Lab 4: Lists and Data Abstraction

lab04.zip (lab04.zip)

Due at 11:59pm on Friday, 09/21/2018.

Lab Check-in 1 questions here (/lab-check-in/lab-check-in01/).

Starter Files

Download lab04.zip (lab04.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (ok) autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

- To receive credit for this lab, you must complete Questions 1-6 in lab04.py (lab04.py) and submit through OK.
- Questions 7-15 are **optional**. It can be found in the lab04_extra.py (lab04_extra.py) file. It is recommended that you complete these problems on your own time.

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Lists

Lists are Python data structures that can store multiple values. Each value can be any type and can even be another list! A list is written as a comma separated list of expressions within square brackets:

```
>>> list_of_nums = [1, 2, 3, 4]
>>> list_of_bools = [True, True, False, False]
>>> nested_lists = [1, [2, 3], [4, [5]]]
```

Each element in a list is assigned an index. Lists are *zero-indexed*, meaning their indices start at `0` and increase in sequential order. To retrieve an element from a list, use list indexing:

```
>>> lst = [6, 5, 4, 3, 2, 1]
>>> lst[0]
6
>>> lst[3]
3
```

Often times we need to know how long a list is when we're working with it. To find the length of a list, call the function `len` on it:

```
>>> len([])
0
>>> len([2, 4, 6, 8, 10])
5
```

> **Tip:** Recall that empty lists, `[]`, are false-y values. Therefore, you can use an if statement like the following if you only want to do operations on non-empty lists:
>
> ```
> if lst:
>     # Do stuff with the elements of list
> ```
>
> This is equivalent to:
>
> ```
> if len(lst) > 0:
>     # Do stuff
> ```

You can also create a copy of some portion of the list using list slicing. To slice a list, use this syntax: `lst[<start index>:<end index>]`. This expression evaluates to a new list containing the elements of `lst` starting at and including the element at `<start index>` up to but not including the element at `end index`.

```
>>> lst = [True, False, True, True, False]
>>> lst[1:4]
[False, True, True]
>>> lst[:3]  # Start index defaults to 0
[True, False, True]
>>> lst[3:]  # End index defaults to len(lst)
[True, False]
>>> lst[:]  # Creates a copy of the whole list
[True, False, True, True, False]
```

List Comprehensions

List comprehensions are a compact and powerful way of creating new lists out of sequences. The general syntax for a list comprehension is the following:

```
[<expression> for <element> in <sequence> if <conditional>]
```

The syntax is designed to read like English: *"Compute the expression for each element in the sequence if the conditional is true for that element."*

Let's see it in action:

```
>>> [i**2 for i in [1, 2, 3, 4] if i % 2 == 0]
[4, 16]
```

Here, for each element `i` in `[1, 2, 3, 4]` that satisfies `i % 2 == 0`, we evaluate the expression `i**2` and insert the resulting values into a new list. In other words, this list comprehension will create a new list that contains the square of each of the even elements of the original list.

If we were to write this using a for statement, it would look like this:

```
>>> lst = []
>>> for i in [1, 2, 3, 4]:
...     if i % 2 == 0:
...         lst += [i**2]
>>> lst
[4, 16]
```

Note: The `if` clause in a list comprehension is optional. For example, you can just say:

```
>>> [i**2 for i in [1, 2, 3, 4]]
[1, 4, 9, 16]
```

Data Abstraction

Data abstraction is a powerful concept in computer science that allows programmers to treat code as objects -- for example, car objects, chair objects, people objects, etc. That way, programmers don't have to worry about *how* code is implemented -- they just have to know *what* it does.

Data abstraction mimics how we think about the world. When you want to drive a car, you don't need to know how the engine was built or what kind of material the tires are made of. You just have to know how to turn the wheel and press the gas pedal.

An abstract data type consists of two types of functions:

- **Constructors**: functions that build the abstract data type.
- **Selectors**: functions that retrieve information from the data type.

Programmers design ADTs to abstract away how information is stored and calculated such that the end user does *not* need to know how constructors and selectors are implemented. The nature of *abstract* data types allows whoever uses them to assume that the functions have been written correctly and work as described.

Required Questions

Lists Practice

Q1: List Indexing

Use Ok to test your knowledge with the following "List Indexing" questions:

```
python3 ok -q indexing -u
```

For each of the following lists, what is the list indexing expression that evaluates to `7`? For example, if `x = [7]`, then the answer would be `x[0]`. You can use the interpreter or Python Tutor to experiment with your answers.

```
>>> x = [1, 3, [5, 7], 9]
______

>>> x = [[7]]
______

>>> x = [3, 2, 1, [9, 8, 7]]
______

>>> x = [[3, [5, 7], 9]]
______
```

Toggle Solution

What would Python display? If you get stuck, try it out in the Python interpreter!

```
>>> lst = [3, 2, 7, [84, 83, 82]]
>>> lst[4]
______

>>> lst[3][0]
______
```

Toggle Solution

Q2: WWPD: Lists?

What would Python display? Try to figure it out before you type it into the interpreter!

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q lists -u
```

```
>>> [x*x for x in range(5)]
______

>>> [n for n in range(10) if n % 2 == 0]
______

>>> ones = [1 for i in ["hi", "bye", "you"]]
>>> ones + [str(i) for i in [6, 3, 8, 4]]
______

>>> [i+5 for i in [n for n in range(1,4)]]
______
```

Toggle Solution

```
>>> [i**2 for i in range(10) if i < 3]
______

>>> lst = ['hi' for i in [1, 2, 3]]
>>> print(lst)
______

>>> lst + [i for i in ['1', '2', '3']]
______
```

Toggle Solution

Q3: If This Not That

Define `if_this_not_that`, which takes a list of integers `i_list` and an integer `this`. For each element in `i_list`, if the element is larger than `this`, then print the element. Otherwise, print `"that"`.

```
def if_this_not_that(i_list, this):
    """Define a function which takes a list of integers `i_list` and an integer
    `this`. For each element in `i_list`, print the element if it is larger
    than `this`; otherwise, print the word "that".

    >>> original_list = [1, 2, 3, 4, 5]
    >>> if_this_not_that(original_list, 3)
    that
    that
    that
    4
    5
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q if_this_not_that
```

City Data Abstraction

Say we have an abstract data type for cities. A city has a name, a latitude coordinate, and a longitude coordinate.

Our ADT has one **constructor**:

- `make_city(name, lat, lon)`: Creates a city object with the given name, latitude, and longitude.

We also have the following **selectors** in order to get the information for each city:

- `get_name(city)`: Returns the city's name
- `get_lat(city)`: Returns the city's latitude
- `get_lon(city)`: Returns the city's longitude

Here is how we would use the constructor and selectors to create cities and extract their information:

```
>>> berkeley = make_city('Berkeley', 122, 37)
>>> get_name(berkeley)
'Berkeley'
>>> get_lat(berkeley)
122
>>> new_york = make_city('New York City', 74, 40)
>>> get_lon(new_york)
40
```

All of the selector and constructor functions can be found in `city.py`, if you are curious to see how they are implemented. However, the point of data abstraction is that we do not need to know how an abstract data type is implemented, but rather just how we can interact with and use the data type.

Q4: Distance

We will now implement the function `distance`, which computes the distance between two city objects. Recall that the distance between two coordinate pairs `(x1, y1)` and `(x2, y2)` can be found by calculating the `sqrt` of `(x1 - x2)**2 + (y1 - y2)**2`. We have already imported `sqrt` for your convenience. Use the latitude and longitude of a city as its coordinates; you'll need to use the selectors to access this info!

```
from math import sqrt
def distance(city1, city2):
    """
    >>> city1 = make_city('city1', 0, 1)
    >>> city2 = make_city('city2', 0, 2)
    >>> distance(city1, city2)
    1.0
    >>> city3 = make_city('city3', 6.5, 12)
    >>> city4 = make_city('city4', 2.5, 15)
    >>> distance(city3, city4)
    5.0
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q distance
```

Q5: Closer city

Next, implement `closer_city`, a function that takes a latitude, longitude, and two cities, and returns the name of the city that is relatively closer to the provided latitude and longitude.

You may only use the selectors and constructors introduced above and the `distance` function you just defined for this question.

> **Hint**: How can use your `distance` function to find the distance between the given location and each of the given cities?

```
def closer_city(lat, lon, city1, city2):
    """
    Returns the name of either city1 or city2, whichever is closest to
    coordinate (lat, lon).

    >>> berkeley = make_city('Berkeley', 37.87, 112.26)
    >>> stanford = make_city('Stanford', 34.05, 118.25)
    >>> closer_city(38.33, 121.44, berkeley, stanford)
    'Stanford'
    >>> bucharest = make_city('Bucharest', 44.43, 26.10)
    >>> vienna = make_city('Vienna', 48.20, 16.37)
    >>> closer_city(41.29, 174.78, bucharest, vienna)
    'Bucharest'
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q closer_city
```

Q6: Don't violate the abstraction barrier!

When writing functions that use an ADT, we should use the constructor(s) and selector(s) whenever possible instead of assuming the ADT's implementation. Relying on a data abstraction's underlying implementation is known as *violating the abstraction barrier*, and we never want to do this!

It's possible that you passed the doctests for `distance` and `closer_city` even if you violated the abstraction barrier. To check whether or not you did so, uncomment the following lines in your `lab04.py` file:

```
# make_city = lambda name, lat, lon: { 'name': name, 'lat': lat, 'lon': lon }
# get_name = lambda city: city['name']
# get_lat = lambda city: city['lat']
# get_lon = lambda city: city['lon']
```

These statements change the implementation of the city ADT. The nature of the abstraction barrier guarantees that changing the implementation of an ADT shouldn't affect the functionality of any programs that use that ADT, as long as the constructors and selectors were used properly.

Now, rerun your tests for `distance` and `closer_city` *without changing any of your code*:

```
python3 ok -q distance
python3 ok -q closer_city
```

If you've followed the rules and used the constructor and selectors when you should've, the doctests should still pass!

If you passed the Ok tests before uncommenting those lines but not afterward, the fix is simple! Just replace any code that violates the abstraction barrier, i.e. creating a city with a new list object or indexing into a city, with the appropriate constructor or selector.

Make sure that your functions pass the tests with both the first and the second implementations of the City ADT and that you understand why they should work for both before moving on.

Optional Questions

All these questions can be found in `lab04_extra.py`.

More Lists Practice

Q7: Flatten

Write a function `flatten` that takes a (possibly deep) list and "flattens" it. For example:

```
>>> lst = [1, [[2], 3], 4, [5, 6]]
>>> flatten(lst)
[1, 2, 3, 4, 5, 6]
```

Hint: you can check if something is a list by using the built-in `type` function. For example,

```
>>> type(3) == list
False
>>> type([1, 2, 3]) == list
True
```

```
def flatten(lst):
    """Returns a flattened version of lst.

    >>> flatten([1, 2, 3])     # normal list
    [1, 2, 3]
    >>> x = [1, [2, 3], 4]      # deep list
    >>> flatten(x)
    [1, 2, 3, 4]
    >>> x = [[1, [1, 1]], 1, [1, 1]] # deep list
    >>> flatten(x)
    [1, 1, 1, 1, 1, 1]
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q flatten
```

Q8: Merge

Write a function `merge` that takes 2 *sorted* lists `lst1` and `lst2`, and returns a new list that contains all the elements in the two lists in sorted order.

```
def merge(lst1, lst2):
    """Merges two sorted lists.

    >>> merge([1, 3, 5], [2, 4, 6])
    [1, 2, 3, 4, 5, 6]
    >>> merge([], [2, 4, 6])
    [2, 4, 6]
    >>> merge([1, 2, 3], [])
    [1, 2, 3]
    >>> merge([5, 7], [2, 4, 6])
    [2, 4, 5, 6, 7]
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q merge
```

Connect N

You've probably heard of Connect 4 (https://www.wikiwand.com/en/Connect_Four), a two-player game where the players take turns dropping a colored piece from the top of a column in a grid. The piece ends at the last empty spot in this column - that is, as close to the bottom as possible. A player can only put pieces in columns with open spaces.

The winner is the first player who gets N of their pieces next to each other - either horizontally, vertically or diagonally. The game ends at this point, or as soon as the board is full.

We can generalize this game so that the goal is to connect N pieces instead of just 4. In this section, we will be implementing a command line version of Connect N!

Building Connect N

Let's build the combat field for players `'X'` and `'O'`.

In this lab, we will represent the playing board as a list of lists. We call such a list two-dimensional because we can visualize it as a rectangle. For instance, this list:

```
`[['-', '-', '-', '-'], ['O', 'O', 'O', 'X'], ['X', 'X', 'X', 'O']]`
```

would represent the following board:

```
- - - -
O O O X
X X X O
```

What does the number of nested lists represent? What about the number of elements in each nested list? When you have made up your mind, you are ready to build the board!

Notice that just like lists are zero-indexed, our board is zero-indexed. This means that the columns and rows in the above board would be numbered like this:

```
0  - - - -
1  O O O X
2  X X X O
   0 1 2 3
```

Q9: Creating an empty board

We are going to use data abstraction as we build our game, so let's start by making the constructors. We will represent an empty spot by the string `'-'`. In `lab04_extra.py`, fill out the *constructors*.

First, implement the function `create_row`, which returns one empty row in our board according to our abstraction (i.e., a row is one list).

This function should consist of a one-line return statement.

> *Hint*: You can create a list in one line using a list comprehension.

```
def create_row(size):
    """Returns a single, empty row with the given size. Each empty spot is
    represented by the string '-'.

    >>> create_row(5)
    ['-', '-', '-', '-', '-']
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q create_row
```

Then, use `create_row` to implement `create_board`, which returns a board with the specified dimensions.

This function should consist of a one-line return statement.

```
def create_board(rows, columns):
    """Returns a board with the given dimensions.

    >>> create_board(3, 5)
    [['-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-'], ['-', '-', '-', '-', '-']]
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q create_board
```

Q10: Updating the board

Over the course of a game, the board will change and we will need to keep our representation of the board up-to-date. To do so, we will be creating a new board that represents the new state of the game every time a piece is played. Implement `replace_elem`, which takes a list, an index, and an element to be placed at that index in the returned **new list**.

This function should consist of a one-line return statement.

```
def replace_elem(lst, index, elem):
    """Create and return a new list whose elements are the same as those in
    LST except at index INDEX, which should contain element ELEM instead.

    >>> old = [1, 2, 3, 4, 5, 6, 7]
    >>> new = replace_elem(old, 2, 8)
    >>> new
    [1, 2, 8, 4, 5, 6, 7]
    >>> new is old   # check that replace_elem outputs a new list
    False
    """
    assert index >= 0 and index < len(lst), 'Index is out of bounds'
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q replace_elem
```

Q11: Manipulating pieces

Now that we have the board ready, let's make our selectors! First, we need a way to find out which piece (`'-'`, `'X'` or `'O'`) is at a given position. Implement `get_piece` so it does this.

> *Note*: Because `get_piece` is a *selector*, it is allowed to break through the data abstraction barrier. This means that it is aware that the board is implemented as a list and can use list operations to update it. This allows us to abstract away the inner implementation for all of the other functions that both the programmer and other users will use.

This function should consist of a one-line return statement.

```
def get_piece(board, row, column):
    """Returns the piece at location (row, column) in the board.

    >>> rows, columns = 2, 2
    >>> board = create_board(rows, columns)
    >>> board = put_piece(board, rows, 0, 'X')[1] # Puts piece "X" in column 0 of board ar
    >>> board = put_piece(board, rows, 0, 'O')[1] # Puts piece "O" in column 0 of board ar
    >>> get_piece(board, 1, 0)
    'X'
    >>> get_piece(board, 1, 1)
    '-'
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Right now, all spots in our board are empty, so the output of `get_piece` won't be very interesting - and neither will the game. Let's change that! Go ahead and implement `put_piece`, which places the given `player`'s piece in the given `column`. `put_piece` should return a 2-element tuple that contains (`<row index>`, `<new board>`). The first element is the index of the row the piece ends up in, or -1 if the column is already full. The second element is the new board after the piece has been placed. If the column was full then just return the original `board`.

Assume that the given column is on the board. Remember that you can get pieces in the board by using `get_piece`. The argument `max_rows` may be helpful in determining which rows you should check for an empty slot to put the piece in.

> *Hint*: You will probably need to use the `replace_elem` function you wrote above *twice* to create the new board.

```
def put_piece(board, max_rows, column, player):
    """Puts PLAYER's piece in the bottommost empty spot in the given column of
    the board. Returns a tuple of two elements:

        1. The index of the row the piece ends up in, or -1 if the column
           is full.
        2. The new board

    >>> rows, columns = 2, 2
    >>> board = create_board(rows, columns)
    >>> row, new_board = put_piece(board, rows, 0, 'X')
    >>> row
    1
    >>> row, new_board = put_piece(new_board, rows, 0, 'O')
    >>> row
    0
    >>> row, new_board = put_piece(new_board, rows, 0, 'X')
    >>> row
    -1
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

You must complete both functions in order to test them. Tests for `get_piece` will not pass unless you've implemented `put_piece` correctly.

Use Ok to test your code:

```
python3 ok -q get_piece
python3 ok -q put_piece
```

You are now crossing the abstraction barrier!!!

You have now implemented the *constructor* and *selectors* as well as ways to modify the attributes of your abstract data type, the board. From now on, you should never need to treat the board as if it were a list. Instead, trust your abstraction barrier and use the functions you have written so far.

Q12: Making a move

Let's first write a function for players to make a move in the game. This is different from the `put_piece` function above in that `put_piece` assumes that the player gives a valid column number. `make_move` should only place the piece on the board if the given column is actually on the board. It returns a 2-element tuple (row index, board).

If the move is valid, put a piece in the column and return the index of the row the piece ends up in (do you have a function that will help you do this?) as well as the new board. If the move is invalid, `make_move` should return -1 and the original board, unchanged.

The arguments `max_rows` and `max_cols` describe the dimensions of the board and may be useful in determining whether or not a move is valid.

```
def make_move(board, max_rows, max_cols, col, player):
    """Put player's piece in column COL of the board, if it is a valid move.
    Return a tuple of two values:

        1. If the move is valid, make_move returns the index of the row the
           piece is placed in. Otherwise, it returns -1.
        2. The updated board

    >>> rows, columns = 2, 2
    >>> board = create_board(rows, columns)
    >>> row, board = make_move(board, rows, columns, 0, 'X')
    >>> row
    1
    >>> get_piece(board, 1, 0)
    'X'
    >>> row, board = make_move(board, rows, columns, 0, 'O')
    >>> row
    0
    >>> row, board = make_move(board, rows, columns, 0, 'X')
    >>> row
    -1
    >>> row, board = make_move(board, rows, columns, -4, '0')
    >>> row
    -1
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q make_move
```

Q13: Printing and viewing the board

Wouldn't it be great if we could actually see the board and the pieces on it? Let's now write a function to help us do that.

The function `print_board` takes in a board (as defined by our abstraction) and the dimensions of the board, and it prints out the current state of the board.

We would like our board to look good, and for this, strings do a better job than lists. Thus, we would like the row `['X', 'X', 'O', '-']` to be printed as `'X X O -'` where the pieces are separated by a single blank space. Remember that you can concatenate strings with the `+` operator, e.g. `'hel' + 'lo' = 'hello'`.

Remember that we're still on the other side of the abstraction barrier, and *you must implement this function as if we didn't know the board is a list of lists*. This is called respecting the data abstraction barrier. Specifically, to get information about `board`, you should use the selectors you've implemented instead of indexing into it.

> *Hint*: You might find that you're failing doctests that seem to match your output. Chances are that you have an extra space character at the end of your rows in your board. A function that might come in handy is `strip()`, which removes leading and trailing whitespace from a string. For example:
>
> ```
> >>> s = '   hello '
> >>> s.strip()
> 'hello'
> ```

```
def print_board(board, max_rows, max_cols):
    """Prints the board. Row 0 is at the top, and column 0 at the far left.

    >>> rows, columns = 2, 2
    >>> board = create_board(rows, columns)
    >>> print_board(board, rows, columns)
    - -
    - -
    >>> new_board = make_move(board, rows, columns, 0, 'X')[1]
    >>> print_board(new_board, rows, columns)
    - -
    X -
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q print_board
```

Now we can actually play the game! To try it out, run the following command in the terminal from your `lab04` directory:

```
$ python3 -i lab04_extra.py
>>> start_game()
```

Q14: Checking for victory

Fun, right? At long as you don't care about winning... The last thing we need for our Connect N game to be fully functioning is the ability to detect a win.

First, let's implement two helper functions `check_win_row` and `check_win_col` that check for horizontal and vertical wins *for the given player*.

Since we check for wins after each turn, and only the player who made the most recent move can have a win, `check_win_row` and `check_win_col` should only check for a win for the player that is passed as an argument. Also remember that `num_connect` tells you how many adjacent pieces are needed for a win. The arguments `max_rows` and `max_cols` describe the dimensions of the game board.

As in `print_board`, use the data abstractions you just built.

```
def check_win_row(board, max_rows, max_cols, num_connect, row, player):
    """ Returns True if the given player has a horizontal win
    in the given row, and otherwise False.

    >>> rows, columns, num_connect = 4, 4, 2
    >>> board = create_board(rows, columns)
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> board = make_move(board, rows, columns, 0, 'O')[1]
    >>> check_win_row(board, rows, columns, num_connect, 3, 'O')
    False
    >>> board = make_move(board, rows, columns, 2, 'X')[1]
    >>> board = make_move(board, rows, columns, 0, 'O')[1]
    >>> check_win_row(board, rows, columns, num_connect, 3, 'X')
    False
    >>> board = make_move(board, rows, columns, 1, 'X')[1]
    >>> check_win_row(board, rows, columns, num_connect, 3, 'X')
    True
    >>> check_win_row(board, rows, columns, 4, 3, 'X')    # A win depends on the value of
    False
    >>> check_win_row(board, rows, columns, num_connect, 3, 'O')   # We only detect wins
    False
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q check_win_row
```

```
def check_win_column(board, max_rows, max_cols, num_connect, col, player):
    """ Returns True if the given player has a vertical win in the given column,
    and otherwise False.

    >>> rows, columns, num_connect = 5, 5, 2
    >>> board = create_board(rows, columns)
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> board = make_move(board, rows, columns, 1, 'O')[1]
    >>> check_win_column(board, rows, columns, num_connect, 0, 'X')
    False
    >>> board = make_move(board, rows, columns, 1, 'X')[1]
    >>> board = make_move(board, rows, columns, 1, 'O')[1]
    >>> check_win_column(board, rows, columns, num_connect, 1, 'O')
    False
    >>> board = make_move(board, rows, columns, 2, 'X')[1]
    >>> board = make_move(board, rows, columns, 1, 'O')[1]
    >>> check_win_column(board, rows, columns, num_connect, 1, 'O')
    True
    >>> check_win_column(board, rows, columns, 4, 1, 'O')
    False
    >>> check_win_column(board, rows, columns, num_connect, 1, 'X')
    False
    """
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q check_win_column
```

Q15: Winning!

Finally, let's implement a way to check for any wins. Implement `check_win` so that it returns `True` if there is a win in any direction - that is, horizontally, vertically or diagonally.

You should use the functions you just wrote, `check_win_row` and `check_win_column`, along with the provided function `check_win_diagonal(board, max_rows, max_cols, num_connect, row, col, player)`, which returns `True` if the given player has a diagonal win passing the spot (row, column) and `False` otherwise.

```
def check_win(board, max_rows, max_cols, num_connect, row, col, player):
    """Returns True if the given player has any kind of win passing through
    (row, col), and False otherwise.

    >>> rows, columns, num_connect = 2, 2, 2
    >>> board = create_board(rows, columns)
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> board = make_move(board, rows, columns, 1, 'O')[1]
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> check_win(board, rows, columns, num_connect, 0, 0, 'O')
    False
    >>> check_win(board, rows, columns, num_connect, 0, 0, 'X')
    True

    >>> board = create_board(rows, columns)
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> board = make_move(board, rows, columns, 0, 'O')[1]
    >>> board = make_move(board, rows, columns, 1, 'X')[1]
    >>> check_win(board, rows, columns, num_connect, 1, 0, 'X')
    True
    >>> check_win(board, rows, columns, num_connect, 0, 0, 'X')
    False

    >>> board = create_board(rows, columns)
    >>> board = make_move(board, rows, columns, 0, 'X')[1]
    >>> board = make_move(board, rows, columns, 1, 'O')[1]
    >>> board = make_move(board, rows, columns, 1, 'X')[1]
    >>> check_win(board, rows, columns, num_connect, 0, 0, 'X')
    False
    >>> check_win(board, rows, columns, num_connect, 1, 0, 'X')
    True
    """
    diagonal_win = check_win_diagonal(board, max_rows, max_cols, num_connect,
                                      row, col, player)
    "*** YOUR CODE HERE ***"
```

Toggle Solution

Use Ok to test your code:

```
python3 ok -q check_win
```

Congratulations, you just built your own Connect N game! Don't you think the person next to you would be down for a game? As before, run the game by running the following command in your terminal:

```
$ python3 -i lab04.py
>>> start_game()
```

We implemented the `play` function for you, but if you are curious, you should take a look at it. As you will see, thanks to the layers of data abstraction, the `play` function is actually very simple. Notice how we use your `make_move`, `print_board`, and `check_win` to play the game without even knowing how the board and pieces are implemented.

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)