The deadline for this exercise sheet is **Monday, 30.04.2018, 8:00.**

1 Vector Math

We can model vectors with tuples and lists. Python however is not equipped with the tools to do vector math, but we can write the functions for this ourselves!
Write a script `vectors.py` which defines the following functions:

- `add(x, y)`: Adds x and y such that the result follows $z_i = x_i + y_i$. **Solution:**

```
def add(x, y):
        """Adds up the vectors x and y"""
        return [x[i] + y[i] for i in range(len(x))]
```

- `sub(x, y)`: Subtracts y from x such that the result follows $z_i = x_i - y_i$. **Solution:**

```
def sub(x, y):
    """returns the result of subtracting y from x"""
        return [x[i] - y[i] for i in range(len(x))]
```

- `dot(x, y)`: Calculates the scalar (dot) product (or inner product) of x and y. The dot product $<x, y>$ is defined as $<x, y> = \sum_{i=1}^{N} x_i y_i$. **Solution:**

```
def dot(x, y):
    """Returns the inner product of the two vectors"""
        sum([x[i]y[i] for i in range(len(x))])
```

- `angle(x, y)`: Calculates the angle θ between the between x and y. The angle can be found by using an alternative definition of the dot product: $<x, y> = \|x\|\|y\| \cos\theta$, which we can then solve for θ and we get $\theta = \arccos \frac{<x, y>}{\|x\|\|y\|}$.
  Note: $\|x\|$ where x is a vector is called the vector norm and is calculated by $\sqrt{< x, x >}$. You can find the arccos function in the `math` package, which we previously used for pi. The function is called `acos`. **Solution:**

```
import math


def angle(x, y):
    """Returns the angle between the two vectors in radien"""
    divisor = math.sqrt(dot(x, x)) * math.sqrt(dot(y, y))
    return math.acos(dot(x, y) / divisor)
```

- `pdist(x, y, **kwargs)`: The distance between the two points x and y. Your *kwargs* should recognise the keywords `metric` and `p`. Your `metric` keyword should be able to take one of the values `'euclidean'`, `'minkowski'`, `'cityblock'`, and `p` can take any integer greater or equal than one. The distance is calculated depending on the metric and should default to `euclidean`. The distances are calculated as follows:
  - `euclidean`: $d_{euclid}(x, y) = \sqrt[2]{\sum_{i=1}^{N} \|x_i - y_i\|^2}$.
  - `minkowski`: $d_{minkowsky}(x, y) = \sqrt[\mathtt{p}]{\sum_{i=1}^{N} \|x_i - y_i\|^{\mathtt{p}}}$.
  - `cityblock`: $d_{cityblock}(x, y) = \sum_{i=1}^{N} \|x_i - y_i\|$

    (*Note:* $\|x\|$ is the absolute value.)

  See figure 1 for a nicer visualisation.
  Don't be afraid of this. It looks scary at first, but it is not beyond the scope of what you already learned. **Solution:**

```python
import math


def pdist(x, y, **kwargs):
    """
    Calculates the distance between x and  y using the given metric.

    The default metric used for distance calculation is the euclidean metric. Other options are 'cityblock' and 'minkowski'. If using minkowski distance, the parameter p >= 1 can be supplied.
    """
    # check whether metric is given, and use euclidean as default
    metric = kwargs['metric'] if 'metric' in kwargs else 'euclidean'
    if metric == 'minkowski':
        p = kwargs['p'] if 'p' in kwargs else 2
    if metric == 'euclidean':
        squared = [abs(z)**2 for z in sub(x,y)]
        return math.sqrt(sum(squared))
    elif metric == 'cityblock':
        diff = [abs(z) for z in sub(x, y)]
        return sum(diff)
    elif metric == 'minkowski':
        if p < 1:
            return None
        to_p = [abs(z)**p for z in sub(x, y)]
        return sum(to_p) ** (1/p)
    else:
        return None
```

- **BONUS:** `outer(x, y)`: calculates the outer product of two vectors. The outer product of two vectors is the tensor product of the two. For two vectors with size n the outer product will yield a matrix of size (n, n). The outer product is calculated as $C_{i_j} = x_i y_j$. **Solution:**

```
def outer(x, y):
    """Returns a list of lists representinx a matrix calculated from
        the two vectors, each list represents one row"""
    # one liner:
    # return [[xi * yi for yi in y] for xi in x]
    result = []
    for i in range(len(x)):
        row = []
        for j in range(len(y)):
          row.append(x[i] * y[j])
        result.append(row)
        return result
```

You can assume that all vectors have the same size. To verify your function works correctly you can test it with the following values:
$a = (1, 2, 3), b = (4, 5, 6), c = [0, 1, 0, 0, 1], d = [1.5, 2.5, 3.5, 4.5, 5.5]$

Call	Result
`add(a,b)`	(5, 7, 9)
`add(b,a)`	(5, 7, 9)
`sub(a,b)`	(-3, -3, -3)
`dot(c,d)`	8.0
`angle(a,b)`	approx. 0.23
`pdist(a,b)`	approx. 5.20
`pdst(a,b, metric='cityblock')`	9
`pdist(c,d, metric='minkowski', p=3)`	6.14
`outer(a,b)`	$\begin{bmatrix} 4 & 5 & 6 \\ 8 & 10 & 12 \\ 12 & 15 & 18 \end{bmatrix}$

2 Which to What

We will give you a few examples of datasets. Which collection type would you use to save this data in?
Please explain your answers.

- options a user can click on in a program menu
  - **Solution:** List
- a country's name, population and capital
  - **Solution:** Tuple or Dict
- food ingredients
  - **Solution:** List
- data about a music album

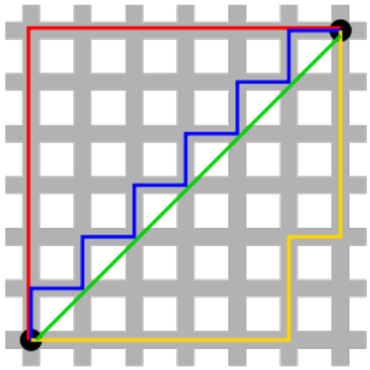

Figure 1: Blue, red, and yellow are examples for the cityblock distance (it looks like following the streets of a grid city) and green is the euclidean distance. Taken from `https://en.wikipedia.org/wiki/Taxicab_geometry#/media/File:Manhattan_distance.svg`

  - **Solution:** Dict

- people who visit Basic Programming in Python or Scientific Programming in Python
  - **Solution:** Set
- IDs of upcoming orders in an online shop
  - **Solution:** List
- nicknames and account information in a forum (e-mail address, password, real name [optional], birth date [optional], ...)
  - **Solution:** Dict

3 The Transform

In the file `my_collection.py` you will find two lists defined: `subjects` and `attributes`. The first list contains subject ids, wheras the second one contains attributes corresponding to the subject ids. So the subject at index 0 of `subjects` has the attribute at index 0 of `attributes` and the subject at index 9 has the attribute at index 9. You get the idea.
You may notice that each subject id appears multiple times. Your task now is to create a function to create one dictionary which uses the subject id as a key and a *list* of attributes as the value. **Solution:**

```
# the given lists
all_subjects = [0, 0, 1, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5, 5,
    5, 6, 6, 6, 6, 6, 6, 7, 7, 7, 7, 7, 7, 8, 8, 8, 8, 8, 9, 9, 9]
```

```python
all_attributes = ['Materialistic', 'Neat', 'Active', 'Welcoming', 'Creative', 'Ambitious', 'Geek', 'Welcoming', 'Neat', 'Creative', 'Geek', 'Quiet', 'Shy', 'Neat', 'Ambitious', 'Adventurous', 'Active', 'Welcoming', 'Adventurous', 'Neat', 'Ambitious', 'Excitable', 'Active', 'Welcoming', 'Quiet', 'Excitable', 'Ambitious', 'Adventurous', 'Quiet', 'Geek', 'Active', 'Spiritual', 'Quiet', 'Excitable', 'Materialistic', 'Geek', 'Welcoming', 'Excitable', 'Adventurous']

# This is the function you need to implement
def clean_up(subjects, attributes):
  """
  This function takes a list of subject ids which correspond to attributes
  in the second list, and forms a dictionary out of them, with the unique
  subject id as the key and a list of their attributes as the corresponding
  value.
  """
  # and as a one-liner:
  # return {sub : attributes[subjects.index(sub):subjects.index(sub)+subjects.count(sub)] for sub in set(subjects)}
  # create the empty dict that we will keep adding the stuff into one by one
  subject_dict = dict()
  # idx is the counter going from 0 to 38
  # using enumerate saves as the line: subj_id = subjects[idx]
  for idx, subj_id in enumerate(subjects):
  # if this is the first time we encounter this subject id, add it to the dict
  # the value is an empty list for now, we will now add all the attributes
    if subj_id not in subject_dict:
      subject_dict[subj_id] = []
    # add the current attribute to the list of the subject
    subject_dict[subj_id].append(attributes[idx])

  return subject_dict

# So nice, tidy and clean.
subject_dict = clean_up(all_subjects, all_attributes)
```