OBJECTS (USING AND STARTING TO MAKE THEM)!

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.

S7 Write out the dictionary that should be returned by this function when called on the list below.

```
count_names_by_letter(["Susan", "Sally", "Paul", "Nico"])
```

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.

What's wrong with this implementation? (**L11**; describe, don't fix)

```
def count_names_by_letter(names: list[str]) -> dict[str, int]:
    initial_counts = dict()
    for name in names:
        first_letter = name[0]
        initial_counts[first_letter] += initial_counts[first_letter]
    return initial_counts
```

(Then, we'll fix it together.)

Previously, `initial_counts[first_letter] += initial_counts[first_letter]` leads to a `KeyError` if `first_letter` isn't already present.

```
def count_names_by_letter(names: list[str]) -> dict[str, int]:
    initial_counts = dict()
    for name in names:
        first_letter = name[0]
        if first_letter not in initial_counts:
          initial_counts[first_letter] = 1
        else:
          initial_counts[first_letter] += initial_counts[first_letter]
    return initial_counts
```

Two cases now: set a *default value* if the key isn't already present; update it if it is.

REVIEW: WHAT IS AN OBJECT/CLASS?

A class in Python is a construct that allows us to "bundle data and functionality together." *

- A class defines a new data type!
- Allows instances of that class to be created.

* *From the Python documentation on classes*

A class consists of:

- Some attributes (also called fields) that store data
- Some functions that operate with these fields
  These allow us to create abstractions that are easier to wrap your head around.

REVIEW: CLASS AS A TOOL FOR ABSTRACTION

Some features of a class could also be achieved from a tuple, but consider...
Which of these better communicates its purpose?

```
c = (0.5, 0.5, 0.25)
```

REVIEW: CLASS AS A TOOL FOR ABSTRACTION

Some features of a class could also be achieved from a tuple, but consider...
Which of these better communicates its purpose?

```
c = (0.5, 0.5, 0.25)

c = Circle(x_center = 0.5, y_center = 0.5, radius = 0.25)
```

REVIEW: ATTRIBUTES

To build a class, we need to decide which attribute we will include in our abstraction.
Let's say we wanted to make an object that represented a song in a streaming playlist, what attributes might we want to store in that class? What types would they be? **(L13)**

REVIEW: SYNTAX

If we have an object that we want to access the fields of, we can do so using the `.` operator

```
bleak_film = Movie("Sirât", 2025, 122, "Drama", "Digital", 3.5)
the_name = bleak_film.name
print(bleak_film.name)
if (bleak_film.length > 120):
    print("TOO LONG")
bleak_film.genre = "extremely depressing " + bleak_film.genre
```

(NOTE: we do not use `()` when accessing attributes directly. `()` is usually used to indicate some sort of function call)

PRACTICE:

Wich of these are (A) *method* calls, (B) accessing *attributes*, or (C) neither

- (M1) `name.upper()`
- (M2) `my_movie.name`
- (M3) `my_move.price_adjust_inflation(2020)`
- (M4) `penndraw.set_pen_color(penndraw.BLACK)`
- (M5) `len(name)`
- (M6) `number.numerator`

OBJECTS & ABSTRACTION

Classes provide the implementations for how objects behave.

- This lets the author of the class worry about *how to make objects work* and lets the user of the object benefit from their careful planning!
- Using objects that others have implemented is what makes programming fast & fun.

ONLINE MESSAGEBOARD

I have written two classes for you: `Board` and `Message`. We could worry about how they work... or we could not.

```
class Board:
    def __init__(self, url, username):
        self.url = url.rstrip("/")
        self.username = username

    def post(self, text):
        """Post a new message to the board. Returns the new Message."""
        response = requests.post(
            f"{self.url}/messages",
            json={"username": self.username, "text": text},
        )
        ...
```

THE APPLICATION PROGRAMMING INTERFACE (API)

An **API** refers to the ways that software (e.g. an object) is intended to be used.

- it hides the implementation details because they don't matter
- it exposes the "verbs" (methods, here) that the software allows you to use

MESSAGE

A `Message` object represents a single message on an online messageboard. It has no important methods. We won't even create one directly ourselves. It stores the following attributes:

- `id`, a unique identifier for this post
- `author`, the username of the person who wrote the post
- `text`, the body of the post
- `timestamp`, the time at which the message was received and posted

Given a `Message` object called `msg`, you could access any of these attributes (e.g. `msg.id`)

BOARD

A `Board` object represents a connection to a messageboard. Its attributes are not important. Its methods include the following:

METHOD	DESCRIPTION
`b = Board(url, username)`	Connect to a messageboard living at the given URL, representing yourself with the chosen `username`.
`b.post(text)`	Post a new message to the messageboard. This message will be publicly visible.
`b.get_recent(n)`	Find a list of the `n` most recently posted messages.
`b.search(keyword)`	Find a list of messages that contain the given `keyword`.

METHOD	DESCRIPTION
`b = Board(url, username)`	Connect to a messageboard living at the given URL, representing yourself with the chosen `username`.
`b.post(text)`	Post a new message to the messageboard. This message will be publicly visible.
`b.get_recent(n)`	Find a list of the `n` most recently posted messages.
`b.search(keyword)`	Find a list of messages that contain the given `keyword`.

C12 Create a new `Board` object `b` that connects to `https://demo.com`. Use your PennKey as your username.

METHOD	DESCRIPTION
`b = Board(url, username)`	Connect to a messageboard living at the given URL, representing yourself with the chosen `username`.
`b.post(text)`	Post a new message to the messageboard. This message will be publicly visible.
`b.get_recent(n)`	Find a list of the `n` most recently posted messages.
`b.search(keyword)`	Find a list of messages that contain the given `keyword`.

```
b = Board("https://demo.com", "sharry")
```

Also C12 Write another line or two that determines how many users are posting about `"cat"`, and then write a line that creates a new post saying `"There are X posts about cats"`.

FUN(?) ACTIVITY

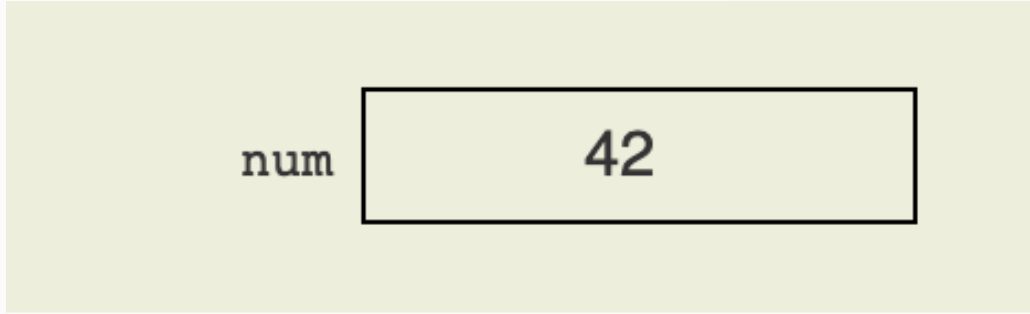
We have a public messageboard server available at:
https://server-demo-dqpp.onrender.com

Copy `messageboard.py` over to Codio from the course website. Then, starting from the template below, can you leave a message on the messageboard?

```
from messageboard import Board
board = # TODO! All the rest
```

VARIABLES, BEFORE

A variable is like a "box" inside of which a piece of data is placed.



VARIABLES, NOW

A variable is a **named portion of memory** that contains data of a particular type.

Variables do not directly contain data. Instead, data is stored in a separate portion of the computer's memory.

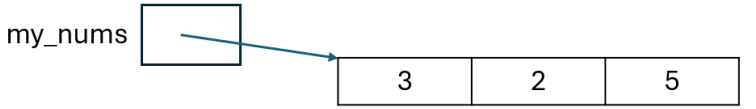
Instead of storing the data directly, variables of these types tell us how to find the data elsewhere!

Let's drill down.

ALL TYPES ARE REFERENCE TYPES

References

- Reference variables **do not store simple values directly!**
- Reference variables store a **reference** to some object
  - Literally: an address that describes where the object is stored in the computer's memory.




```
my_nums = [3]
my_nums.append(2)
my_nums.append(5)
```

MUTABILITY

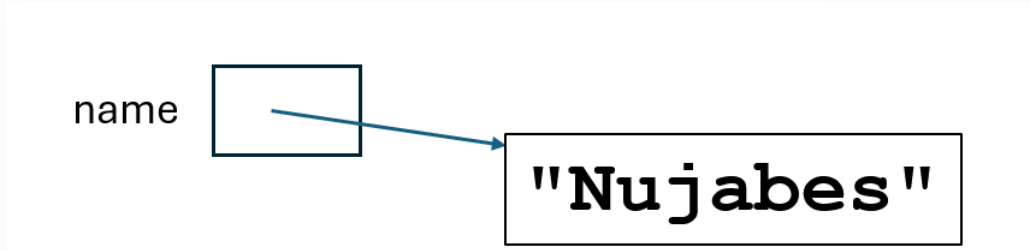
Some types are designed to be immutable types. `string`, `int`, `float`, `bool`, `tuple`*.
Even if we pass a reference to them, we cannot modify them.

```
number = 5
x = number + 3    # number is not changed
number += 2       # equivalent to number = number + 2

name = "Nujabes"
name.upper()          # does nothing, returns a new string "NUJABES"
name = name.upper()   # Reassigns name to a new string
```

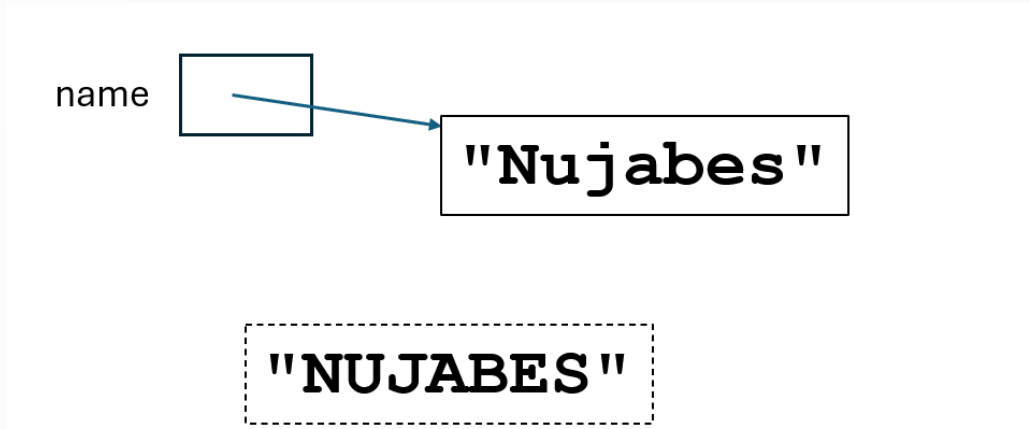
MEMORY DIAGRAM: IMMUTABLE TYPE

```
name = "Nujabes"      <-
name.upper()                # does nothing
name = name.upper()         # Reassigns!
```



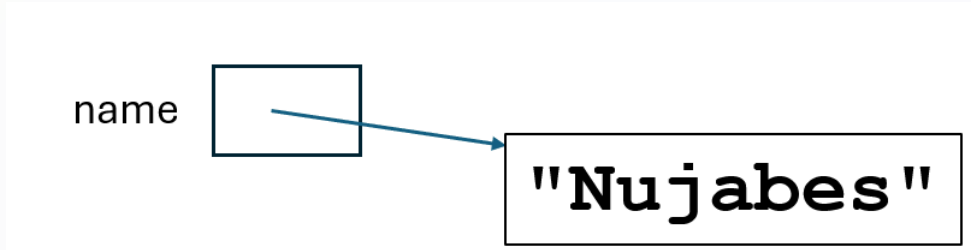
MEMORY DIAGRAM: IMMUTABLE TYPE

```
name = "Nujabes"
name.upper()          <-  # does nothing
name = name.upper()       # Reassigns
```



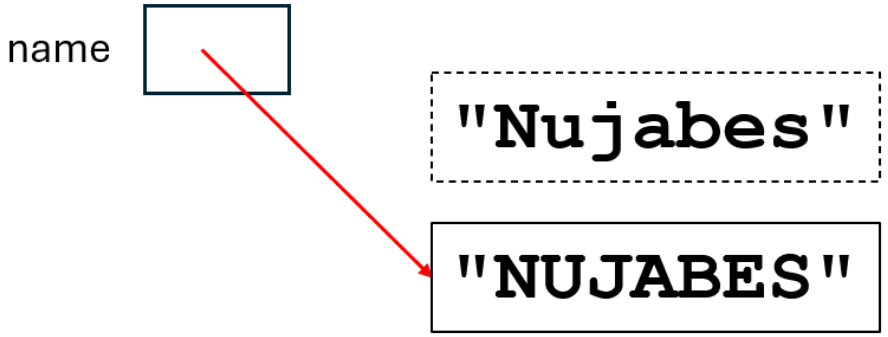
MEMORY DIAGRAM: IMMUTABLE TYPE

```
name = "Nujabes"
name.upper()          <-  # does nothing
name = name.upper()       # Reassigns
```



MEMORY DIAGRAM: IMMUTABLE TYPE

```
name = "Nujabes"
name.upper()                # does nothing,
name = name.upper()  <- # Reassigns name
```

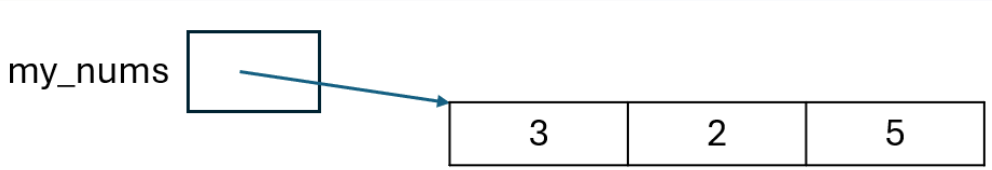


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]      # <-------
    other = my_nums
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```

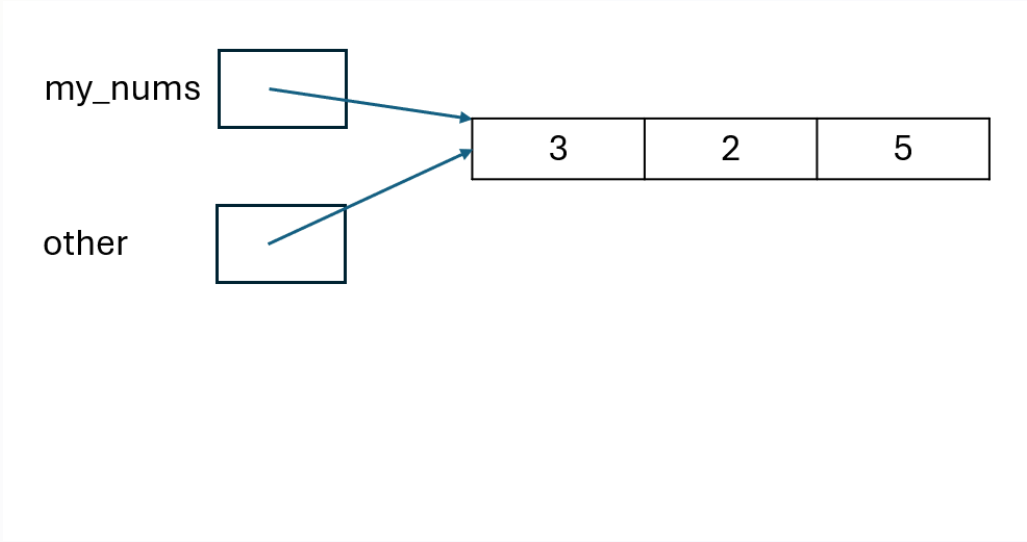


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums          # <-------
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```

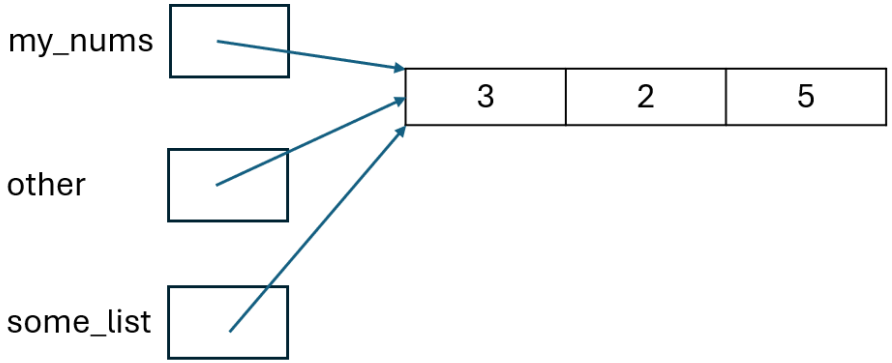


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):          # <-------
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)             # <-------
    other[1] = 1100
    print(my_nums)
```

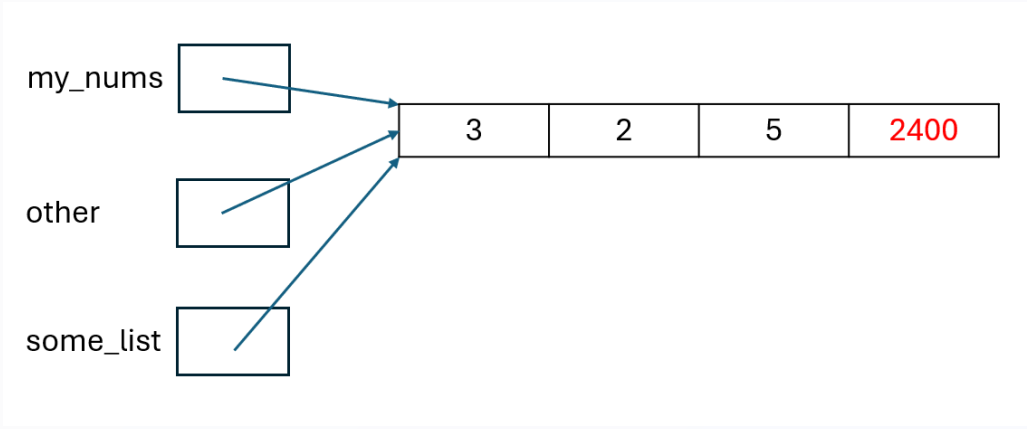


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):
    some_list.append(2400)   # <-------

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)
    other[1] = 1100
    print(my_nums)
```

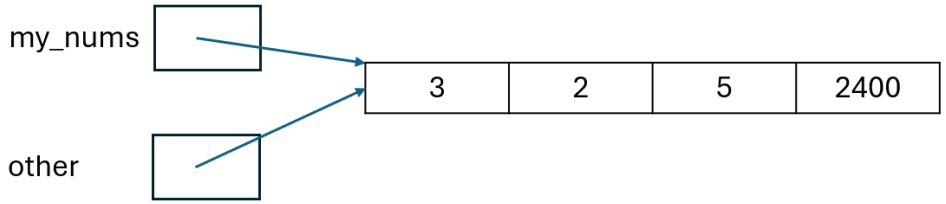


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)              # <--------
    other[1] = 1100
    print(my_nums)
```

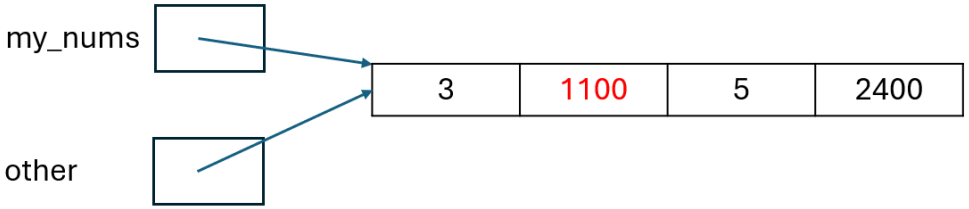


REFERENCES TO MUTABLE TYPES

References get more tricky when we start thinking about mutable types.

```
def func(some_list):
    some_list.append(2400)

def main():
    my_nums = [3, 2, 5]
    other = my_nums
    func(my_nums)
    other[1] = 1100          # <--------
    print(my_nums)
```



PRACTICE

What gets printed?

S8

```
def add_five(num):
    num += 5

def main():
    x = 3
    add_five(x)
    print(x)
```

S9

```
def list_add_five(to_add):
    copy = to_add
    copy.append(copy[0] + 5)

def main():
    my_list = [3]
    list_add_five(my_list)
    print(my_list)
```

PRACTICE

Given a class called `Point` with two fields, `x` and `y`, what gets printed?

(S10)

```
p = Point(x=2024, y=10)  # you can assume this works
not_p = p
not_p.x = 2015
p.x += 2

m = p.y
m += 1

print(p.x)
print(m)
print(p.y)
```

ANSWER

```
b = Board("https://demo.com", "sharry")
cat_posts = b.search("cats")
b.post(f"There are {len(cat_posts)} posts about cats.")
```