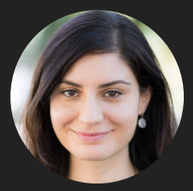
02-06-2023

CS051A

INTRO TO COMPUTER SCIENCE WITH TOPICS IN AI

6: Sequences



Alexandra Papoutsaki

she/her/hers

Lectures

Zilong Ye

he/him/his

Labs

Welcome to lecture 6! Are there any questions? Don’t forget you have lab today or tomorrow and assignment 3 is out and due as always this coming Sunday.

Lecture 6: Sequences

- Lists
- Sequences
- Tuples

Over last week, we started learning how to make our code more intelligent with boolean (`bool`) expressions, `if` statements, and `for` and `while` loops. Today, we will continue with learning how our programs can store and process significantly more data than we have seen so far.

scores-list.py

- A program that contains a set of functions for reading in scores and calculating various statistics on them.

```
# scores-list.py
# A set of functions for reading in scores and calculating
# various statistics from the input scores.

def get_scores():
    """
    Reads user input of numerical scores as floats into a list
    and returns the list
    :return: None
    """
```

Let’s start by looking at what happens when we run scores-list.py, a program that contains a set of functions for reading in scores and calculating various statistics on them.

`scores-list.py` main - What does it do?

- First, it prompts the user to enter a list of scores one at a time
  - Uses a while loop that keeps asking the user for a new score. What is the exit condition?
    - Checks to see if the line is empty: `while line != ""`
- Then, calculate various statistics based on what was entered. How are we calculating these statistics?
- Average?
  - We could keep track of the sum and the total number of scores entered and divide them at the end.
- Max (min)?
  - Keep track of the largest (smallest) score seen so far. Each time a new one is entered, see if it's larger (smaller). If so, update the largest (smallest).
- Median?
  - The challenge with median is that we can't calculate it until we have all of the scores. We need to sort them and then find the middle score.
- Why can't we do this using `int`/`float` variables?
  - We don't know how many scores are going to be entered. Even if we did, if we had 100 students in the class, we'd need 100 variables!

What does scores-list.py do exactly? First, it looks like it prompts the user to enter a list of scores one at a time. When we want to stop providing scores, we just don’t provide anything and hit Enter. This is achieved using a while loop where the exit condition is to examine whether the provided line is empty.
The program then proceeds with calculating various statistics based on what was entered. How would we go by calculating the average? We could keep track of the sum and the total number of scores entered and divide them at the end. What about finding the max or min score? We can keep track of the largest (smallest) score seen so far. If we encounter a larger (smaller) one, then we know that this is the new largest (smallest). The median is the middle number in a sorted list. The challenge here is that we can’t calculate it until we have all the scores. We would need to sort them and then find the middle one.
You might be thinking, why can’t do all this using variables of type int or float? First, we don’t know in advance how many scores the user will enter. And even if we did, our program would be hard to scale. For example, if we had 100 students, we would need 100 variables! Luckily, Python has a good solution for us called lists.

Lists

- List: a data structure.
  - Data structure: a way of storing and organizing data.
- Lists allow us to store multiple values using only a single variable to refer to them!
- Creating lists: provide elements separated by comma and enclosed in square brackets.
- Lists are a type and represent a value, just like `float`, `int`, `bool` and `str`. We can assign them to variables, print them, etc.

```
>>> [7, 4, 3, 6, 1, 2]
[7, 4, 3, 6, 1, 2]
>>> 10
10
>>> [10]
[10]
>>> my_list = [7, 4, 3, 6, 1, 2]
>>> my_list
[7, 4, 3, 6, 1, 2]
>>> type(my_list)
<class 'list'>
```

So what are lists? A list is a data structure, that is a tool that most programming languages have and which allows us to store and organize data. Even cooler? We only need a single variable to refer to the multiple values that a list stores. The syntax for creating lists is providing the elements separated by comma and enclosed in square brackets. Lists are a type and represent a value, just like `float`, `int`, `bool` and `str`. We can assign them to variables, print them, etc.

Accessing Lists

- []: creates an empty list.
- We can access a particular value in the list by using the [] to "index" into the list.
  - Indexing starts at 0!
- Be careful of index out of range errors!
  - We can only index from 0… length-1.
- Negative indexing counts back from the end of the list.

```
>>> my_list = [7, 4, 3, 6, 1, 2]
>>> my_list[3]
6
>>> my_list[0]
7
>>> my_list[20]
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versi
    exec(code, self.locals)
  File "<input>", line 1, in <module>
IndexError: list index out of range
>>> my_list[-1]
2
>>> type(my_list[3])
<class 'int'>
```

If we want to create an empty list, we use empty square brackets. To access a particular value in a list, we can use the square brackets to index into the list, that is ask for the first, second, third, etc, element. As always, in CS we start counting at 0 so indexing starts at 0. Be careful when indexing lists as you can only ask for the elements between 0… length-1. If you go >= length, you will get an index out of range error. Well, to be precise, indexing can be negative, too! When you provide a negative index, then you get the values from the end of the list reversely. Notice that the type of my_list[3] is an int, not a list!

Storing other things in lists

- A list is a contiguous set of spaces in memory.
  - `[ _ , _ , _ , _ ]`
  - We can store anything in each of these spaces.
- In general, it's a good idea to have lists be homogeneous, i.e. be of the same type.

```
>>> ["this", "is", "a", "list", "of", "strings"]
['this', 'is', 'a', 'list', 'of', 'strings']
>>> list_of_strings = ["this", "is", "a", "list", "of", "
>>> list_of_strings[0]
'this'
>>> [1, 5.0, "my string"]
[1, 5.0, 'my string']
>>> mixed_list = [1, 5.0, "my string"]
>>> type(mixed_list[0])
<class 'int'>
>>> type(mixed_list[1])
<class 'float'>
>>> type(mixed_list[2])
<class 'str'>
```

A list is a contiguous set of spaces in memory. If we were to draw its representation, it would looks like [_,_,_,_]. We can store anything in each of these spaces, even mix and match different types! In general, it’s a good idea to have lists be homogeneous, i.e., be of the same type. Many programming languages, unlike Python, won’t let you mix and match different types.

Slicing

- Sometimes, we want more than just one item from the list (this is called slicing).
- We can specify a range in the square brackets, [], using the colon (:)
  - `list[start:end]` will return a new list with the elements from `start` index through `end-1`.
  - `list[start:]` will return a new list with the elements from `start` to the end of the list.
  - `list[:end]` will return a new list with the elements from 0 through `end-1`.
  - `list[:]` will return a copy of the entire list.

```
>>> list_of_numbers = [32, 4, -1, 15, -20]
>>> list_of_numbers[0:3]
[32, 4, -1]
>>> list_of_numbers[1:4]
[4, -1, 15]
>>> list_of_numbers[1:]
[4, -1, 15, -20]
>>> list_of_numbers[:2]
[32, 4]
>>> list_of_numbers[:]
[32, 4, -1, 15, -20]
>>> list_of_numbers[1:1]
[]
>>> list_of_numbers[-3:-1]
[-1, 15]
```

Sometimes we want a “slice” off a list, that is a contiguous subset of the elements that it contains. We can specify the range of the slicing using the colon notation.
list[start:end] will return a new list with the elements from start index through end-1.
list[start:] will return a new list with the elements from start to the end of the list.
list[:end] will return a new list with the elements from 0 through end-1.
list[:] will return a copy of the entire list.
Look at the following examples:
list_of_numbers = [32, 4, -1, 15, -20]
list_of_numbers[0:3]
[32, 4, -1]
list_of_numbers[1:4]
[4, -1, 15]
list_of_numbers[1:]
[4, -1, 15, -20]
list_of_numbers[:2]
[32, 4]
list_of_numbers[:]
[32, 4, -1, 15, -20]
list_of_numbers[1:1]
[]
list_of_numbers[-3:-1]

[-1, 15]

Looping over lists

- We can use the for loop to iterate over each item in the list.
- This is often called a "foreach" loop, i.e. for each item in the list, do an iteration of the loop.

```
>>> list_of_numbers = [32, 4, -1, 15, -20]
>>> for value in list_of_numbers:
...     print(value)
...
32
4
-1
15
-20
```

To iterate over each item in a list, we can use a for loop with the syntax
for element in list:
    statement
This is often called a "foreach" loop, i.e. for each item in the list, do an iteration of the loop.

Practice time

- Write a function called `sum` that returns the sum of all the values in a list of numbers.

```
>>> def sum(numbers):
...     total = 0
...
...     for val in numbers:
...         total += val
...
...     return total
...
>>> sum([13, -2, 47, 9, -5])
62
```

Let’s practice by writing a function called sum that returns the sum of all the values in a list of numbers.
Using a foreach loop, we can try something like:

```
def sum(numbers):
    total = 0

    for val in numbers:
        total += val

    return total
```

Calculating the average of a list - the inelegant way

```python
def inelegant_average(scores):
    """
    Calculates the average of the values in list scores in an inelegant way
    :param scores: (list) a list of numbers that correspond to scores
    :return: (float) the average of the values in scores
    """
    sum = 0.0
    count = 0

    for score in scores:
        sum += score
        count += 1

    return sum / count
```

If we wanted to calculate the average of the values in a list using what we just learned, we would probably come up with a function that looks like:

```
def inelegant_average(scores):
    """
    Calculates the average of the values in list scores in an inelegant way
    :param scores: (list) a list of numbers that correspond to scores
    :return: (float) the average of the values in scores
    """
    sum = 0.0
    count = 0

    for score in scores:
        sum += score
        count += 1

    return sum / count
```

Although this is definitely correct, it is not the most elegant way of going about it.

Calculating the average of a list - the elegant way

```python
def average(scores):
    """
    Calculates the average of the values in list scores in an elegant way
    :param scores: (list) a list of numbers that correspond to scores
    :return: (float) the average of the values in scores
    """
    return sum(scores) / len(scores)
```

A much more elegant way would be to use the built-in Python sum and len functions which would simplify our code to:

```
def average(scores):
    """
    Calculates the average of the values in list scores in an elegant way
    :param scores: (list) a list of numbers that correspond to scores
    :return: (float) the average of the values in scores
    """
    return sum(scores) / len(scores)
```

Built-in functions over lists

- Length of list
  - `len(list)`
- Max of list
  - `max(list)`
- Min of list
  - `min(list)`
- Sum of list
  - `sum(list)`

```
>>> list_of_numbers = [32, 4, -1, 15, -20]
>>> len(list_of_numbers)
5
>>> len([])
0
>>> max(list_of_numbers)
32
>>> min(list_of_numbers)
-20
>>> sum(list_of_numbers)
30
```

In general, there are a number of built-in functions that allow us to get information about a list. E.g., `len` returns the number of elements in a list, `max` the maximum element, `min` the minimum, and `sum` sums them up.

List methods

- Lists are objects therefore have methods.
  - Object: a software bundle that consists of properties and behavior. Behavior is controlled by methods.
  - We call a method of an object using the dot operator.
- Syntax: `myList.someMethod(argument)`
- https://docs.python.org/3/tutorial/datastructures.html
- Or `help([])`
- Or `help(list)`

Lists are objects and therefore have methods. What is an object you may ask? Objects are a central concept in the object-oriented programming paradigm. They represent a software bundle that consists of properties and behavior. The behavior is controlled by methods which are similar to functions but operate on the properties of a specific object. We will talk about objects later in the class but in general, to call a method of an object we will use the dot operator. For example, if we had a list called `myList`, the Python syntax for calling one of its methods, let's say `someMethod`, would be `myList.someMethod(argument)`. You can read more about the methods that list has by visiting the official python tutorial or using `help`.

append

- Adds a value at the end of a list.

```
>>> list_of_numbers = [32, 4, -1, 15, -20]
>>> list_of_numbers.append(47)
>>> list_of_numbers
[32, 4, -1, 15, -20, 47]
```

- Notice that append does not return a new list, it just modifies the existing list!

Let's see some basic list methods. The first one is called append and it adds or appends a value at the of a list. Notice that append does not return a new list, it just modifies the existing list.

pop

- Removes a value from the end of a list and returns it.

```
>>> list_of_numbers
[32, 4, -1, 15, -20, 47]
>>> list_of_numbers.pop()
47
>>> list_of_numbers
[32, 4, -1, 15, -20]
```

- Notice that pop both modifies the list and returns the last value. If you want to use this value, you need to store it.

```
>>> popped = list_of_numbers.pop()
>>> popped
-20
>>> list_of_numbers
[32, 4, -1, 15]
```

- pop also has another version where you can specify the index.

```
>>> list_of_numbers.pop(1)
4
>>> list_of_numbers
[32, -1, 15]
```

If you want to remove and return a value from the end of a list, you would use the pop method. Notice that pop both modifies the list and returns the last value. If you want to use this value, you need to store it in some variable, otherwise it will be lost! There is another version of pop that takes a parameter which corresponds to the index of the element you want to remove and return.

insert

- Inserts a value at a specific index.

```
>>> list_of_numbers
[32, -1, 15]
>>> list_of_numbers.insert(2, 100)
>>> list_of_numbers
[32, -1, 100, 15]
```

- Notice that `insert` does not return a new list but modifies the underlying one.

If you want to insert a value at a specific index, not just append it at the end of the list, you would use the method `insert`. As with `append`, notice that insert does not return a new list but modifies the underlying one.

sort

- Sorts a list in ascending order.

```
>>> list_of_numbers
[32, -1, 100, 15]
>>> list_of_numbers.sort()
>>> list_of_numbers
[-1, 15, 32, 100]
>>> list_of_strings
['this', 'is', 'a', 'list', 'of', 'strings']
>>> list_of_strings.sort()
>>> list_of_strings
['a', 'is', 'list', 'of', 'strings', 'this']
```

- Again, `sort` does not return a new list but modifies the underlying one.

Finally, there is another handy method called `sort` which sorts a list in ascending order, i.e. 1, 2, 3 if you have numbers, a, b, c if you have strings etc. Again, `sort` does not return a new list but modifies the underlying one.

scores-list.py

- There is a function called `get_scores`. It gets the scores and returns them as a list.
  - starts with an empty list,
  - uses `append` to add them on to the end of the list,
  - returns the list when the loop finishes.
- `median` function
  - sorts the values
    - notice again that sort does NOT return a value, but sorts the list that it is called on.
  - returns the middle entry

Let's look in more depth at the `scores-list.py` file and specifically at the functions `get_scores` and `median`. `get_scores` asks the user for one score at a time and returns them in a list. How does it accomplish it? It starts with an empty list, uses `append` to add them to the end of the list, and returns the list when the loop finishes (no new score is provided).
The `median` function works by sorting the value, finding the middle entry based on whether you have an odd or even number of elements, and returns the middle entry. Notice again that `sort` does not return a value but sorts the list that is called on. This is a recurring theme with the list methods we have seen.

Lists are mutable

- We can change (or mutate) the values in a list.
- Notice that many of the methods that we call on lists change the list itself.
- We can mutate lists with methods, but we can also change particular indices.

```
>>> list_of_numbers
[-1, 15, 32, 100]
>>> list_of_strings[2] = 100
>>> list_of_numbers
[-1, 15, 32, 100]
```

This is because lists are mutable. This means that we can change (or mutate) the values in a list. The methods we have called on lists change the list itself. We can also change particular indices of the list.

Lecture 6: Sequences

- Lists
- Sequences
- Tuples

Which brings us to a new and more general category of data structures called sequences.

Sequences

- Lists are part of a general category of data structures called sequences.
- Sequences represent a... sequence of things.
- **All** sequences support a number of shared behavior.
  - The ability to index using `[]`.
  - The ability to slice using `[:]`.
  - A number of built-in functions:
    - `len`, `max`, `min`.
  - The ability to iterate over them with a for loop.
- We've actually seen one other sequence. Strings!

Lists are part of a general category of data structures called sequences which as their name says represent a sequence of things. All types of sequences support a number of shared behavior: indexing, slicing, built-in functions, such as `len`, `max`, and `min`, and the ability to iterate over them with a foreach loop. We've actually seen one other sequence beyond lists already. Strings!

Strings as sequences

- We can do all sorts of sequence-like things to strings!
- Strings, however, are immutable! We cannot mutate them.

```
>>> fruit = "banana"
>>> fruit[4]
'n'
>>> fruit[2:5]
'nan'
>>> len(fruit)
6
>>> for letter in fruit:
...     print(letter)
...
b
a
n
a
n
a
```

```
>>> fruit[4] = "c"
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/code.py", line 90, in runcode
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Since strings are sequences, we can do all sorts of sequence-like things to them including indexing, slicing, and using built-in functions. In contrast to lists though, strings are immutable. If you try to alter a string, you will get an error!

more-lists.py

- What does the `list-to-string` function do?
- Creates a list from a string:
  - Takes as input a list. A list of almost any type, as long as we can call `str()` on.
  - Concatenates all the items in the list into a single string.
  - `result` starts out as the empty string.
  - It iterates through each item in the list and concatenates them on to the `result`
  - Returns the entire `result` list minus the last element (which is " ")

Let's look at the more-lists.py file. What do you think that the lists-to-string function do? Aptly named, it creates a list from a string. It takes as input a list which can hold almost anything as long as we can call the str() function. It then concatenates all. The items in the list into a single string result which starts empty. As we iterate through each item in the list, we concatenate it to the result (separated by " "). At the end, we return the entire result minus the last element which is a " ".

Alternate way of iterating over lists

```
>>> for letter in fruit:
...     print(letter)
...
b
a
n
a
n
a
>>> for i in range(len(fruit)):
...     print(fruit[i])
...
b
a
n
a
n
a
```

We have already seen that you can use a foreach loop to iterate through each of the elements of a list. Alternatively, we can use a traditional for loop and index through each of the list elements.

Practice time

- Write a function called `multiply_lists` that takes two lists of numbers and creates a new list with the values pairwise multiplied. E.g.,

```
>>> list1 = [1, 2, 1, 2]
>>> list2 = [1, 2, 3, 4]
>>> multiply_lists(list1, list2)
[1, 4, 3, 8]
```

```
def multiply_lists(list1, list2):
    """
    Creates a new list that is the result of the multiplication of two equa
    :param list1: (list) the first list of numbers
    :param list2: (list) the second list of numbers
    :return: (list) a list where each index corresponds to the multiplicati
    in list1 and list2
    """
    result = []

    if len(list1) != len(list2):
        print("Error: lists are not of equal length!")
    else:
        for i in range(len(list1)):
            result.append(list1[i] * list2[i])

    return result
```

Let’s practice by writing a function called multiply_lists that takes two lists of numbers and creates a new list with the values pairwise multiplied. E.g.,
list1 = [1, 2, 1, 2]
list2 = [1, 2, 3, 4]
multiply_lists(list1, list2)
[1, 4, 3, 8]
You should end up with something similar to:

```
def multiply_lists(list1, list2):
    """
    Creates a new list that is the result of the multiplication of two equally-lengthed lists of numbers
    :param list1: (list) the first list of numbers
    :param list2: (list) the second list of numbers
    :return: (list) a list where each index corresponds to the multiplication of the numbers existing in the same index
    in list1 and list2
    """
    result = []

    if len(list1) != len(list2):
        print("Error: lists are not of equal length!")
    else:
        for i in range(len(list1)):
```

```
        result.append(list1[i] * list2[i])

    return result
```

Lecture 6: Sequences

- Lists
- Sequences
- Tuples

So far, we have seen two types of sequences: lists and strings. There is a third category called tuples.

Tuples

- Tuple: an immutable list. Type of sequence.
- Tuples can be created using parentheses (instead of []).

```
>>> my_tuple = (1, 2, 3, 4)
>>> my_tuple
(1, 2, 3, 4)
>>> another_tuple = ("a", "b", "c", "d")
>>> another_tuple
('a', 'b', 'c', 'd')
```

- Notice that when they print out, they also show using parentheses.

For a variety of reasons (we'll get into some eventually), we also have immutable lists, called tuples. Tuples can be created using parenthesis (instead of square braces). Here’s an example:
my_tuple = (1, 2, 3, 4)
my_tuple
(1, 2, 3, 4)
another_tuple = ("a", "b", "c", "d")
another_tuple
('a', 'b', 'c', ‘d')
Notice that when they print out, they also show using parentheses.

Tuples as immutable sequences

```
>>> my_tuple[0]
1
>>> my_tuple[3]
4
>>> for val in my_tuple:
...     print(val)
...
1
2
3
4
>>> my_tuple[1:3]
(2, 3)
>>> my_tuple[0] = 1
Traceback (most recent call last):
  File "/Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/cod
    exec(code, self.locals)
  File "<input>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Since tuples are sequences, we can use the standard way of indexing them, slicing them, iterating through them. But tuples are immutable sequences so we can't change them, call methods such as `append`, etc.

Unpacking tuples

- If we know how many items are in a tuple, we can "unpack" it into individual variables.

```
>>> my_tuple = (1, 2, 3)
>>> my_tuple
(1, 2, 3)
>>> (x, y, z) = my_tuple
>>> x
1
>>> y
2
>>> z
3
```

```
>>> (x, y, z) = (10, 11, 12)
>>> x
10
>>> y
11
>>> z
12
>>> x, y, z = "apple", "banana", "pineapple"
>>> x
'apple'
>>> y
'banana'
>>> z
'pineapple'
```

If we know how many items are in a tuple, we can "unpack" it into individual variables. E.g.,
my_tuple = (1, 2, 3)
my_tuple
(1, 2, 3)
(x, y, z) = my_tuple
x
1
y
2
z
3
Notice that you can do the same thing using the following syntax which doesn’t even include parentheses:
(x, y, z) = (10, 11, 12)
x
10
y
11
z
12
x, y, z = "apple", "banana", "pineapple"

x
'apple'
y
'banana'
z
‘pineapple'

movies.py

- Tuples are useful for representing data with fixed entries.
- Look at the `print_movies` function movies.py.
  - It iterates over the list, just like any other list.
  - `movie_pair` is a tuple (each entry in the list is a tuple). We unpack the tuple to get at the two values in the tuple.
    - We also could have written `movie_pair[0]` and `movie_pair[1]` (see `print_movies2`), though unpacking is much cleaner.
  - Once we have the two values, we can print them out
    - `\t` is a special character that represents a tab (like `\n`, which represents the end of line character)
- Look at the `print_movies3` function.
  - We can unpack the two values of the tuple *in* the for loop. Any of the variants is fine for this class!

Tuples are useful for representing data with fixed entries. Look at the print_movies function movies.py a file that works on a database of movies and their corresponding critic scores.
The function iterates over the list, just like any other list.
movie_pair is a tuple (each entry in the list is a tuple). We unpack the tuple to get at the two values in the tuple. We also could have written movie_pair[0] and movie_pair[1] (see print_movies2), though unpacking is much cleaner. Once we have the two values, we can print them out. Please note that
\t is a special character that represents a tab (like \n, which represents the end of line character)
Now, look at the print_movies3 function. We can unpack the two values of the tuple *in* the for loop. Any of the variants is fine for this class!

`get_movie_score` function

- What does the `get_movie_score` function do?
  - Takes two parameters, a movie database and a movie title.
  - It iterates through the movie database and tries to find the matching title.
    - If it finds it, it returns the score.
    - If it doesn't find it, it will iterate through all of the movie entries, finish the for loop and return -1.0

Now let’s look into the get_movie_score function? What do you think this function does? It takes two parameters, a movie database and a movie title. It iterates through the movie database and seeks the title. If it finds it, it returns the critic score. If it has gone through the entire database and it’s not there, it returns -1.0.

Practice time

- Write a function called `my_max` that takes a list of positive numbers and returns the largest one.

```
>>> def my_max(numbers):
...     max = -1
...
...     for num in numbers:
...         if num > max:
...             max = num
...
...     return max
...
```

- Key idea: have a variable that keeps track of the largest number seen so far. At each iteration, compare the current number to max, if it's bigger, update the max value.
- Why initialize it to -1? We need to initialize it to something that is smaller than any of the values. We could also have done something like `max = numbers[0]` (assuming that the input would have at least one value).

Let’s practice by writing a function called my_max that takes a list of positive numbers and returns the largest one. A key idea we will use is to have a variable that keeps track of the largest number seen so far. At each iteration, we compare the current number to max. If it's bigger, update the max value.
Why initialize it to -1? We need to initialize it to something that is smaller than any of the values. We could also have done something like max = numbers[0] (assuming that the input would have at least one value).

`get_highest_rated_movie` function

- What does the `get_highest_rated_movie` function do?
  - Very similar idea to `my_max` function.
    - We're finding the largest score.
    - We also keep track of the movie with the highest score so that we can return that at the end.

Now let's look into the get_highest_rated_movie function? What do you think this function does? It takes one parameters, a movie database, and it acts similarly to my_max function by searching for the largest element (in this case critic score). We also keep track of the movie with the highest score so that we can return that at the end.

Practice time

- Write a function called `get_movies_above_threshold` that takes as input a movie database and a critic score threshold and returns all of the movies above that threshold.

```
def get_movies_above_threshold(movie_db, threshold):
    """
    Given a database and a threshold critic score, it returns a list of movies with scores above the threshold
    :param movie_db: (list) a list of tuples that correspond to movies (str) and scores (float)
    :param threshold: (num) the threshold critic score to filter movies by.
    :return: (list) a list of movie titles that have critic scores higher than the threshold
    """
    movies_above = []

    for (movie, score) in movie_db:
        if score >= threshold:
            movies_above.append(movie)

    return movies_above
```

For the final practice for today’s lecture, let’s write a function called get_movies_above_threshold that takes as input a movie database and a critic score threshold and returns all of the movies above that threshold. Hint: it should look a lot like some of the other examples that we've seen that build up a list of answers (e.g. string_to_list in more-lists.py code). Look at get_movies_above_threshold function in movies.py code. We start out with an empty list, iterate through the movie database, check each movie score; if the score is greater than or equal to the threshold, append it to the end of our list of movies that are above the threshold. Once we go through the entire database, we return the list of movies above the threshold.

Resources

- Textbook: Chapters 9 and 10
- scores-list.py
- more-lists.py
- movies.py

Practice Problems

- Practice 4 (solution)

Homework

- Assignment 3