DICTIONARIES

REMINDER ABOUT AUTOGRADER OUTPUT

- Gradescope has automatic style deductions
- Take a look sometime today to make sure that you have no automatic style errors!
- Other things that are **your responsibility to check for:**
  - submitting all files
  - runtime issues

If you see 0/40, that's not something you should ignore!

DICTIONARIES

Dictionaries (also called "dicts") are the much more commonly used **unordered** collection

- Associates **keys** to **values**
- Allow for looking up some information associated with a search key
- Keys must be unique, values do not need to be unique

WHAT IS A MAPPING?

Any association from **keys** (things you can search by) to **values** (information you might want to know.)

The Penn Directory, for example:

```
Name : Email
Harry Smith : sharry@seas
Travis McGaha : tqmcgaha@seas
...
```

Here, the names are keys and the emails are values.

OTHER DICTIONARY EXAMPLES

Here are `dict` implementations of literal dictionaries, e.g. English definition dictionary or English to French translation dictionary.

```
english = {"cat" : "small feline",
           "car" : "four-wheeled vehicle",
           "dog" : "best friend"}
```

```
english_to_french = {"cat" : "chat (nm)",
                     "car" : "voiture (nf)",
                     "dog" : "chien (nm)"}
```

DICT SYNTAX

Dict literals are defined with curly braces (`{}`) and separate keys and values with a colon.

- `{3, 10, 15}`
  - is a **set** with three elements
- `{"Harry" : "sharry", "Travis" : "tqmcgaha"}`
  - is a **dict** with two elements (key-value pairs)
- `{}` is an empty dict
  - writing just `dict()` gets the same result

DICTIONARY PRACTICE: READING

Given the following dictionaries, which ones are legal dictionaries? (Legal / Illegal) **L1**

```
speak = {
  "dog": "woof",
  "cat": "meow",
  "seal": "arf",
  "fox": "woof"
}
```

```
movies = {"Camp Miasma",
          "Blue Heron", "Digger"}
```

```
recs = {201:["Zwe", "Hannah"],
        203: ["Adi", "Sofia"],
        201: ["Hannah", "Zwe"]}
```

DICTIONARIES AS ENCODINGS

Cryptograms are puzzles where you're presented with what seems to be a garbled mess of letters. Your job is to come up with an assignment of *encoded* letters to *decoded* letters so that the output is real English.

```
XQL DKVX HBX  # encoded
```

```
# mapping encoded -> decoded
{"X" : "T", "Q" : "H", "L" : "E", "K" : "O",
 "V" : "S", "H" : "C", "B": "A"}
```

```
# decoded applying the dictionary
THE LOST CAT
```

TRY IT!

C1 Write a dictionary that can be used to decode the cryptogram into a valid English phrase. Additionally, think concretely about what techniques you're using to solve the puzzle. Can you speak coherently about them?

```
WZG PFW JFW UR WZG YFW
```

Hint: start with `{"W" : "T", "U" : "O"}`

GETTING

Given a dictionary `d`, you can access the value associated with key `k` by writing `d[k]`:

```
last_names["Harry"]                          # ---> "Smith"

area_codes[215]                              # ---> "Philadelphia"

months["October"]                            # ---> 10

food_group["Milk"]                           # ---> "Dairy"
servings_per_day["Dairy"]                    # ---> 3
servings_per_day[food_group["Milk"]]         # ---> 3 (do you see why?)
```

Any of these would throw an error if the key is not already present in the dictionary!

C2 Write a function that accepts a cryptogram as a string and a dictionary that provides the mapping for how to decode it. Return the decoded cryptogram.

```
def solve(cryptogram: str, decoder: dict) -> str:
  ...
```

```
WZG PFW JFW UR WZG YFW
decoder = {
    "W": "T", "Z": "H", "G": "E", "P": "C", "F": "A",
    "J": "S", "U": "O", "R": "N", "Y": "M",
}
```

C2 Write a function that accepts a cryptogram as a string and a dictionary that provides the mapping for how to decode it. Return the decoded cryptogram.

```
def solve(cryptogram: str, decoder: dict) -> str:
  result = ""
  for char in cryptogram:
    if char in decoder:
      result += decoder[char]
    else:
      result += char
  return result
```

```
WZG PFW JFW UR WZG YFW
decoder = {
    "W": "T", "Z": "H", "G": "E", "P": "C", "F": "A",
    "J": "S", "U": "O", "R": "N", "Y": "M",
}
```

SETTING

Given a dictionary `d`, you can set or update the value associated with key `k` by writing `d[k] = <new_value>`:

```
last_names = {"Harry" : "Smtih"}
```

You can add a new key or update the value of an existing one:

```
last_names["Talie"] = "Massachi"
last_names["Harry"] = "Smith"
print(last_names)
```

```
{"Harry" : "Smith", "Talie" : "Massachi"}
```

MORE EXAMPLES

```
last_names["Talie"] = "Massachi"

recitation_attendances["Renate"] = 4

total_score["Harry"] = round_one_score["Harry"] + \
                       round_two_score["Harry"]

# these are equivalent ways of updating
recitation_attendances["Aksel"] += 1
recitation_attendances["Aksel"] = recitation_attendances["Aksel"] + 1
```

SHORT SETTING ACTIVITY

```
ages = {"Stella" : 16, "Meredith" : 54}
```

George is twice as old as Stella. Jeff's age is the difference between George and Meredith. Bunny is one year older than that. *Set everyone's age into the dictionary.*

```
ages["George"] = 2 * ages["Stella"]
ages["Jeff"] = ages["Meredith"] - ages["George"]
ages["Bunny"] = ages["Jeff"] + 1
```

SHORT INCREMENTING ACTIVITY

Model the following narrative by updating the values in the dictionary.

> *Sharon and Ben are enrolled in the course. Sharon and Ben attended Recitation 0. Ben attended Recitation 1. Both attended Recitation 2.*

```
attendance = {"Sharon" : 0, "Ben" : 0}
attendance["Sharon"] += 1
attendance["Ben"] += 1
attendance["Ben"] += 1
attendance["Sharon"] += 1
attendance["Ben"] += 1
```

REMEMBER SOMETHING FROM LAST WEEK?

L1 Remembering what you know about sets, can you--in one line!--write an expression that represents the number of *unique* characters in a string `s`?

```
len(set(s))
```

(If we want to make it case-insensitive, then `len(set(s.lower()))` works.)

PROBLEM SOLVING WITH DICTIONARIES

We want to write a function that turns a list of names into a dictionary where the *keys* are the name and the *values* are number of distinct letters in that name

C1 Write out the a unit test that should pass when using the list below as input.

```
def count_letters(names: list[str]) -> dict[str, int]:
  ...
count_names_by_letter(["SUSAN", "SADIA", "PAUL", "TJ"])
```

```
def test_count_letters():
  expected = {"SUSAN" : 4, "SADIA" : 4, "PAUL" : 4, "TJ" : 2}
  actual = count_names_by_letter(["SUSAN", "SADIA", "PAUL", "TJ"])
  assert actual == expected
```

C1 also We want to write a function that turns a list of names into a dictionary where the *keys* are the first initials and the *values* are counts of names that start with that letter.

```
def count_letters(names: list[str]) -> dict[str, int]:
    counts = dict()
    for name in names:
        unique_letters = len(set(name))
        counts[name] = unique_letters
    return counts
```

BOOKS

TITLE	AUTHOR	PAGES
Three Women	Lisa Taddeo	306
Office Politics	Wilfrid Sheed	249
Simple Passion	Annie Ernaux	80

Quick exercise:

- Write a dictionary that maps these book titles to their authors.
- Write a separate dictionary that maps these titles to their pages.

TITLE	AUTHOR	PAGES
Three Women	Lisa Taddeo	306
Office Politics	Wilfrid Sheed	249
Simple Passion	Annie Ernaux	80

```
authors = {"Three Women" : "Taddeo", "Office Politics": "Sheed",
           "Simple Passion" : "Ernaux"}

pages = {"Three Women" : 306, "Office Politics": 249,
           "Simple Passion" : 80}
```

TUPLES

Tuples are **immutable** sequences:

- They cannot be changed once created
- Otherwise, they behave exactly like lists.

Tuples are useful as collections of associated values, e.g. as multiple pieces of information associated with a single key in a dictionary.

```
books = {"Three Women" : ("Taddeo", 306),
         "Office Politics": ("Sheed", 249),
         "Simple Passion" : ("Ernaux", 80)}
```

IF TIME...

We want to write a function that turns a list of names into a dictionary where the *keys* are the first initials and the *values* are tuples with counts of unique letters and counts of total vowels.

```
def count_letters_2(names: list[str]) -> dict[str, tuple]:
    counts = dict()
    for name in names:
        unique_letters = len(set(name))
        vowels = 0
        for char in name:
          if char in "AaEeIiOoUu":
            vowels += 1
        counts[name] = (unique_letters, vowels)
    return counts
```

ZOOMING OUT: *DATABASES*

A **database** is an organized collection of data.

- Can be stored in a single file (like a `.csv`) or hosted in a computer cluster
- Typically organized with one or more ("primary") **keys** that allow for easy lookup of associated **values**.
- Can be modelled in Python with a dictionary that maps keys to tuples!

USE	RELATIONSHIP
Student database	`Penn ID -> (Name, Matriculation Year, PennKey, ...)`
Book Review database	`Title -> (Author, Rating, # Pages)`
Film Box Office database	`Title -> (Domestic Gross, Foreign Gross)`

WORKED EXAMPLE: BOOKS!

I'm vain, so we're going to use my personal data: my collection of books read from Goodreads. **(follow me?)**

We'll use this data to build a recommender system

- "I heard about this author, can you recommend me her best book?"
- "What's the best book from last year?"

UNDERSTANDING THE DATA

Title	Author	Year	Pages	Rating	Review
East of Eden	John Steinbec	1952	601	4.42	It's nice to do a book club with your friends
Trust	Hernan Diaz	2022	402	3.82	Beef tea Milk 👍
Runaway: Stories	Alice Munro	2004	335	3.97	this was good. that's my thought
Annihilation (Southern	Jeff Vanderme	2014	195	3.78	Wait till they come out with Area Y

What's going on here? What do we have to work with? (🗣️🗣️🗣️)

UNDERSTANDING THE DATA

Title	Author	Year	Pages	Rating	Review
East of Eden	John Steinbec	1952	601	4.42	It's nice to do a book club with your friends
Trust	Hernan Diaz	2022	402	3.82	Beef tea Milk 👍
Runaway: Stories	Alice Munro	2004	335	3.97	this was good. that's my thought
Annihilation (Southern	Jeff Vanderme	2014	195	3.78	Wait till they come out with Area Y

For each data point (Book), we have:

- title
- author
- year, page count, rating, review

How many lines of the file do we need to read to process all the data for a single book? (🗣️🗣️🗣️)

UNDERSTANDING THE DATA

Title	Author	Year	Pages	Rating	Review
East of Eden	John Steinbec	1952	601	4.42	It's nice to do a book club with your friends
Trust	Hernan Diaz	2022	402	3.82	Beef tea Milk 👍
Runaway: Stories	Alice Munro	2004	335	3.97	this was good. that's my thought
Annihilation (Southern	Jeff Vanderme	2014	195	3.78	Wait till they come out with Area Y

What are the types for each of these properties? (🗣️🗣️🗣️)

UNDERSTANDING THE DATA

For each data point (Book), we have:

- title
  - `str`
- author
  - `str`
- year, page count, rating, review
  - `int`, `int`, `float`, `str`

PROCESS BOOKS

```
def process_book_file(filename : str) -> dict[str, tuple]:
    """Returns a dictionary d mapping
    book titles to tuples of book information.
    """

    ...
    return d
```

What is the type of `d`? (🗣️🗣️🗣️)

PROCESS BOOKS

```
def process_book_file(filename : str) -> dict[str, tuple]:
    """Returns a dictionary d mapping
    book titles to tuples of book information.
    """
    ...
    return d
```

What are the types of `d`'s keys? What are the types of its values? **Be as detailed as possible.** (🗣️🗣️🗣️)

PROCESS BOOKS

```
def process_book_file(filename : str) -> dict[str, tuple]:
    """Returns a dictionary d mapping
    book titles to tuples of book information.
    """

    ...
    return d
```

Helpful: `file.readlines()` returns a list of all lines in the file that can easily be processed.

- given a list of lines, how do we determine how many books there are?
- if line number `i` has the title of book, what line number is the author on?

TOGETHER: PRINT ALL BOOKS BY AUTHOR

```
def find_by_author(d: dict[str, tuple], author: str) -> set[str]:
    """Return a dictionary containing
    just those titles by the given author.
    """

    ...
```

YOU: FIND BEST BOOK BY AUTHOR

C1

```
def best_by_author(d: dict[str, tuple], author: str) -> str:
    """Return the name of the book that
    has the highest rating by that author.
    """

    ...
```

DICTIONARIES FOR AGGREGATION

REMEMBER: AGGREGATION

Aggregation is a pattern that we use to combine multiple pieces of information from a sequence or dataset.

- Find the biggest value in a list
- Calculate the average exam score
- Determine the best book by a given author from a dataset of books

Roughly, in general:

```
accumulator = <initial value>
for element in elements:
  accumulator = update_aggregator(element, accumulator)
```

QUICK BENCHMARK

By the next exam, you should be able to write this in about a minute without thinking too hard. It should become "muscle memory".

Finish the function so that it counts the occurrences of a particular `target` in a list of `names`.

```
def count_name_occurrences(target: str, names: list[str]) -> int:
  ...
```

QUICK BENCHMARK

By the next exam, you should be able to write this in about a minute without thinking too hard. It should become "muscle memory".

Finish the function so that it counts the occurrences of a particular `target` in a list of `names`.

```
def count_name_occurrences(target: str, names: list[str]) -> int:
  count = 0
  for name in names:
    if name == target:
      count += 1
  return count
```

DICTIONARIES TAKE IT UP ANOTHER LEVEL

Finish the function so that it counts the occurrences of ALL NAMES in a list of `names`.

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
  ...
```

First, a unit test for the below:

```
count_all_name_occurrences(["Harry", "Gary", "Larry", "Gary"])
```

The unit test...

```
def test_count_all_names():
  names = ["Harry", "Gary", "Larry", "Gary"]
  expected = {"Harry" : 1, "Gary" : 2, "Larry" : 1}
  actual = count_all_name_occurrences(names)
  assert actual == expected
```

DICTIONARIES TAKE IT UP ANOTHER LEVEL

C2 Finish the function so that it counts the occurrences of ALL NAMES in a list of `names`.

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
    ...
```

DICTIONARIES TAKE IT UP ANOTHER LEVEL

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
  counts = {}
  for name in names:
    if name not in counts:
      counts[name] = 1
    else:
      counts[name] += 1
  return counts
```

Start with an empty dictionary.

DICTIONARIES TAKE IT UP ANOTHER LEVEL

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
  counts = {}
  for name in names:
    if name not in counts:
      counts[name] = 1
    else:
      counts[name] += 1
  return counts
```

We still look at each name one by one...

DICTIONARIES TAKE IT UP ANOTHER LEVEL

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
  counts = {}
  for name in names:
    if name not in counts:
      counts[name] = 1
    else:
      counts[name] += 1
  return counts
```

The first time we see a name in the list, we need to create an entry for it in the `dict`. The name is the key and the value is the count of the # of times seen, which is `1` now—**why?**

DICTIONARIES TAKE IT UP ANOTHER LEVEL

```
def count_all_name_occurrences(names: list[str]) -> dict[str, int]:
  counts = {}
  for name in names:
    if name not in counts:
      counts[name] = 1
    else:
      counts[name] += 1
  return counts
```

If we've seen a name before, an entry for it already exists. So now we've seen it once more than we had before pulling it out of the list and therefore we increment its count.

GENERAL AGGREGATION FOR DICTIONARIES

```
for thing in things:
    if thing not in agg_dict:
      agg_dict[thing] = <initial value>
    else:
      agg_dict[thing] = update_aggregator(thing, agg_dict[thing])
```

CHALLENGE

Write a function to create a dictionary grouping each name in the list by first letter, e.g.:

```
def test_group_names():
  names = ["Portugal. The Man", "Pantera", "Cameron Winter", "Metric"]
  expected = {"P": ["Portugal. The Man", "Pantera"],
              "C": ["Cameron Winter"],
              "M": ["Metric"]}
  actual = group_names(names)
  assert actual == expected
```

C3 Write `group_names()`. Start with the signature, add type annotations. What are the keys in the resulting dict, and what are the values?

```python
def group_names(names: list[str]) -> dict[str, str]:
  groups = {}
  for name in names:
    first_letter = name[0]
    if first_letter not in groups:
      groups[first_letter] = [name]
    else:
      groups[first_letter].append(name)
  return groups
```