marking Guide (total 5 marks)

   a) 2 marks for correct implementation of `test`

   b) 1.5 marks for providing at least 4 test cases for your `best_team` function of various difficulty

   c) 1.5 marks for providing at least 4 test cases for your `complex_evaluation` function of various difficulty