Bacon Number

You are not logged in.

Please Log In for full access to the web site.
Note that this link will take you to an external site (`https://shimmer.mit.edu`) to authenticate, and then you will be redirected back to this page.

Table of Contents

1) Preparation

This lab assumes you have Python 3.11 or later installed on your machine (3.12 recommended).

The following file contains code and other resources as a starting point for this lab: `bacon.zip`

Most of your changes should be made to `lab.py`, which you will submit at the end of this lab. Importantly, you should not add any imports to the file.

Your raw score for this lab will be counted out of 5 points. Your score for the lab is based on:

- correctly answering the questions throughout this page (2 points),
- passing the tests in `test.py` (2 points), and
- a brief "checkoff" conversation with a staff member about your code (1 point).

Note in order to receive credit for the lab, you will need to come (in-person) to any open lab time after the lab deadline has passed and add yourself to the queue asking for a checkoff.

It is a good idea to submit your lab early and often so that if something goes wrong near the deadline, we have a record of the work you completed before then. We recommend submitting your lab to the server after finishing each substantial portion of the lab, which will also display the results of our style checks.

> **Academic Integrity Policy**
>
> Please also review the academic integrity policies before continuing.

2) Introduction

Have you heard of *Six Degrees of Separation*? This simple theory states that at most 6 people separate you from any other person in the world.

Hollywood has its own version: Kevin Bacon is the center of the universe (not really, but let's let him feel good about himself). Every actor who has acted with Kevin Bacon in a movie is assigned a "Bacon number" of 1, every actor who acted with someone who acted with Kevin Bacon is given a "Bacon number" of 2, and so on. (What Bacon number does Kevin Bacon have? Think about it for a second.)

Note that if George Clooney acts in a movie with Julia Roberts, who has acted with Kevin Bacon in a different film, George has a Bacon number of 2 through this relationship. If George himself has also acted in a movie with Kevin, however, then his Bacon number is 1, and the connection through Julia is irrelevant. We define the notion of a "Bacon number" to be the *smallest* number of films separating a given actor (or actress) from Kevin Bacon.

In this lab, we will explore the notion of the Bacon number. We have prepared an ambitious database of approximately 37,000 actors and 10,000 films so that you may look up your favorites. Did Julia Roberts and Kevin Bacon act in the same movie? And what does Robert De Niro have to do with *Frozen*? Let's find out!

2.1) The Film Database

We've mined a large database of actors and films from IMDB via the www.themoviedb.org API. We present this data set to you as a list of records (3-element tuples), each of the form `(actor_id_1, actor_id_2, film_id)`, which tells us that `actor_id_2` acted with `actor_id_1` in a film denoted by `film_id`.

Keep in mind that "acts with" is a symmetric relationship. If `(a1, a2, f)` is in the database, it is true both that `a1` acted with `a2` *and* that `a2` acted with `a1`, even if `(a2, a1, f)` is not explicitly represented in the database.

However, these relationships do not necessarily exhibit the transitive property. That is, if `(a1, a2, f)` and `(a2, a3, f)` are in the database, it is *not necessarily true* that `a1` and `a3` have acted together (unless `(a1, a3, f)` or `(a3, a1, f)` is in the database explicitly). One way to think about this is that

"act together" might mean "appear on-screen together." So `a1` and `a3` may be in the same film, and each appear separately on-screen with `a2` at some point in the film, but `a1` and `a3` are never on-screen together at the same time.

We store these data as pickle files. The server tests will use `small.pickle` and `large.pickle`, but we have also included a `tiny.pickle` that you will use to write your own tests.

2.2) The Names Database

The functions in `lab.py` expect you to use integer actor IDs, but the tests we give you on this page will have actor names as inputs and outputs.

To help with this mapping, we include a file, `resources/names.pickle`, which contains a representation of the mapping between actor IDs and names. You can use the `load` function of Python's `pickle` module to get the data out of the file and into Python. We have included an example in the `if __name__ == '__main__'` section of `lab.py`.

Answer the following questions *using Python*. Even though these are small snippets, please include them in the `if __name__ == '__main__'` block in your submission and **be prepared to discuss them during your checkoff**.

Which of the following best describes the Python object that results from loading `resources/names.pickle`?

--

What is Samir Bannout's ID number?

Which actor has the ID 94500?

2.3) Using the UI

We have also provided a visualization website which loads your code into a small server (`server.py`) and visualizes your results. To use the visualization, run `python3 server.py` and use your web browser navigate to localhost:6101. You will need to restart `server.py` in order to reload your code if you make changes.

You will be able to see actors as circular nodes (hover above the node to see the actor's name) and the movies as edges linking nodes together.

Above the graph we define three different tabs, one for each component of the lab. Each tab sets up the visualization appropriate for its aspect of the lab.

2.4) `lab.py` and `test.py`

These files are yours to edit in order to complete this lab. You should implement the main functionality of the lab in `lab.py`, and you are strongly encouraged to implement additional test cases (as described throughout the assignment) in `test.py` as you're working.

In `lab.py`, you will find skeletons for most of the functions we expect you to write.

3) Transforming the Data

One thing that is worth noting is that the format in which these data are presented might not be a particularly useful format for the kinds of questions we're going to want to ask throughout the lab.

To this end, we have provided a skeleton for a function called `transform_data`. This function will take in the data in the same format that's stored in the Films database (described above), but you can use this function to create related data structures that might be more useful.

You can modify this function to create and return whatever structures you think will be useful, based on the data that are passed in.

The object returned by this function will not be tested directly (so you are free to choose whatever structure you want), but you should write your other functions under the assumption that the object associated with the `transformed_data` argument references what your `transform_data` function returns, not the raw data.

Why are we asking you to write the `transform_data` function separately from others? The answer is all about running time! You can make a clever representation choice and dramatically speed up the main functions of the lab. Sometimes it pays off to run a relatively slow transformation on data, to put them in a form that allows fast access later. "Fast" is relative to which operations you expect to perform frequently, so we recommend looking at the first couple of pieces of the lab (specifically, up to sections 4 and 5) before writing your first version of `transform_data`. You are, of course, free to return and improve `transform_data` as you progress through the assignment.

The important thing is that we only call `transform_data` once per movie database, so the work you do transforming can pay off to support multiple fast operations thereafter, in different tests!

4) Acting Together

To get used to the structure of the databases, complete the definition of `acted_together` in `lab.py`. This function should take three arguments in order:

- The database to be used (the result of calling your `transform_data` function on one of the databases)
- Two IDs representing actors

This function should return `True` if the two given actors ever acted together in a film and `False` otherwise. For example, Kevin Bacon (id `4724`) and Steve Park (id `4025`) did *not* act in a film together, meaning `acted_together(..., 4724, 4025)` should return `False`.

Note that we consider every actor to have acted with themselves, even if that relationship is not explicitly represented in the database.

In addition, add at least one test case to `test.py`, testing `acted_together` on the tiny database. Note that the `test.py` file loads the databases and stores them into variables such as `raw_db_tiny` and `db_tiny` (the transformed database) for you. **Be prepared to discuss any tests you add to `test.py` during your checkoff.**

In order to help with formulating your new tests, you can load the data from `tiny.pickle` in `lab.py` and print the results to see what it contains. Note that the best way to create these tests is to compute a couple of results by hand, then to check that your code produces the same result. For ease of testing, we recommend giving each new test a name that contains a unique substring, so that you can easily run only those tests using `pytest`'s `-k` flag. For example, since these tests will make use of the `tiny.pickle` database, you might consider prefixing each of these tests' names with `test_tiny_` or something like that.

When you are done implementing this function and it passes the associated tests, use your code to answer the following questions according to the data in the `resources/small.pickle` database. (Hint: You will also need to load `names.pickle`, to determine actor ID numbers from names.)

> According to the `small.pickle` database, have Tony Shalhoub and Daphne Rubin-Vega acted together?
> --

> According to the `small.pickle` database, have Jean-Marc Roulot and Noureddine El Ati acted together?
> --

Please note that `acted_together` is intended to provide a way to help you get familiar with the structure of the databases. You don't have to use the function in subsequent sections. Also note, though, that `test.py` does test this function.

5) Bacon Number

Next, we will try to find all of the actors who have a given Bacon number. We'll implement this as a function called `actors_with_bacon_number` in `lab.py`. This function should take two arguments in order:

- The database to be used (the result of calling your `transform_data` function)
- The desired Bacon number

This function should return a Python set containing the ID numbers of all the actors with that Bacon number. Note that we'll define the *Bacon number* to be the **smallest** number of films separating a given actor from Kevin Bacon, whose actor ID is `4724`.

Before we get to writing this function, we'll develop a couple of test cases that we can use to make sure

our function works properly (once we've written it!). Look at the data in `tiny.pickle` and answer these questions (by computing the responses manually from looking at the data):

> What are the **ID numbers** of the actors who have a Bacon number of 0 in `tiny.pickle`? Enter your answer below as a Python `set` of integers:

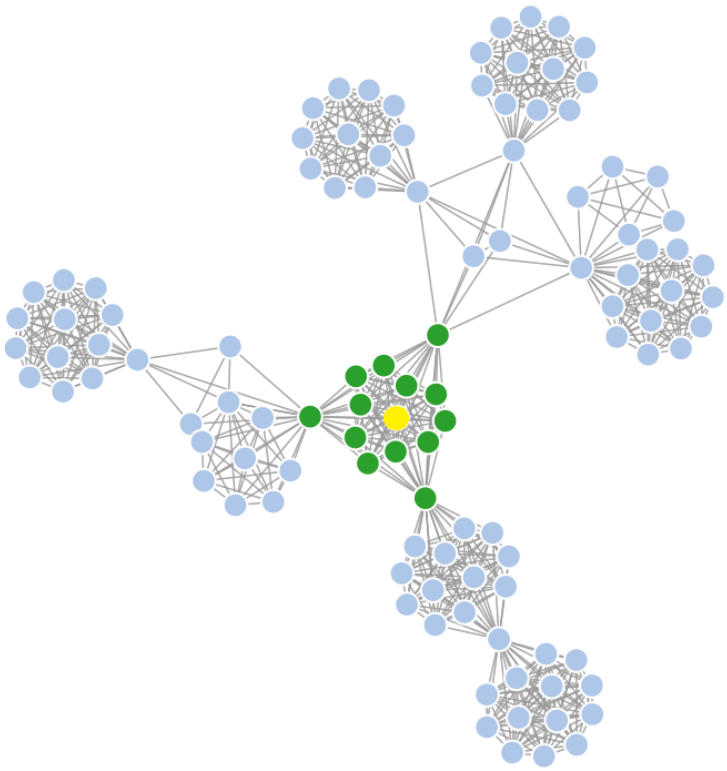
> What are the **ID numbers** of the actors who have a Bacon number of 1 in `tiny.pickle`? Enter your answer below as a Python `set` of integers:

> What are the **ID numbers** of the actors who have a Bacon number of 2 in `tiny.pickle`? Enter your answer below as a Python `set` of integers:

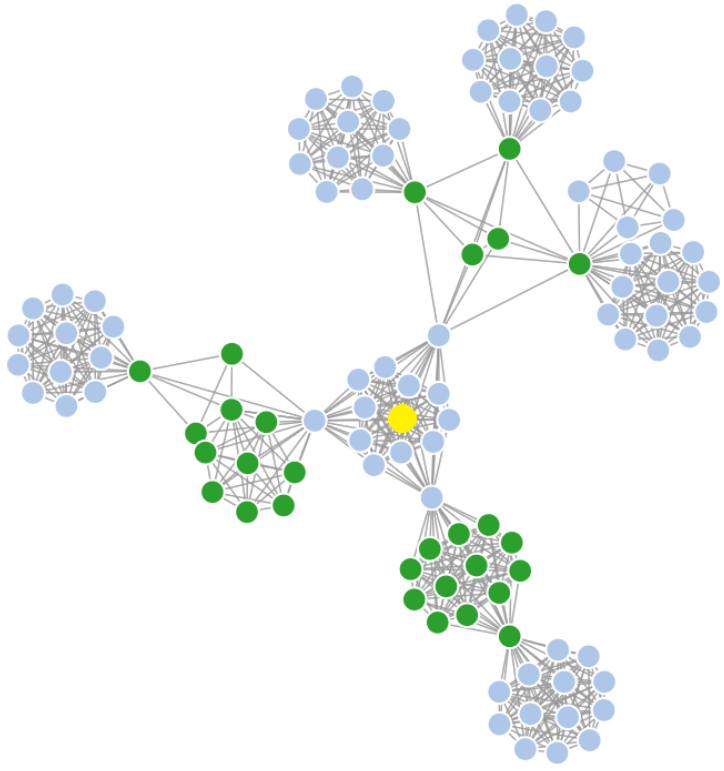
> What are the **ID numbers** of the actors who have a Bacon number of 3 in `tiny.pickle`? Enter your answer below as a Python `set` of integers:

Add the four above questions as tests in `test.py`. In each test case, use your `actors_with_bacon_number` function to compute the sets of actors with Bacon numbers 0, 1, 2, and 3 and compare them against the results you just found above. So that they are recognized as individual tests, **define each as a separate function, starting with `test_`** (e.g. `test_tiny_bacon_number_0`). **Be prepared to discuss these tests during your checkoff.**

Now you're ready to write your Bacon-number code! Here are some things to think about when writing your implementation. Consider the set of actors with a *Bacon number* of 1. Here is a visual representation of the data from the `small.pickle` database. (You can use our provided server to generate pictures like these.)



Given the set of actors with a *Bacon number* of 1, think of how you can find the set of actors with a *Bacon number* of 2:



Once you get a sense for how to get the *Bacon number* 2 actors from the *Bacon number* 1 actors, try to generalize to getting the *Bacon number* `i+1` actors from the *Bacon number* `i` actors.

Note that the test cases in `test.py` run against small and large databases of actors and films, and note that your implementation needs to be efficient enough to handle the large database in a timely manner. Your code should also handle the case of arbitrary Bacon numbers (not just $n \leq 6$) since some databases

may be structured to assign some actors quite large Bacon numbers.

When you're done writing this function and it passes all of your tests, answer the following question that uses the `large.pickle` database:

> In the `large.pickle` database, what is the set of actors with Bacon number 6? Enter your answer below as a Python `set` of strings representing **actor names**:

6) Paths

Now we'll turn our attention to finding the *chain* of actors that connects Kevin Bacon to someone else.

6.1) Bacon Paths

Complete the definition of `bacon_path` in `lab.py`. The function should take two arguments in order:

- The database to be used (the same structure as before)
- An ID representing an actor

Your function should produce a list of actor IDs (any such shortest list if there are several) detailing a "Bacon path" from Kevin Bacon to the actor denoted by `actor_id`. If no path exists, return `None`.

Please note that the paths are not necessarily unique, so any shortest list that connects Bacon to the actor denoted by `actor_id` is valid. The tester does not hard-code the correct paths and only verifies the *length* of the path you find (as well as that it is indeed a path that exists in the database).

For example, if we run this function on `large.pickle` with Julia Roberts's ID (`actor_id=1204`), one valid path is `[4724, 3087, 1204]`, showing that Kevin Bacon (`4724`) has acted with Robert Duvall (`3087`), who in turn acted with Julia Roberts (`1204`).

> Take a look at the data in `tiny.pickle` (by loading and printing it, not by running your path-finding code). From those data, you should be able to manually compute a couple of shortest paths. What's the shortest path that connects actor `4724` to actor `1640`? Enter your answer below as a Python list of **ID numbers**:

Add a test for the case above to `test.py`. You can use this test to help make sure your function is implemented correctly. **Be prepared to discuss this test during your checkoff.**

> If you are unsure about how to approach this part of the lab, this week's flood fill reading is a useful resource. What are the connections between our `flood_fill` implementation and path finding?

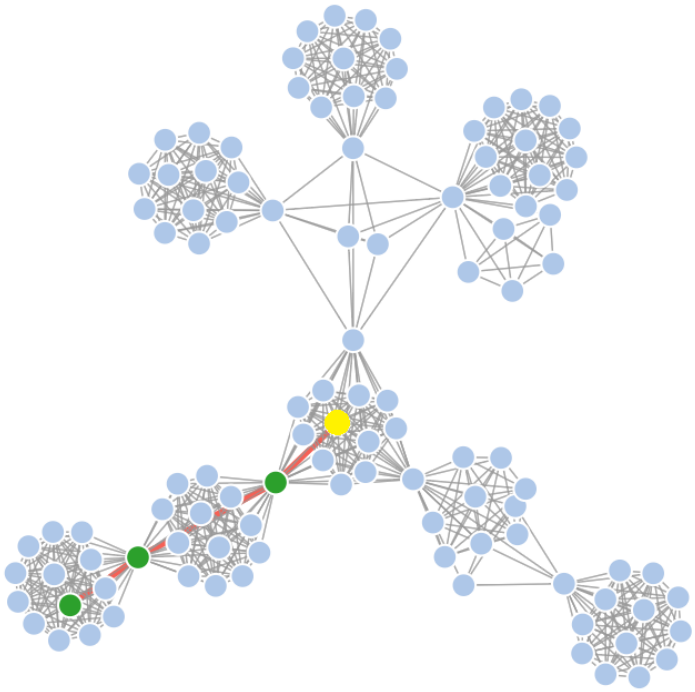
6.1.1) Speed

When implementing the path-finding algorithm, you should optimize your code to handle the large database, which our testing infrastructure will use when testing your code.

In particular, here are a few ideas about speed:

- Searching through data using a `for` loop can be slow. Can you reorganize the data so that your search can be implemented with a single dictionary lookup or set-containment check? If so, you could implement this change in your `transform_data` function.
- Membership tests (the `in` operator) on long lists can be very slow. By contrast, the `in` operator is very fast on sets and dictionaries (regardless of the lengths of these objects). However, sets and dictionaries do not retain information about the order of their elements. Consider whether there are cases in your code where a set or dictionary can be used in place of a list.

You will also need to be careful about your overall algorithm. In particular, ***avoid repeatedly iterating through all of `data`.*** For example, consider the following graph, with a path highlighted:



Here we've started from Kevin Bacon and successfully expanded out our search until we got to the actor we were looking for. What do we need to keep track of during our search if we want to get the path without looking for the actor again?

When you have implemented your function and it passes your tests, use it to answer the question below:

According to the `large.pickle` database, what is the path of actors from Kevin Bacon to Nilo Mur? Enter your answer as a Python list of **actor names** below:

6.2) Arbitrary Paths

What we've done so far is pretty good, but it raises an important question: what makes Kevin Bacon so special? So far, everything we've done has centered around him. Let's expand things a bit and find the path that connects two *arbitrary* actors to each other.

Complete the definition of `actor_to_actor_path` in `lab.py`. The function should take three arguments in order:

- The database to be used (the same structure as before)
- Two IDs representing actors

Your function should produce a list of actor IDs (any such shortest list if there are several) detailing a path from the first actor to the second.

Add at least one test case to `test.py`, based on the contents of the `tiny.pickle` database. It should find the minimal path between two non-Bacon actors. **Be prepared to discuss this test during your checkoff.**

When you have implemented this function and it passes your tests, use it to answer the question below. Even though some of these may be small code snippets, please include them in the `if __name__ == '__main__'` block of your `lab.py` and **be prepared to discuss them during your checkoff**.

> According to the `large.pickle` database, what is the minimal path of actors from Daniel McCabe to Jessica James? Enter your answer as a Python list of **actor names** below:
>
> _____

Now that you have implemented it, take a look at your definition of `actor_to_actor_path`. How does it compare to `bacon_path`? You may find that they are very similar; in that case, it is a good idea to "refactor" your code (rearrange it, perhaps introducing helper functions) to avoid repetitious code where possible. As you are refactoring, it is a good idea to continue testing your functions after each big change, to make sure they are still working as expected.

7) Movie Paths

After completing the work above, you might be interested to know what sequence of movies you could watch in order to traverse the path from one actor to another. For example, to move from Kevin Bacon to Julia Roberts, one could watch movie ID `94671` ("Jayne Mansfield's Car," which connects Kevin Bacon to Robert Duvall) and `18402` ("Something to Talk About," which connects Robert Duvall to Julia Roberts).

Add some new code to your `lab.py` to determine the list of *movie names* that connect two arbitrary actors. To this end, we have included the `movies.pickle` database, which maps movie names to ID numbers.

When you have finished this code, use it to answer the following question.

According to the `large.pickle` database, what is the minimal path of *movie titles* connecting Dustin Hoffman to Anton Radacic? Enter your answer as a Python list of **movie names** below:

8) Generalizing Our Path Finder

As it currently stands, our `actor_to_actor_path` function can currently only search for a particular actor. But it turns out that there are certain kinds of questions we could ask that are, unfortunately, difficult to ask given this implementation.

For example, suppose we want to find the shortest path from some actor to *any actor from a set of other actors*, or from some actor to *any actor in a particular movie*, or something like that.

To answer these kinds of questions, it may be helpful to generalize our notion of path-finding. Do so by filling in the definition of `actor_path` in `lab.py`. This function should take three arguments in order:

- The database to be used (the same structure as before),
- One actor ID to be used as our starting point, and
- A *function* to be used as our goal test. This function should take a single actor ID as input, and it should return `True` if that actor represents a valid ending location for the path, and `False` otherwise.

Your function should produce as output a list containing actor IDs, representing the shortest possible path from the given actor ID to *any actor that satisfies the goal-test function*. If no actors satisfy the goal condition, your function should instead return `None`.

Note that if the starting actor satisfies the goal test, your function should return a list containing only that actor's ID.

8.1) Movie-to-Movie Paths

As our final task for this lab, we would like for you to find chains of actors that connect one given *movie* to another. Implement this behavior by filling in the body of the `actors_connecting_films` function. This function should take three arguments:

- The database to be used (the same structure as before), and
- Two film ID numbers;

and it should return the shortest possible list of actor ID numbers (in order) that connect those two films. Your list should begin with the ID number of an actor who was in the first film, and it should end with the ID number of an actor who was in the second film.

If there is no path connecting those two films, your function should return `None`.

9) `test.py` Submission

Use the box below to upload your `test.py` file so that we can discuss it during your checkoff. This submission will not check your test cases for correctness, so it is worth double-checking that you've

submitted the correct file.

Select File No file selected

10) Code Submission

> **Automated Style Feedback**
>
> Starting with this lab, we will be running your code through an automated system looking for style issues by running Pylint on your code in addition to running a couple of other kinds of analysis. When you submit your code, you will receive feedback not only on passing the test cases but also on the results of this style check.
>
> For labs released after the midterm, these checks will factor into your grade for the lab in place of the checkoff conversations for the labs before the midterm. But for now, the style feedback from the automated checks is purely advisory.

When you have tested your code sufficiently on your own machine, submit your modified `lab.py` using the `6.101-submit` script.

The following command should submit the lab, assuming that the last argument `/path/to/lab.py` is replaced by the location of your `lab.py` file:

```
$ 6.101-submit -a bacon /path/to/lab.py
```

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

Running that script should submit your file to be checked. After submitting your file, information about the checking process can be found below:

When this page was loaded, you had not yet made any submissions.
If you make a submission, results should show up here automatically; or you may click here or reload the page to see updated results.

11) Checkoff

Once you are finished with the code, you will need to come (in-person) to any open lab time and add yourself to the queue asking for a checkoff in order to receive credit for the lab. **You must be ready to discuss your code in detail before asking for a checkoff.**

You should be prepared to demonstrate your code (which should be well-commented, should avoid

repetition, and should make good use of helper functions; see the notes on style for more information). In particular, be prepared to discuss:

- Your implementation of `transform_data`. Why did you choose this structure to work with?
- The additional test cases you added to `test.py`.
- Your implementation of `acted_together`.
- Your implementation of `actors_with_bacon_number`.
- How did you handle large values of `n` in `actors_with_bacon_number`?
- Your implementation of `actor_path`, `actor_to_actor_path`, and `bacon_path`, and your test cases for `actor_to_actor_path`.
- How you transformed actor/movie names into ID numbers, and *vice versa*.
- The additional code you wrote to compute the paths of actor/movie names.
- Your implementation of `actors_connecting_films`

> *You have not yet received this checkoff. When you have completed this checkoff, you will see a grade here.*