Table of Contents

Lab 7: Dictionaries and CSV Files, F-Strings

Learning Outcomes

In this lab, you will learn:

- how to work with dictionaries,
- how to align strings using f-strings,
- how to read and write csv files.

Introduction

In this lab we will be introducing some new datatypes, notably the *dictionary*. Different datatypes are suited to different use-cases, and have different performance characteristics as well (although for the scripts we are creating, performance is not really that much of an issue!). In addition, we will introduce a method of working with *Comma-Separated Value* files, otherwise known as *csv* files. CSV files store data in rows and columns similar to spreadsheets, but do not contain formulae. They can easily be imported into spreadsheet programs such as Microsoft Excel. Oftentimes we would like to use scripting to *parse* data and then export it to a CSV file.

Why not simply create `xls` or `xlsx` spreadsheets? These are proprietary formats, and working with them can cause formatting or compatibility issues. While there exist modules that will work with `xls(x)` files, we can bypass a lot of frustration by working with *open* formats whenever possible.

7.1 Dictionaries

TASK: Understanding the Limitations of Lists

You have already been working with lists, which look like this:

```
student1 = ['eric', 'smith', '217 Au Large Blvd', 'Toronto', 'Ontario', 'M4A 1P3']
```

A list is a collection of elements, grouped together under one variable name. In this example, you can see that `student1` appears to contain a name and mailing address.

Lists elements are accessible by identifying the name of the list and the *index number* of the element. Let's explore this by creating a function that will create a shipping label string. Note that here we are using f-strings to structure our output.

```
def shipping_label(a_lst):
    "takes a list, generates a string"
    a_str = f"{a_lst[0]} {a_lst[1]}\n"
    a_str += f"{a_lst[2]}\n"
    a_str += f"{a_lst[3]}, {a_lst[4]}\n"
```

```
    a_str += f"{a_lst[5]}"
    return a_str
```

Create a new Python script called `lab7a.py` and place both `student1` and the `shipping_label` function inside it. Make sure to *call* the function using `student1` as an argument, and use `print` with the return value. Your output should look like this:

```
eric smith
217 Au Large Blvd
Toronto, Ontario
M4A 1P3
```

This method has a lot of advantages: we are grouping data that needs to stay together. We don't want `student1`'s address to get confused with `student2`'s, for example. But you might notice that the `shipping_label` function is hard to read. Why does the student's first name come first in the list? Why is the Postal code contained in `a_lst[5]`? Let's expand on this example and add some contact information:

```
student1 = ['eric', 'smith', '647-555-01234', '217 Au Large Blvd', 'Toronto',
'Ontario', 'M4A 1P3', 'esmith1@myseneca.ca']
```

Any programmer working on this program now has to remember that the telephone number is the *third* element, and the email address is the *eighth*. Why the third, and why the eighth? There's no reason, in this case the order of elements in the list is completely arbitrary! Lists are excellent for storing a collection of data with a specific order, but in this case we would much rather identify each element by what they *contain*, rather than by the order in which appear.

Using Dictionaries To Solve The Problem With Lists

Dictionaries are like lists, in that they are a group of elements. However, the individual elements are accessible using *keys* rather than index numbers.

```
>>>> mydict = {'key': 'value'}  # the curly braces indicate a dictionary when
there are two elements with a : between them.
>>>> mydict['key']  # 'key' is used instead of 0.
value
```

- Dictionaries are defined by using curly braces (`{}`). The key *must* come before a colon (`:`), and it *must* have a value next to it.
- Keys are used as identifiers for the elements (or values) of a dictionary, just as index numbers are used with lists. Same as lists, the key will be enclosed in square brackets after the variable name.
- Just like lists, values inside a dictionary can be strings, integers, floats, lists, and so on. However, there are a few restrictions for the datatype of the *key*. Keys must be *immutable*.
- Immutable means they can't change. Without going too deep into the topic, for now all you need to know is that keys can be strings, ints, or floats, but *can't* be lists.

TASK: Modify lab7a

Modify `lab7a.py` so that `student1` is replaced with a dictionary:

```
student1 = {'first_name': 'eric', 'last_name':'smith', 'addr1': '217 Au Large
Blvd', 'city': 'Toronto', 'prov': 'Ontario', 'pcode': 'M4A 1P3'}
```

And now modify the function to use this dictionary instead of a list. Make sure that the format stays the same.

```
>>>> from lab7a import student1, shipping_label
>>>> print(shipping_label(student1))
Eric Smith
217 Au Large Blvd
Toronto, Ontario
M4A 1P3
```

Make sure to research a way to get the name capitalized.

7.2 Modifying Dictionaries

Dictionaries contain *mutable* values and *immutable* keys. As stated before, *immutable* objects cannot be modified, only created or removed. *Mutable* objects are modifiable. That means we cannot change a key once it has been created, but we can change the value assigned to it.

Use the Python interpreter to follow these examples.

```
meal_plan = {'breakfast': 'oatmeal', 'lunch': 'sandwiches', 'dinner':
'broccoli'}

meal_plan['lunch'] = 'ice cream'
print(meal_plan)
```

New key/value pairs can be added using the same syntax:

```
meal_plan['snack'] = 'cookies'
print(meal_plan)
```

Removing key/value pairs from a dictionary is more complicated. We can use a method called `.pop()`, which will take a key as argument, return the value associated with the key, and remove that pair from the dictionary. Observe what happens when using this method with `meal_plan`.

```
value_removed = meal_plan.pop('dinner')
print(meal_plan)
print(value_removed)
```

You will a `.pop()` method inside list objects as well. "Popping" is a concept in programming where we select an item for processing, handle it, and remove it from the original collection.

7.3 TASK: Print A Menu

- Call this script `lab7b.py`.
- Copy the `meal_plan` dictionary into this new file. Use either the original, or the modified version, it's up to you.
- Create a function called `print_meal_plan` that will print a menu that includes three meals. (omit the snack).
- `print_meal_plan` should accept a dictionary as an argument, but this time, your function shouldn't return anything. It should instead have `print` statements inside.
- You won't submit this version of the file, but save it for now and make sure it works.

Sample Output

```
MENU FOR TODAY
==============
Breakfast: oatmeal
Lunch: sandwiches
Dinner: broccoli
```

7.4 F-String Formatting

You'll notice that the formatting for our menu printing program isn't very nice. Usually when we are creating menus, we would like titles to be centred, and columns to be justified to the left or right. Fortunately, things like this are easy to do with f-strings.

Open a new interpreter window. Let us start by defining a total width for our menu:

```
width = 40
```

It will be easier to visualise this space if we print characters to fill up this space. Enter this code:

```
print("=" * width)
```

What you should see printed is 40 equal signs. It's a nice feature that we can use the * *multiply* a string!

Now we can demonstrate alignment using f-strings.

```
print(f"{'x' : <40}")  # Notice that " is on the outside, and ' is on the
inside.
print(f"{'x' : ^40}")
print(f"{'x' : >40}")
```

What you should see now are three lines with x, with the x justified to the left, to the centre, and to the right.

This code, however, has a weakness: what if we decide that our 'width' should be 50 instead of 40? Right now we have hardcoded that width inside our f-strings. Fortunately we can use variables to specify width, but we need to contain it inside curly braces.

```
print(f"{'x' : <{width}}")  # Notice there is not space between < and
{width}.
print(f"{'x' : ^{width}}")
print(f"{'x' : >{width}}")
```

We can combine things to create columns. I'm going to use hardcoded widths again to make things easier to see.

```
print(f"|{'x' : <20}|{'y' : >20}|")
```

7.5 TASK: Improve The Menu Script

- Modify your `lab7b.py` file. It's not necessary to submit the old version, just this new version.
- Use f-strings to format the output correctly. The total width should be 50. Use a multiplication operator to create the horizontal line between the title and meals. Match the sample output exactly.

Sample Output

```
                  MENU FOR TODAY
==================================================
Breakfast                                  oatmeal
Lunch                                   sandwiches
Dinner                                    broccoli
```

7.6 Common Issues With Dictionaries

Handling KeyErrors

There is one issue that arises when working with a dictionary that has been modified. What if we try to get the value of a key that doesn't exist? Refer back to the interpreter and your `meal_plan` example to simulate this. Someone has removed the 'snack' key, now what happens when we try to access it?

```
>>>> print(meal_plan['snack'])
KeyError: 'snack'
```

We have already learned about using exception handling to deal with issues such as these. However it can be tedious to put every instance of this in a try/except block. As an alternative, you can use the method `.get()`. This has the added feature of allowing us to substitute any missing value with a default of our choosing.

```
>>>> meal_plan.get('dinner')
broccoli
>>>> meal_plan.get('snack')
None
>>>> meal_plan.get('snack', 'fruit')
fruit
```

This second argument is useful, it contains a *default value* of `None`, but we can override that default value with one of our own. Use `help(meal_plan)` and look at some of the other

methods available; you will no doubt see more examples of these arguments with defaults already set.

7.7 Iterating Dictionaries

You can use a for loop to iterate through a dictionary. The only difference being, by default the *looping variable* will contain the key for each key/value pair found. Also you will note that dictionaries don't have an *order*. You have no guarantee about which key will appear first during the iteration.

```
>>>> mydict = {'brand': 'subaru', 'model': 'impreza', 'color': 'green',
'year': '2017'}

>>>> for key in mydict:
>>>>     print(key)

brand
color
model
year
```

There are other ways of iterating through a dictionary, which may be more useful depending on what you're doing. By using the methods `.keys()` and `.values()` you can generate a *list* of either keys or values respectively.

```
>>>> mydict.keys()
['brand', 'model', 'color', 'year']
>>>> mydict.values()
['subaru', 'impreza', 'green', '2017']
```

Finally, by using the method `.items()` you can generate a *tuple* which contains both.

```
for k, v in mydict.items():
    print{f"Key is: {k} and value is: {v}.")
```

What Is A Tuple?

Remember how we said that dictionary keys must be immutable, and therefore we cannot use a list as a key? A *tuple* is yet another datatype in Python. It is very similar to a list in almost every way, except that it is immutable. This means that you cannot `append()` or `remove()` an element from a tuple, you can only create or delete the entire tuple.

Tuples have different syntax as well.

```
tuple_with_one_element = ('hello', )  # notice the comma. This is only needed
for tuples with one thing in them.
tuple_with_more = ('hello', 'world', '!')
```

Why use a tuple? There is a small difference in performance when working with a tuple versus a list: a tuple is much faster in a lot of cases. And you can use a tuple as a dictionary

key, as it is immutable. As you learn about more modules in the Python standard library, you will no doubt encounter functions that return tuples rather than lists, and so it's worth noting their existence here.

7.8 TASK: Modifying A Dictionary

For this task we are going to create a list of meal plans for a Summer Camp. For each day of camp, we will be using an empty template dictionary with keys for breakfast, lunch, and dinner. When we add a day we will modify the values for each dictionary key, and add the dictionary into a list. This will also help us to illustrate another important programming concept.

- Create a new file called `lab7c.py`.
- Inside of this file, we want to create a dictionary called `template`. This `template` should have keys called `'breakfast'`, `'lunch'`, and `'dinner'`, and the value for each should be `None`.
- Create a **main block** in your script. Inside the main block, create a loop that will ask `Would you like to add a day? (y/n)`. If the user enters `y`, create a new dictionary from the template. For example: `new_day = template`.
- Your script should now *iterate through each key in your `new_day` dictionary*, and ask the user to enter a meal idea for each. Use one of the iterating methods we have discussed.
- Append `new_day` to a list of days called `days`.
- Once the user has stopped adding new days, ask them to enter a number. The number should be between 1 and the total number of days in your list. For example: `Please enter a day number for the menu you would like to print (1-3):`.
- If the number is out of range of the list, print an error message and exit.
- If the number is within range, use the function you created in `lab7b.py` to print that day's menu. Use `import` to do this.

Sample Output

```
Would you like to add a day? (y/n) y
Please enter what you would like to eat for breakfast:eggs
Please enter what you would like to eat for lunch:tuna
Please enter what you would like to eat for dinner:pizza
Would you like to add a day? (y/n) y
Please enter what you would like to eat for breakfast:hot dogs
Please enter what you would like to eat for lunch:veal
Please enter what you would like to eat for dinner:pasta
Would you like to add a day? (y/n) n
Please enter a day number for the menu you would like to print (1-2): 1
                MENU FOR TODAY
================================================
Breakfast                                hot dogs
Lunch                                        veal
Dinner                                     pasta
```

Explanation Of The Error

You might have noticed that the output is wrong! We are asking to print the first day's menu (eggs, tuna, and pizza) but it is printing the last day's meals.

- Use the debugger to watch what happens as you enter meals. What is happening?
- The reason that we are seeing the template change as well as each dictionary in the list, is that we aren't actually creating copies of the template. The code `new_day = template` is creating a *reference* to `template`.
- What is a reference? It means that we have created one dictionary, and we are storing it in memory. Python doesn't want to use more memory than it needs to, so by default a reference will *point to* the original dictionary rather than duplicate it into a address. In other words, `new_meal` is just an alias for `template`, and appending it into our days list is just creating another alias. **There is only one dictionary in our entire program!**
- In order for this program to work the way we want it to, we will need to *explicitly* create a copy of `template` each time we run our program. Replace the `new_meal = template` line with the following: `new_meal = template.copy()`.
- Run this code again using the debugger. This time, you shouldn't see the `template` dictionary change, and each item in your `days` list will have unique meals.
- Save `lab7c.py` and submit it as part of your lab.

NOTE: `.copy()` creates what is known as a *shallow copy*. There also exists (in the `copy` module) a function called `deepcopy()`. These concepts become more important when the object we are copying contain more objects inside it, but this is outside the scope of our course. If you would like to learn more, refer to this discussion on StackOverflow.

7.9 CSV Files

CSV files are used to store tables. They don't contain formula information like spreadsheets, but can easily be imported into a spreadsheet program.

A typical CSV file will have columns and rows. Data is store between double-quotes, and a comma (`,`) will define the boundary between columns. A newline character (`\n`) will define the boundary between rows. For example:

```
"First Name","Last Name","Student Number","GPA"
"Eric","Smith","0123","2.0"
"Bart","Simpson","0567","4.0"
"Sung-won","Kim","0489","3.8"
```

Note: You can download sample.csv from our site.

Reading From A CSV File

Python has a module called `csv` which will allow us to work with a CSV file very easily. You can refer to the documentation to learn more about this module. Since we've learned about using dictionaries, it will be easiest to use `DictReader` to import the data as a *list* of *dictionaries*.

To open a csv file, you will need to create a file object, same as you learned in the last lab. It's always good practice to add exception handling, and to close the file afterward.

Once that is done, you can use `DictReader` with the file object as an argument.

- Create `sample.csv` using the example above.
- Create `lab7d.py` and copy the following into it:

```
import csv

list_of_dicts = []
f = open('sample.csv', 'r')
reader = csv.DictReader(f)
for row in reader:
    list_of_dicts.append(row)
f.close()
```

- Open `lab7d.py` in VSCode. Use the debugger to step through the code, and to look at `list_of_dicts` once it was been created.
- You should see a list of dictionaries, with common keys taken from the first line of the CSV file.
- The sample output below has one dictionary on each new line, how would you do this using Python code?

```
{'First Name': 'Eric', 'Last Name': 'Smith', 'Student Number': '0123', 'GPA':
'2.0'}
{'First Name': 'Bart', 'Last Name': 'Simpson', 'Student Number': '0567',
'GPA': '4.0'}
{'First Name': 'Sung-won', 'Last Name': 'Kim', 'Student Number': '0489',
'GPA': '3.8'}
```

Writing To A CSV File

To perform the inverse operation, you might expect the `csv` module to have a `DictWriter` method. And you would be correct.

One difference between reading and writing, is that the header (containing column names such as 'First Name') aren't written by default, so we will need to perform the operation manually.

We create a list called `fieldnames`, and we use the `.keys()` method only on the *first* dictionary to get the fieldnames to use in our CSV file. Then we can use `w.writeheader()` to write this list as our first row in the file. Then a simple for loop can be used to convert each dictionary into a new row.

```
f = open('sample2.csv', 'w')
fieldnames = list_of_dicts[0].keys()
w = csv.DictWriter(f, fieldnames=fieldnames)
w.writeheader()
for row in list_of_dicts:
    w.writerow(row)
f.close()
```

7.10 TASK: Substitute Data In A CSV File

- Modify `lab7d.py` to perform this task.
- Put both the `DictReader` and `DictWriter` code into functions. The filename should be the first argument for each function. The reader function should return a list of dictionaries, and the writer function should take a list of dictionaries as its second argument.
- Your script should call the reader function, and return the list of dictionaries. Then, it should perform a *substitution* on each row of the list of dictionaries:
  - If 'First Name' is Christopher, change it to Chris.
  - If 'Last Name' is Patal, change it to Patel.
  - If 'Last Name' is Smith, change it to Nichols.
  - If 'Address' is 81 Vanier, change it to 72 Princeton.
  - If 'Last Name' is Geary, change *'Address'* to 455 Bloor.
  - If 'City' is North York, change it to Toronto.
  - If 'Country' is Canada, change it to CA. Warning: you will need to handle the case where a csv file may not contain this key!
- Your code should then save overwrite the file contents of the filename given. Your script should use a command line argument from the user to set the filename.
- Test with both of these files: test1.csv and test2.csv.
- Submit your script as part of your lab submission.

CHALLENGE TASK: Printing a Table

- Copy `lab7d.py` and rename the copy `challenge7.py`.
- Add a new function that will print the modified table.
- Use keys for column titles.
- Use f-strings to format the output in such a way that the contents are easy to read.

Sample Output

```
First Name | Last Name | GPA
---------------------------
Bart       | Simpson   | 4.0
...
```

Deliverables

You can download the check script here.

- ☐ lab7a.py
- ☐ lab7b.py
- ☐ lab7c.py
- ☐ lab7d.py
- ☐ challenge7.py
- ☐ lab7-check-output.txt

Glossary

- key
- value
- mutable
- immutable
- tuple
- reference