PA06: Class Finder

University of Virginia, Department of Computer Science
CS 1112 - Intro to Programming
Fall 2024

Due by: 11:00pm on Wednesday, November 6, 2024

Contents

Objective: The objective of this assignment for students is to learn how to work with **dictionaries** and **nested data structures** for a practical application. By the end of the assignment, students should be able to:

- Navigate and iterate through a dictionary
- Understand the differences between accessing entries in a dictionary and entries in a list

1 Background: Dictionaries

Dictionaries is a fundamental data structure used to store and manage a collection of key-value pairs. Unlike simple lists or arrays, dictionaries allow data to be organized and accessed using a unique key as opposed to an index. This key-value relationship provides a flexible and efficient way to manage data. For example, a dictionary of food by letters could look like the following:

```
foods_by_letter_dictionary = {
    'A': ['apple', 'almond', 'artichoke', 'asparagus'],
    'B': ['basil', 'beef', 'bread', 'broccoli'],
    'C': ['cashew', 'cherry', 'chickpea', 'chocolate']
}
```

In this dictionary, keys 'A', 'B', 'C' are associated with lists of foods. To access the elements you use the keys:

```
print(foods_by_letter_dictionary['A'])
```

will print `['apple', 'almond', 'artichoke', 'asparagus']`.

```
print(foods_by_letter_dictionary['A'][0])
```

will print `'apple'`, because this is the element at index 0 of the value of key 'A' (if the value of the key is a list).

Dictionaries, therefore, offer a more intuitive and expressive way to represent and retrieve data, particularly when the data's association with specific keys is clear.

2 Task

Course enrollment is right around the corner! There are so many available courses and sections to choose from, so how do you effectively build your schedule? **You**, as a up and rising developer, will write some functions that will retrieve relevant information from a database of classes and pick your classes faster than anyone else.

In a file called `class_finder.py`, write a small library of functions to interact with this course database. The course database is a dictionary of string keys and list values as pairs. Each key is a string corresponding to a course department abbreviation and its course section. Each value will be a list of information about the course represented in either string or integer format. A short example of the database we will test your code against might look like the following (*this dictionary has been shortened to the first 3 key-value pairs*):

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85],
    "CS 2100-002": ["Data Structures and Algorithms 1", "Lecture", \
        "Derrick Stone", "Mo/We/Fr", 1100, 1150, "Olsson 120", 4, 145, 145]
}  # assume more entries exist...
```

The format of a key-value pair in the "database" is, in order,

```
    "class key": ["class name", "course type (i.e. 'lecture'/'lab'/'discussion')", \
        "instructor", "meeting days", start time (int), end time (int), \
            "meeting room", credit value (int), \
                number of students currently enrolled (int), course capacity (int)]
```

To borrow from the full example above, an example value for the key "CS 1110-100" would be:

```
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85]
```

You must match our file name, class_finder.py, ***exactly***.
If you do not match our filename exactly, our autograder will be unlikely to find it.
You ***must not delete*** any part of courses_dict dictionary from your file!

For all functions, you **do not** need to worry about getting the full database dictionary- you can use our small example in this document to test your code, but we will pass in our own database in the autograder. The format of each value of a key-value pair in the dictionary, which is described above, will be consistent. Be sure to also keep in mind the ways of accessing a value in the dictionary. (i.e. by using a key and/or indexing if the value of a key is a list)

You must also name all of the **functions** exactly as we have presented them in this document. Don't forget **in-line comments** and well as **multi-line docstring comments**, too!

2.1 class_has_lab(class_key_without_section_tag)

Given a string `class_key` without the section tag (i.e. "CS 1110" without the "-100"), find out whether or not the given class name has an associated lab section for that particular class name. if the `class_name` has an associated lab section, return True. Otherwise, return False.
Example usage: given:

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85]
}
```

Calling `class_has_lab("CS 1110")` should return **True**.

However, calling `class_has_lab("CS 1010")` should return **False**.

Be aware that many classes have both a lecture and a lab. Be sure that just because there is an instance of the class being a lecture does not mean that there is no lab associated with the class.

2.2 available_seats(class_key)

Given a string `class_key` that represents a key in the dictionary corresponding to a distinct class section, return the number of available seats. This function should return a integer. If an invalid key is passed, return 0. (there cannot be seats available for a class that does not exist).
Example usage: given

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85]
}
```

Calling `available_seats("CS 1110-100")` should return **2**. *(85-83=2)*

Calling `available_seats("CS 9730-001")` should return **0**. *(Invalid key)*

2.3 lookup_by_name(class_name)

Given a string `class_name`, find out the associated class name's class key. If the class name has multiple class keys in the database, fetch the first appearance of the key. This can occur if the class has both a lecture section, multiple lecture sections, or a combination of lecture and lab sections. If an improper or non-existent class name is passed in, the function should return an empty string.

Practically, you can think of how you might talk with others and say something like "I plan on taking DSA1 (*Data Structures and Algorithms 1*) next semester. How is it?" SIS does not directly recognize the name *Data Structures and Algorithms 1*, so you have to first link up the name with the *subject* and *catalog number* for the class (e.g., "CS" + "2100" to produce "CS 2100"). We're asking you to do that "linking" here in this function.
Example usage: given

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85]
}
```

Calling `lookup_by_name("Introduction to Information Technology")` should return **"CS 1010-001"**.

Calling `lookup_by_name("Advanced Embedded Computing Systems")` should return **""**.

2.4 professor_teaches(professor, class_name="any")

Given a string `professor` and a `class_name` (defaulted to any), check to see if the professor teaches `class_name`. If the `class_name` is left empty, check to see whether or not the professor is teaching **any** classes next semester by looking through the whole dictionary. This function will return a Boolean value True or False. For example, the structure of this function would be:

```
if class_name == "any":  # if class_name is left empty
    # Code here to see if the professor is teaching ANY classes next semester
else:  # if both the professor and class_name were provided
    # Code here to see if the professor teaches class_name
```

Background: Default parameters assigns a default value to the parameter in a given function. If the function is called without providing an argument for the parameters that have default values, the default values will be used. For this specific function, if no argument was passed in for `class_name`, the default value for `class_name` would be "any".
Example usage: given

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85]
}
```

Calling `professor_teaches("David Evans")` should return **"True"**.

Calling `professor_teaches("David Evans", "Introduction to Information Technology")` should return **"True"**.

Calling `professor_teaches("David Evans", "Introduction to Programming")` should return **"False"**. *(Prof. Evans doesn't teach this course, so False.)*

You may assume that our test cases will have valid professor names.

2.5 class_conflict(first_class_key, second_class_key)

Given two strings which both represent a `class_key`, check to see if the two given classes have conflicting start and end times. This function will return a Boolean value True or False. (Dont forget: first be sure to check if "first_class_key" and "second_class_key" are legitimate keys in the dictionary! If they are not, immediately return False.) For this assignment, classes conflict if and only if:

- The classes have at least one day in common; **and**
- The classes are arranged so that the start time of one is *at the same time* or *after* the start time of the other; **and**
- The classes are arranged so that the end time of one is *before* or *at the same time as* the end time of the other.
  - We have arranged the class times into integers representing 24-hour time (1300 for 1:00pm, 1400 for 2:00pm, 1500 for 3:00pm, ...) for easy comparison using mathematical logic. You may assume that all times will be valid and will follow this format.

Example usage: given

```
courses_dict = {
    "CS 1010-001": ["Introduction to Information Technology", "Lecture", \
        "David Evans", "Tu/Th", 1100, 1215, "Olsson 009", 3, 45, 80],
    "CS 1110-100": ["Introduction to Programming", "Lab", \
        "Arohi Khargonkar", "Th", 930, 1045, "Olsson 018", 3, 83, 85],
    "CS 3100-001": ["Data Structures and Algorithms 2", "Lecture", \
        "Robbie Hott", "Tu/Th", 930, 1045, "Rice 130", 3, 150, 150]
}
```

Calling `class_conflict("CS 1010-001", "CS 3100-001")` should return **"False"**.

Calling `class_conflict("CS 1110-100", "CS 3100-001")` should return **"True"**.

Be aware that a class does not only have conflict with another class if the start and end times are the same, one of the values can be **different**. If two classes started at different times, but one of them started when the other class has yet to conclude, that will also cause a class conflict.

You may assume that the classes we pass in for testing meet at the same times (e.g., that a class which meets Monday, Wednesday, and Friday (`"M/W/F"`) will meet at the same times, like 9:00am-9:50am, on all three days). When identifying days, be sure to split the days on the delimiter `'/'` so that you can compare days for different classes as individual strings. You may also assume that we will only pass in valid days for that part of the list.

2.6 build_schedule(class_key_list)

You ***do NOT*** need to edit this function at all! (Unless you want a challenge.)

Given a list of `class_keys`, this function attempts to create a class schedule without any conflicts. You **do not** need to add to this function directly; however, you may edit and add to this function *as you wish* and try to implement new features and challenge yourself. However, that is not necessary. **NO CODE is required** or expected to gain full credit!

2.7 Reflection

In a separate PDF document (preferably using about 150-250 words total), answer the following question:

1. For the `build_schedule` function, briefly describe how you would make changes to the existing code. What additional new features would you add? Why? What additional helper functions would you write in order to add new features and information to the schedule? **NO CODE IS REQUIRED OR EXPECTED**. Give us a general outline/explanation of your thought process.

Keep the following in mind as you write your code, as you will be graded on these aspects:

- Don't forget your header at the top of your `.py` file (see section 3.1)
- Cite your resources clearly or state you did not use any
- Use appropriate variables names throughout your program
- Include in-line comments throughout your code to explain what you are doing (see example of in-line comment below:)

```
food = "apple"  # This is a string representing an apple
```

- Document all of your functions with **multi-line docstrings**. Note that the following is what we are referring to:

```
def func(first_param, second_param):
    '''
    First line - Description of the function - what does it do?
    :param first_param: what is it's data type,
        what does this parameter represent in the function?
    :param second_param: what is it's data type,
        what does this parameter represent in the function?
    :return: what is it's data type,
        what does the return value represent in the function?
    '''
    # BODY OF CODE HERE
```

(Remember, multi-line docstring comments are **not** a replacement for regular in-line comments. There is a place and need for both! Note that the docstring goes immediately after the function header, and before the code in the function).

Still having some issues submitting? Please keep the following in mind:

- Ensure your **file name** matches the file name we ask you to use in the assignment
- Ensure all of your **function definitions** match the ones we ask you to use (same function *name*, and function *parameters*)
- Put the **courses_dict** dictionary *(all of it!)* at the top of your file - you must submit your file with that included! You *will have problems with Gradescope* if you forget to include the dictionary at the top of your file!
- Keep the **build_schedule** function and code from the starter code in your file even if you didn't edit it (remember, you are not required to edit it to gain full credit on this assignment!)

3 Submission and Collaboration Policy

3.1 Your Submission–Comment Header

Your `.py` file should contain the following information (header) at the **top** of your file:

```
# NAME: e.g. I. Lv. Sneks
# COMPUTING ID: e.g. ils3py@virginia.edu
# PA NUMBER and NAME: e.g. PA## - Name of the Assignment
# Resources used (if applicable):
```

For **Resources**, include URLs to any online resources where you studied or reviewed code that is specific to these problems, including any physical textbooks you referenced. Include any other resources you used here. *Note on resources:* Please feel free to refer to the lecture slides, demos, and in-class labs to complete this assignment (all located on Canvas).

Absolutely no collaboration with any fellow students (who are not current CS 1112 course TAs) is allowed. Your work must represent individual effort. You must write your own code: not *just* type it, but also compose it yourself as your own original work.

You must cite any and every source you consult, other than those explicitly provided by the course itself. If you work with, obtain or receive help from another source (Internet website, textbook, TA, tutor, online video, etc.), nothing should be copied or retyped into the submitted solution. References must be documented in a comment in the code on the assignment. Any copied work is an Honor Code violation.

3.2 Gradescope

3.2.1 Submitting on Gradescope

1. Go to Gradescope (linked in Canvas course website)
2. Click on **PA06 – Class Finder** in the assignment list on Gradescope
3. Upload your Python program as `class_finder.py`. *(Check your file name!)*
4. Write your reflection in a separate document and save/download it as a **PDF** (please ask us if you are unsure how to save your file in PDF format. We are happy to help!) Note: this file **must be a PDF document** and not a .docx or .txt file or anything else. *You may lose points on this portion of the assignment if you do not submit it, or if you do not save this file as a PDF document. (If you don't know how to do this, no problem - ask us!)*
5. Submit to the same Gradescope assignment PA06 – Class Finder
6. Ensure you have uploaded **two (2) files before** you submit!

Helpful guidance: Do not wait until the last minute to start this assignment! Remember you can **submit multiple times** to Gradescope. Look at the feedback you receive and then go back and fix your code, then you can resubmit. There is NO penalty for submitting multiple times to Gradescope. If you face any difficulties or have questions, post on Piazza. Also, don't hesitate to reach out to your professor or teaching assistants (TAs) for guidance. We are here to help!