FIT1045 Algorithms and programming in Python, S1-2020
Programming Assignment (rev. 03/04/20)

Assessment value: 22% (10% for Part 1 + 12% for Part 2)

Due: Week 6 (Part 1), Week 11 (Part 2)

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.
- To explain the computational problems and the challenges they pose as well as your chosen strategies and programming techniques to address them.

Submission Procedure

You are going to create a single Python module file `products.py` based on the provided template. Submit a first version of this file at the due date of Part 1, Friday midnight of Week 6. Submit a second version of this file at the due date of Part 2, Friday midnight of Week 11.

For each of the two versions:

1. Save the current version of your `products.py` into a zip file called `yourStudentID_yourFirstName_yourLastName.zip`.
2. Submit this zip file to Moodle.
3. Your assignment will not be accepted unless it is a readable zip file containing a file called `products.py`.

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.
A common mistake students make is to use Google to find solutions to the questions. Once you have seen a solution it is often difficult to come up with your own version. The best way to avoid making this mistake is to avoid using Google. You have been given all the tools you need in workshops. If you find you are stuck, feel free to ask for assistance on Moodle, ensuring that you do not post your code.

Marks and Module Documentation

This assignment will be marked both by the correctness of your module and your analysis of it, which has to be provided as part of your function documentations. Each of the assessed function of your module must contain a docstring that contains in this order:

1. A one line short summary of what the function is computing
2. A formal input/output specification of the function
3. Some usage examples of the function (in doctest style)
4. A paragraph explaining the challenges of the problem that the function solves and your overall approach to address these challenges
5. A paragraph explaining the specific choices of programming techniques that you have used

6. [only for Part 2] A paragraph analysing the computational complexity of the function
7. [only for FIT1053 for Part 2]: A paragraph discussing the correctness of the function, giving either a theoretical argument (potentially using loop invariants) or a discussion of how the given doctest example systematically cover a certain range of possible inputs and special cases.

For example, if `scaled(row, alpha)` from Lecture 6 was an assessed function (in Part 2 and you are a FIT1045 student) its documentation in the submitted module could look as follows:

```
def scaled(row, alpha):
    """ Provides a scaled version of a list with numeric elements.

    Input : list with numeric entries (row), scaling factor (alpha)
    Output: new list (res) of same length with res[i]==row[i]*alpha

    For example:
    >>> scaled([1, 4, -1], 2.5)
    [2.5, 10.0, -2.5]
    >>> scaled([], -23)
    []

    This is a list processing problem where an arithmetic operation has to be carried out
    for each  element of an input list of unknown size.  This means that a loop is required
    to iterate over all elements and compute their scaled version. Importantly, according
    the specification, the function has to provide a new list instead of mutating the input
    list. Thus, the scaled elements have to be accumulated in a new list.

    In my implementation, I chose to iterate over the input list with a for-loop and to
    accumulate the computed scaled entries with an augmented assignment statement
    (+=) in order to avoid the re-creation of each intermediate list. This solution allows a
    concise and problem-related code with no need to explicitly handle list indices.

    The computational complexity of the function is O(n) for an input list of length n. This
    is because the for loop is executed n times, and in each iteration there is constant
    time operation of computing the scaled element and a constant time operation of
    appending it to the result list (due to the usage of the augmented assignment
    statement instead of a regular assignment statement with list concatenation).
    """
    res = []
    for x in row:
        res += [alpha*x]
    return res
```

Beyond the assessed functions, you are highly encouraged to add additional functions to the module that you use as part of your implementation of the assessed functions. In fact, such a decomposition is essential for a readable implementation of the assessed function, which in turn forms the basis of a coherent and readable analysis in the documentation. All these additional functions also must contain a minimal docstring with the items 1-3 from the list above. Additionally, you can also use their docstrings to provide additional information, which you can refer to from the analysis paragraphs of the assessed functions.

This assignment has a total of 22 marks and contributes to 22% of your final mark. For each day an assignment is late, the maximum achievable mark is reduced by 10% of the total. For example, if the assignment is late by 3 days (including weekends), the highest achievable mark is 70% of 15, which is 10.5. Assignments submitted 7 days after the due date will normally not be accepted.

Mark breakdowns for each of the assessed functions can be found in parentheses after each task section below. Marks are subtracted for inappropriate or incomplete discussion of the function in their docstring. **Again, readable code with appropriate decomposition and variable names is the basis of a suitable analysis in the documentation.**

Overview

In this assignment, we implement the backend of an intelligent product search engine where potential customers can search a database of products characterised by different features taking into account the customer's preferences. Products are represented as lists of feature values, and, correspondingly, a product database is a table where each row represents an individual product.

For example, consider the following database of phones using the features of `'name'`, `'manufacturer'`, `'size'`, `'battery'`, `'price'` (*Source* (`https://www.allphones.com.au/`); note that this database is just for the purpose of illustration and your module must work for other database as well which may use different and a different number of features.).

```
>>> phones = [['iPhone11', 'Apple', 6.1, 3110, 1280],
...           ['Galaxy S20', 'Samsung', 6.2, 4000, 1348],
...           ['Nova 5T', 'Huawei', 6.26, 3750, 497],
...           ['V40 ThinQ', 'LG', 6.4, 3300, 598],
...           ['Reno Z', 'Oppo', 6.4, 4035, 397]]
```

Part 1: Selection and Ranking (10%, due in Week 6)

Task A: Product selection (`satisfies`, 3 Marks; `selection`, 3 Marks)

The first functionality to implement is product selection based on one or more hard criteria. For that we will represent conditions in Python as triples `(i, c, v)` where `i` is an integer feature index, `c` is a relation symbol in `['<', '<=' '==', '>=', '>', '!=']`, and `v` is some feature value. For example, for the phone database above the following conditions represent that a phone is inexpensive, has a large screen, and it is manufactured by Apple, respectively.

```
>>> inexpensive = (4, '<=', 1000)
>>> large_screen = (2, '>=', 6.3)
>>> apple_product = (1, '==', 'Apple')
```

For this task, your module has to contain at least the two mandatory functions `satisfies` and `selection`. The first determines whether a product satisfies an individual condition. The second selects all products from an input table that satisfies all of a given list of conditions. The details are as follows.

Implement a function `satisfies(product, cond)` with the following specification.

Input: A product feature list `product` and a condition `cond` as specified above.

Output: `True` if condition holds for the product otherwise `False`.

For example, for the above phone database and conditions, `satisfies` behaves as follows.

```
>>> satisfies(['Nova 5T', 'Huawei', 6.26, 3750, 497], inexpensive)
True
>>> satisfies(['iPhone11', 'Apple', 6.1, 3110, 1280], inexpensive)
False
>>> satisfies(['iPhone11', 'Apple', 6.1, 3110, 1280], large_screen)
False
>>> satisfies(['iPhone11', 'Apple', 6.1, 3110, 1280], apple_product)
True
```

Implement a function `selection(products, conditions)` with the following specification.

Input: A product table `products` and a list of conditions `conditions` as specified above.

Output: The list of products satisfying all condition `cond in conditions`.

Building again on the above scenario, the behaviour of this function can be illustrated by the following examples:

```
>>> cheap = (4, '<=', 400)
>>> selection(phones, [cheap, large_screen])
[['Reno Z', 'Oppo', 6.4, 4035, 397]]
>>> not_apple = (1, '!=', 'Apple')
>>> selection(phones, [not_apple])
[['Galaxy S20', 'Samsung', 6.2, 4000, 1348], ['Nova 5T', 'Huawei', 6.26, 3750, 497],
['V40 ThinQ', 'LG', 6.4, 3300, 598], ['Reno Z', 'Oppo', 6.4, 4035, 397]]
```

Task B: Product Ranking (`linearly_ranked`, 4 Marks)

In addition to selecting products based on hard conditions, users should also be allowed to indicate their preferences by numeric weights for individual (numerical) product features. We represent a set of `weights` as a list, which contains at position `j` the weight for feature `j` or `None` if the user does not have a preference weight for that feature (or if that feature is not numeric).
For example, the weighting of a user looking for a cheap phone (every $100 worth one negative point) with small screen (every 0.1 inch worth one positive point) and long battery life (every 1000mAh worth one positive point) is represented by the following list:

```
>>> screen_battery_price = [None, None, 10, 1/1000, -1/100]
```

Implement a function `linearly_ranked(products, weights)` that accepts as input a product database and preference weights and returns as output a list of products ordered according to a numeric score resulting from the linear combination of all specified individual preferences. The detailed specification of this function is as follows.

Input: Product table `products`, list of weights `weights` of length equal to the number of columns in `products` containing at position `i` the linear weight for product feature column `i` or `None` if no weight is specified for column `i` (must be `None` for non-numeric column). At least one weight must be different from `None`.

Output: A new table containing all products from the input table in decreasing order of the linear score given, for a product `prod in products`, by the sum of `prod[i]*weights[i]` over all column indices `i` with `weights[i]!=None`.

For example with the phone database from the overview section and a simple weighting scheme putting all weight on *battery capacity* we get:

```
>>> battery = [None, None, None, 1, None]
>>> linearly_ranked(phones, battery)
[['Reno Z', 'Oppo', 6.4, 4035, 397], ['Galaxy S20', 'Samsung', 6.2, 4000, 1348],
['Nova 5T', 'Huawei', 6.26, 3750, 497], ['V40 ThinQ', 'LG', 6.4, 3300, 598],
['iPhone11', 'Apple', 6.1, 3110, 1280]]
```

In contrast, with the above compromise between *screen size*, *battery*, and *price* we get:

```
>>> linearly_ranked(phones, screen_battery_price)
[['Reno Z', 'Oppo', 6.4, 4035, 397], ['Nova 5T', 'Huawei', 6.26, 3750, 497],
['V40 ThinQ', 'LG', 6.4, 3300, 598], ['Galaxy S20', 'Samsung', 6.2, 4000, 1348],
['iPhone11', 'Apple', 6.1, 3110, 1280]]
```

The input table is not changed by the function. That is after calling it, we still have:

```
 >>> phones
[['iPhone11', 'Apple', 6.1, 3110, 1280], ['Galaxy S20', 'Samsung', 6.2, 4000, 1348],
['Nova 5T', 'Huawei', 6.26, 3750, 497], ['V40 ThinQ', 'LG', 6.4, 3300, 598],
['Reno Z', 'Oppo', 6.4, 4035, 397]]
```

Part 2: Product recommendations (12%, due in Week 11)

Task A: Relevance filtering (`relevant`, 6 Marks)

The first advanced functionality to be included in your module is the ability to filter our irrelevant products from a product list based on specified *directional preferences* for individual numeric feature dimensions. That is, users can specify for numeric features, whether they prefer large values of this feature or small values.

For that, an individual directional preference is represented as a pair `(i, p)` where `i` refers to a numeric dimension or index position of a product feature list and `p` is either $+1$ or -1 depending on whether the user considers feature `i` to be positive or negative.

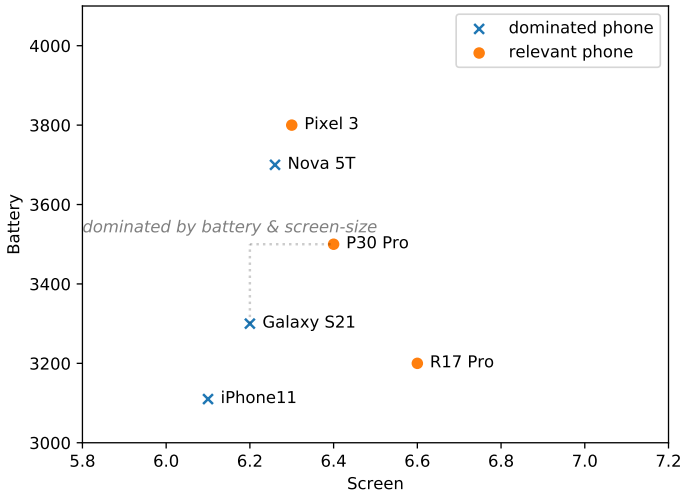

Figure 1: Product relevance based on user preference

Based on a list of such preference, we define the notion of one product *dominating* another as follows:

a) `prod1` dominates `prod2` with respect to preference `(i,p)` if `prod1[i] > prod2[i]` in case `p=1` (or `prod1[i] < prod2[i]` in case `p=-1`)

b) `prod1` dominates `prod2` if `prod1` dominates `prod2` with respect to at least one specified preference and not `prod2` does not dominate `prod1` with respect to any specified preference

Given a list if products, we consider an individual product *relevant* if it is not dominated by another product in the list.

Implement a function `relevant(products, preferences)` with the following specification.

Input: a list of products `products` and a list of preferences `preferences` as specified above

Output: the list of products that are relevant, i.e., not dominated by another product on the list

Examples

```
>>> phones = [['iPhone11', 'Apple', 6.1, 3110, 1280],
...           ['Galaxy S21', 'Samsung', 6.2, 3300, 1348],
...           ['Nova 5T', 'Huawei', 6.26, 3700, 497],
...           ['P30 Pro', 'Huawei', 6.4, 3500, 398],
...           ['R17 Pro', 'Oppo', 6.6, 3200, 457],
...           ['Pixel 3', 'Google', 6.3, 3800, 688]]


>>> large_battery_but_cheap = [(3,1), (4,-1)]
>>> relevant(phones, large_battery_but_cheap)
[['Nova 5T', 'Huawei', 6.26, 3700, 497], ['P30 Pro', 'Huawei', 6.4, 3500, 398],
['Pixel 3', 'Google', 6.3, 3800, 688]]
```

```
>>> big_screen = [(2,1)]
>>> relevant(phones, big_screen)
[['R17 Pro', 'Oppo', 6.6, 3200, 457]]

>>> big_screen_but_cheap = [(2,1),(4,-1)]
>>> relevant(phones, big_screen_but_cheap)
[['P30 Pro', 'Huawei', 6.4, 3500, 398], ['R17 Pro', 'Oppo', 6.6, 3200, 457]]
```

See Figure 1 for an illustration of the preference `big_screen_large_battery = [(2,1),(3,1)]` which returns three phones. 'P30 Pro' dominates every other phone either in terms of battery or screen size, 'Pixel 3' dominates in terms of battery and 'R17 Pro' dominates in terms of screen size. Consequently, we end up with the following behaviour.

```
>>> big_screen_large_battery = [(2,1),(3,1)]
>>> relevant(phones, big_screen_large_battery)
[['P30 Pro', 'Huawei', 6.4, 3500, 398], ['R17 Pro', 'Oppo', 6.6, 3200, 457],
['Pixel 3', 'Google', 6.3, 3800, 688]]
```

Task B: Product recommendation (`inferred_conditions`, 6 Marks)

Finally, we will implement a basic machine learning functionality: based on simple binary user feedback on individual products ("like"/"dislike"), our search engine should be able to automatically categorise other products as either interesting or uninteresting for the the same user.

Technically, **implement a function** `inferred_conditions(pos_ex, neg_ex)` that accepts as input two product tables based on the same feature columns, one containing *positive examples*, i.e., products that the user likes and the other one containing *negative examples*, i.e., products that the user dislikes. Your function should return as output a list of conditions `conditions` *on the given numerical features* that can be used in conjunction with the function `selection` from Part 1 to create a personalised recommendation for the user when applied to a table of new products.

In particular, the returned conditions should be consistent with the provided list of positive examples (not deselect any of the products known to be liked by the user) but exclude as many of the known negative examples as possible. In summary, the specification of the function `inferred_conditions` is as follows:

Input: a list of products `pos_ex` of positive product examples and a list of products `neg_ex` of negative product examples, both based on the same feature columns

Output: a list of conditions `conds` on the numeric feature columns (i.e., those that don't contain strings) that follow the same specification as used in Part 1 of the assignment and that satisfy the following to criteria:

1. `selection(pos_examples, conds) == pos_examples`, i.e., the inferred conditions select all positive examples, and
2. `len(selection(neg_examples, conds))` is minimal among all condition sets that satisfy the first criterion.

For instance, in our exemplary application to phones, we could imagine the following tables for positive and negative examples.

```
>>> pos_ex = [['iPhone11', 'Apple', 6.1, 3110, 1280],
...           ['Nova 5T', 'Huawei', 6.26, 3750, 497],
...           ['V40 ThinQ', 'LG', 6.4, 3500, 800]]
>>> neg_ex = [['Galaxy S20', 'Samsung', 6.46, 3000, 1348],
...           ['V40 ThinQ', 'LG', 5.8, 3100, 598],
...           ['7T', 'OnePlus', 6.3, 3300, 1200]]
```

Another table of new phones could be as follows.

```
>>> new_phones = [['Galaxy S9', 'Samsung', 5.8, 3000, 728],
...               ['Galaxy Note 9', 'Samsung', 6.3, 3600, 700],
...               ['A9 2020', 'Oppo', 6.4, 4000, 355]]
```

See Figure 2 for an illustration of the content of these three tables mapped onto the two feature dimensions of "Screen size" and "Battery capacity". Our function allows us to infer conditions that can be used with the function `selection` to recommend new phones.

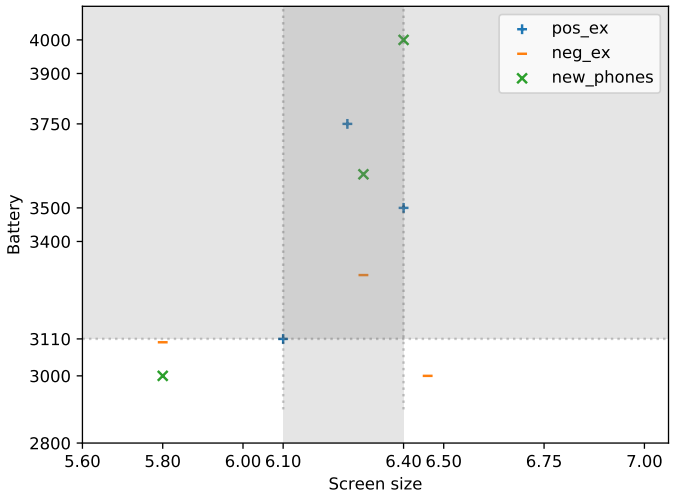

Figure 2: Example phone tables mapped onto the feature dimensions 2 ("Screen size") and 3 ("Battery capacity") with conditions `(2, '>=', 6.1)`, `(2, '<=', 6.4)`, and `(4, '>=', 3110)`. These conditions select all positive examples and only one negative example, which cannot be separated from the three positive examples (in these two dimensions).

```
>>> conds = inferred_conditions(pos_ex, neg_ex)
>>> selection(pos_ex, conds)
[['iPhone11', 'Apple', 6.1, 3110, 1280], ['Nova 5T', 'Huawei', 6.26, 3750, 497],
['V40 ThinQ', 'LG', 6.4, 3500, 800]]
>>> selection(neg_ex, conds)
[['7T', 'OnePlus', 6.3, 3300, 1200]]
```

The inferred conditions can then be used to recommend a specific subset of the new phones to the user.

```
>>> selection(new_phones, conds)
[['Galaxy Note 9', 'Samsung', 6.3, 3600, 700], ['A9 2020', 'Oppo', 6.4, 4000, 355]]
```