Panda Internet Radio

Goal: Implementing the Panda `next_song` function based on history of song likes and dislikes.

Side Effect: Learning about `docstrings` and `doctests`

Note: This tutorial implements a slightly different spec than defined by the panda.ipyb readme and tested by test.py

Distance Function

A song is represented as a list of genes, it's "genome". Each gene can have value either 0 or 1.

We want a `distance` function that will give us a sense of how different two songs are.

In [ ]:
```
def distance(song_0, song_1):
    pass
```

In [ ]:
```
distance([0, 1], [1, 0])
```

In [ ]:
```
import doctest
doctest.run_docstring_examples(distance, globals(), verbose=False)
```

Docstrings and doctests

`doctests` are test cases embedded within `docstrings`, that can actually be run and tested automatically:

- Nothing output if all tests succeed
- An error reported if one or more tests fail

These are part of the standard python library, so are always available. Read more about them at
https://docs.python.org/3/library/doctest.html (https://docs.python.org/3/library/doctest.html)

A useful approach is to include the following at the bottom of your file (e.g., your `lab.py`):

In [ ]:
```
if __name__ == '__main__':
    # running lab.py invokes the doctests for *all* functions in the file...
    import doctest
    doctest.testmod()
```

Inside jupyter, we'll instead invoke specific doctests directly, e.g.:

In [ ]:
```
import doctest
doctest.run_docstring_examples(distance, globals(), verbose=False)
```

Average Distance

Add another data structure -- a "music library" implemented as a dictionary consisting of song_ids as keys, and the corresponding genome for the song as the value.

Now we'd like to get a sense of the average distance between one song and a whole list of other songs.

In [ ]:
```
def average_distance(song_id_list, song_id, music):
    """Return average distance between song_id and music library

    inputs: list of song_ids, a single song_id, and a music dictionary
    returns: average distance, computed as the sum of distances
      divided by the number of distances considered, between song given
      by song_id and the songs in song_id_list
    note: average_distance from empty list is 0
    >>> music = {'Stairway': [0,0],
    ...          '5th': [0,1],
    ...          'Blues': [1,1],
    ...          'Requiem': [1,0]}
    >>> average_distance([], 'Stairway', music)
    ???
    >>> average_distance(['Stairway'], 'Stairway', music)
    ???
    >>> average_distance(['5th'], 'Stairway', music)
    ???
    >>> average_distance(['5th','Blues'], 'Stairway', music)
    ???
    >>> average_distance(['5th','Blues','Requiem'], 'Stairway', music)
    1.3333333333333333
    """
    pass
```

In [ ]:
```
doctest.run_docstring_examples(average_distance, globals(), verbose=False)
```

Note that doctest using string comparisons between the expected and actual output, not more sophisticated tests like ==. Thus in the above, we need 0.0 for expected return values, not 0.

Goodness Function

The "goodness" of a song is defined to be the average distance of the song from a list of disliked songs, minus the average distance of the song from a list of liked songs. This is meant to favor songs far away from disliked songs, but close to liked songs.

In [ ]:
```
def goodness(likes, dislikes, song_id, music):
    """Return `goodness` of song_id based on history of likes/dislikes

    inputs: likes, dislikes are lists of 'liked' and 'disliked' song_ids.
            song_id is the id of a song we'd like to know the "goodness" of.
            music is a music dictionary.
    returns: "goodness" value (float) of song_id
    >>> music = {'Stairway': [0,0],
    ...          '5th': [0,1],
    ...          'Blues': [1,1],
    ...          'Requiem': [1,0]}
    >>> likes = []
    >>> dislikes = []
    >>> goodness(likes, dislikes, 'Stairway', music)
    ???
    >>> likes = ['Requiem']
    >>> dislikes = ['5th', 'Blues']
    >>> goodness(likes, dislikes, 'Stairway', music)
    ???
    """
    return average_distance(dislikes, song_id, music) - \
           average_distance(likes, song_id, music)
```

In [ ]:
```
doctest.run_docstring_examples(goodness, globals(), verbose=False)
```

Next Song

Now to answer the key question -- what song should be picked next, based on previously played song likes and dislikes?

In [ ]:
```
def next_song(likes, dislikes, music):
    """Return next song to play based on history of likes/dislikes

    inputs: likes is list of 'liked' previously played song ids.
            dislikes is list of 'disliked' previously played song ids.
            music is a music dictionary.
    returns: ID for an unplayed song in dictionary with best goodness value

    >>> music = {'Stairway': [0,0],
    ...          '5th': [0,1],
    ...          'Blues': [1,1],
    ...          'Requiem': [1,0]}
    >>> likes = []
    >>> dislikes = ['Blues']
    >>> next_song(likes, dislikes, music)
    'Stairway'

    >>> likes = ['Blues']
    >>> dislikes = []
    'Requiem'
    """
    pass
```

In [ ]:
```
doctest.run_docstring_examples(next_song, globals(), verbose=False)
```

In [ ]:
```
# Let's debug this case a little more...
music = {'Stairway': [0,0],
         '5th': [0,1],
         'Blues': [1,1],
         'Requiem': [1,0]}
likes = ['Blues']
dislikes = []
print('next_song (best goodness) is song id:',
      next_song(likes, dislikes, music))
for song_id in music:
    print('goodness for song id', song_id, "=",
          goodness(likes, dislikes, song_id, music))
```

So, we only had one of the possible return values in our test case. Maybe we should be more thorough. Might the following work?

In [ ]:
```
likes = ['Blues']
dislikes = []
next_song(likes, dislikes, music)
```

Not really. So how deal with that ambiguous return value, using doctest?

Using Python random module to generate large test cases

(Note -- in 6.009 we're not allowing `import random`; but this gives you the idea)

How do you check your output is correct?

In [ ]:
```
import random
```

In [ ]:
```
def generate_test(music_size, gene_length, like_size, dislike_size):
    random.seed(6009) # A fixed random seed results in deterministic output.
    music = {}
    for i in range(music_size):
        music["song_"+str(i)] = [random.randint(0,1) for _ in range(gene_length)]
    likes = random.sample(music.keys(), like_size)
    dislikes = random.sample(music.keys(), dislike_size)
    return music, likes, dislikes
```

In [ ]:
```
music, likes, dislikes = generate_test(30, 10, 10, 0)
print("music:", music)
print("Example: song_1 =", music["song_1"])

next_song_id = next_song(likes, dislikes, music)
print("next_song_id:", next_song_id)
print("goodness:", goodness(likes, dislikes, next_song_id, music))
```

How do we know if the next_song_id is reasonable or correct?

In [ ]:
```
# Check what the best goodness actually is, among unplayed songs...
goodness_values = [goodness(likes, dislikes, song, music)
                       for song in music
                           if song not in likes and song not in dislikes]
max(goodness_values)

#or as a comprehension without building the list...
max(goodness(likes, dislikes, song, music)
        for song in music
            if song not in likes and song not in dislikes)
```

Note that now we're **writing more complicated code to verify** or check answers. That's a job most likely better suited to `unittest`!