hw00-python-exercises

August 23, 2026

1 Python exercises

1.1 Write Your Name Here:

2 Instructions

In this assignment, your core Python skills will be tested by having you complete or write various functions and classes using concepts such as lists, dictionaries, loops, and recursion. Write code in the sections marked with *# YOUR CODE HERE*.

Do not import any libraries, other than the ones already imported in the Utilities section below!

For each exercise, you will be given a set of instructions and a test function denoted with the prefix `TEST`. The testing function will contain a `todo_check()` function which will give you a rough estimate of whether your code is functioning as excepted. Feel free to write your own tests to make sure your code works with various different inputs.

3 Submission instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Edit -> Clear Output of All Cells. This will clear all the outputs from all cells (but will keep their content).
4. Select Run -> Run All Cells. This will run all the cells in order, and may take several seconds.
5. Once you've rerun everything, select File -> Save and Export Notebook As -> PDF and download a PDF version showing the code and the output of all cells, and save it in the same folder that contains the notebook file.
6. Look at the PDF file and make sure all your solutions and outputs are there, displayed correctly.
7. Submit **both** the PDF and your notebook file .ipynb on Canvas. If you encounter issues conveting to PDF (make sure 'nbconvert' is installed first), saving as HTML is acceptable too (PDF is preferable).
8. Make sure your your Canvas submission contains the correct files by downloading it after posting it on Canvas.

3.1 Virtual Environments and Packages

It is recommended that you install the packages required for this class in a *virtual environment*, for which I recommend the Python virtual environment tool `venv` (other package managers exist that you can use, such as 'conda'). To use `venv` (to manage the environment) and `pip` (to install packages) please see this quick guide. You can first create a folder named "itcs5356" for the course, inside which you will place all your homework assignment folders. A prototypical setup in a Unix-based system looks like this:

1. *mkdir itcs5356* # this creates the course folder.
2. *cd itcs5356* # this changes the current directory to that folder.
3. *python3 -m venv .venvml* # this creates the virtual environment named 'venvml' (you can use any name you want).
4. *source .venvml/bin/activate* # this activates the virtual environment 'venvml'. After this, anything you isntall with pip will be installed in this environment.
5. *python3 -m pip install –upgrade pip* # this installs the 'pip' package manager.
6. *pip install jupyterlab* # this installs the 'jupyterlab' package.
7. *pip install nbconvert* # this isntalls the 'nbconvert' package, needed to convert notebooks to PDF.

We will use 'jupyter-lab' to edit the notebooks for the homework assignments. You can similarly use the 'pip' command (inside the activated '.venvml' environment) to install other packages required for future homework assignments.

4 Python Exercises

4.1 List Centering (10 points)

Complete the `center_list()` function, which should take in a list `x` as input, and subtract the mean of `x` from every element in `x`. Be sure to store the mean of `x` in the variable `mean` and the centered list in the variable `centered_list`. Both `mean` and `centered_list` should be returned! Run the `TEST_center_list()` function to test your code.

Example

The output for the list `[5, 10, 15, 20, 25]` should be a mean of 15 and the new list `[-10.0, -5.0, 0.0, 5.0, 10.0]`

[ ]:
```
def center_list(x):
    mean = None
    centered_list = []
    # YOUR CODE HERE



    return mean, centered_list
```

```
[ ]: def TEST_center_list():
         print("{:=^50}".format('Inputs'))
         x = [5, 10, 15, 20, 25]
         print(f"x: {x}")

         print("{:=^50}".format('Outputs'))
         mean, centered_list = center_list(x)
         print(f"Mean: {mean}")
         print(f"Centered list: {centered_list}")

         assert mean == 15.0
         assert centered_list ==  [-10.0, -5.0, 0.0, 5.0, 10.0]

     TEST_center_list()
```

4.2 Matrix Replacement (10 points)

Complete the `matrix_replacement()` function, which should take in a list of lists `X` (representing a matrix) as input. The goal of the `matrix_replacement()` function is to replace (i.e., filter) all `None` values with the integer value of 0. If you edited `X` directly or created a new matrix with the replaced values, be sure to return it. Run the `TEST_matrix_replacement()` function to test your code.

Example

The output for the list `[[1, None, 2], [None, 3, 1], [5, 6, None]]` should be `[[1, 0, 2], [0, 3, 1], [5, 6, 0]]`

```
[ ]: def matrix_replacement(X):
         # YOUR CODE HERE




         return X
```

```
[ ]: def TEST_matrix_replacement():
         print("{:=^50}".format('Inputs'))
         X = [[1, None, 2],[None, 3, 1],[5, 6, None]]
         print(f"x: {X}")

         print("{:=^50}".format('Outputs'))
         results = matrix_replacement(X)
         print(f"Results: {results}")

         assert results == [[1, 0, 2], [0, 3, 1], [5, 6, 0]]
```

```
TEST_matrix_replacement()
```

4.3 Histogram (10 points)

Complete the `matrix_histogram(x)` function, which should take in a list `x` as input. The goal of the `make_histogram(x)` function is to output a Python dictionary mapping each unique integer in `x` to the number of times it appears in `x`. Be sure to store the resulting histogram into the `result` variable and to return it. Run the `TEST_make_histogram()` function to test your code.

Example

The output for the list `[1, 2, 1, 3, 2, 1, 6, 2, 6, 1]` should be `{1: 4, 2: 3, 3: 1, 6: 2}`

[ ]:
```
def make_histogram(x):
    result = {}
    # YOUR CODE HERE




    return result
```

[ ]:
```
def TEST_make_histogram():
    print("{:=^50}".format('Inputs'))
    x = [1, 2, 1, 3, 2, 1, 6, 2, 6, 1]
    print(f"x: {x}")

    print("{:=^50}".format('Outputs'))
    results = make_histogram(x)
    print(f"Result: {results}")

    assert isinstance(results, dict)
    assert results == {1:4, 2:3, 3:1, 6:2}

TEST_make_histogram()
```

4.4 Tree traversal (10 points)

Write a Python function `treefun(t)` that takes as input a rooted tree represented as $[root, subtree_1, subtree_2, ..., subtree_n]$ and returns a tuple `(cnodes, leaves)` where `cnodes` is the number of nodes in the tree and `leaves` is a list of all the leaves of the tree, in left to right order.

Examples - Below is an expanded version of the list given as an example to better visualize a tre ecan be represented sequentially in a list (pre-order).

```
[1,
    [2
```

```
        [3],
        [4],
    ],
    [5],
    [6,
         [7,
              [8]
         ]
         [9]
    ]
]
```

- The output for `treefun([1, [2, [3], [4]], [5], [6, [7, [8]], [9]]])` should be the tuple `(9, [3, 4, 5, 8, 9])`

[ ]:
```
def treefun(t):
    result = (0, [])

    # YOUR CODE HERE




    return result


# This call should return the tuple (9, [3, 4, 5, 8, 9])
treefun([1, [2, [3], [4]], [5], [6, [7, [8]], [9]]])
```

[ ]:
```
def TEST_iterate_over_tree():
    print("{:=^50}".format('Inputs'))
    X = [1, [2, [3], [4]], [5], [6, [7, [8]], [9]]]
    print(f"X: {X}")

    print("{:=^50}".format('Outputs'))
    count, leafs = treefun(X)
    print(f"Count: {count}")
    print(f"leafs: {leafs}")

    assert count == 9
    assert leafs == [3, 4, 5, 8, 9]

TEST_iterate_over_tree()
```

4.5 One-hot encodings (10 points)

Define a function `encoding(data)` that takes as input a list of tuples called `data`, where each tuple contains one or more nonnegative integers. The functions should output a list of one-hot encodings of the tuples in data, one encoding per tuple, by proceeding in two steps: 1) calculate the maximum

integer across all the tuples in data, call this M. 2) for each tuple create the corresponding encoding as a list of M + 1 numbers that are all zeros with the exception of the positions contained in the tuple, where it should contain ones.

For example, if `data = [(3, 1, 5), (2, 0), (1, 2)]`, then `encoding(data)` should return `[[0, 1, 0, 1, 0, 1], [1, 0, 1, 0, 0, 0], [0, 1, 1, 0, 0, 0]]`.

Then write a function `most_frequent(data)` that returns the integer that appears the most often in the tuples in `data`. If there are two or more integers that appear most often, return the smallest one. For the example above, it should return `1`.

[ ]:
```
def encoding(data):
    # YOUR CODE HERE



    return result
# The call below should return [[0, 1, 0, 1, 0, 1], [1, 0, 1, 0, 0, 0], [0, 1,␣
 ↪1, 0, 0, 0]]
encoding([(3, 1, 5), (2, 0), (1, 2)])
```

[ ]:
```
def test_encoding():
    # Test case 1
    data1 = [(3, 1, 5), (2, 0), (1, 2)]
    expected1 = [[0, 1, 0, 1, 0, 1], [1, 0, 1, 0, 0, 0], [0, 1, 1, 0, 0, 0]]
    result1 = encoding(data1)
    assert result1 == expected1, f"Test case 1 failed: Expected {expected1},␣
 ↪but got {result1}"

    # Test case 2
    data2 = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
    expected2 = [[0, 1, 1, 1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 1, 1, 0, 0,␣
 ↪0], [0, 0, 0, 0, 0, 0, 0, 1, 1, 1]]
    result2 = encoding(data2)
    assert result2 == expected2, f"Test case 2 failed: Expected {expected2},␣
 ↪but got {result2}"

    # Test case 3
    data3 = [(0, 0, 0), (0, 0, 0), (0, 0, 0)]
    expected3 = [[1], [1], [1]]
    result3 = encoding(data3)
    assert result3 == expected3, f"Test case 3 failed: Expected {expected3},␣
 ↪but got {result3}"

    print("All test cases passed!")

test_encoding()
```

[ ]:
```
def most_frequent(data):
    # YOUR CODE HERE
    return result

most_frequent([(3, 1, 5), (2, 0), (1, 2)])
```

[ ]:
```
def test_encoding():
    data1 = [(3, 1, 5), (2, 0), (1, 2)]
    expected1 = 1
    result1 = most_frequent(data1)
    assert result1 == expected1, f"Test case 1 failed: Expected {expected1},␣
 ↪but got {result1}"

    data2 = [(1, 2, 3), (4, 5, 6), (7, 8, 9)]
    expected2 = 1
    result2 = most_frequent(data2)
    assert result2 == expected2, f"Test case 2 failed: Expected {expected2},␣
 ↪but got {result2}"

    data3 = [(1, 1), (2, 2, 2), (3, 8, 3)]
    expected3 = 2
    result3 = most_frequent(data3)
    assert result3 == expected3, f"Test case 3 failed: Expected {expected3},␣
 ↪but got {result3}"

    print("All test cases passed!")

test_encoding()
```

4.6 Working with text files (10 points)

Write a function text_stats(fname) that read a text file **line by line** and returns a tuple containing the following elements:

1. The total number of *lines* in the file.
2. The number of *words* in the file. A *word* is defined as a maximally contiguous sequence of one ore more letters. For example, the string 'The SARS-Covid-19 pandemic started in 2020' contains the words 'The', 'SARS', 'Covid', 'pandemic', 'started', and 'in', hence 6 words.
3. A histogram of the word lengths in the file, i.e. a dictionary that maps integers n to the number words of length n that are in the file. The keys n are between 1 and the maximum word length in the file.

My code has 13 lines.

[ ]:
```
def text_stats(fname):
    # YOUR CODE HERE
```

```
    return 0, 0, {}



# The call below calls should return
#(102,
# 833,
# {4: 230,
# 2: 145,
# 3: 162,
# 5: 91,
# 6: 47,
# 7: 49,
# 1: 44,
# 9: 8,
# 10: 11,
# 8: 33,
# 12: 5,
# 13: 2,
# 11: 6})
text_stats('../data/thinking-meat.txt')
```

[ ]:
```
def test_text_stats():
    # Test case 1
    expected1 = expected1 = (299,12929, {5: 1852,3: 1413,4: 1941,11: 306,10:␣
 ↪152,2: 1464,6: 1667,9: 1237,7: 1488,8: 1102,12: 165,1: 142})
    result1 = text_stats('../data/gibberish2.txt')
    assert result1 == expected1, f"Test case 1 failed: Expected {expected1},␣
 ↪but got {result1}"

    print("All test cases passed!")

test_text_stats()
```

4.7 Flatten lists (10p)

Write a function flatten(l) that takes as input a *deep* list that may contain other lists that may contain other lists … and returns a *shallow* list that contains just the atmoic elements of the original list.

[ ]:
```
def flatten(l):
    result = []

    # YOUR CODE HERE
```

```
    return result

l = [1, [2, 3], [4, [5, 6, [7, [8, 9]]], 10], 11, 12]

# This call should print [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
print(flatten(l))
```

4.8 Bonus points: Brute force sorting (10p)

Define a function force_sort(l) that takes as input a list of integers l and outputs a list of the same integers in sorted order. The function should use a brute-force sorting algorithm, meaning that it generates permutations of the initial list until it finds a permutation that is sorted. You should write code to generate permutations yourself (do not use library functions for that). Write a separate function check_sorted(l) that returns True if and only if the input list is sorted.

[ ]:
```
def check_sorted(l):
    # YOUR CODE HERE




    return True


def force_sort(l):
    # YOUR CODE HERE




    return []

# This call should return [-1, 3, 5, 9, 13].
print(force_sort([9, 13, -1, 5, 3]))

# This call should return False.
print(check_sorted([-1, 3, 13, 5, 9]))
```