CIS 1100

Objects (Using and starting to make them)!

Python
Fall 2024
University of Pennsylvania

Reminder: Voting

The last day to register to vote in Pennsylvania is TODAY. You have until midnight.

You all have lived here for at least a month

- Any US Citizen who has been living in PA for at least
  a month prior to the election can register to vote in PA.
- You do not have to be registered to vote in the state that you came to Penn from.
- By several measures, a vote for president in PA is significantly more impactful than
- A vote in nearly any other state. We also have a closeish Senate race.
  - Not necessarily true for NV, WI
  - Presidential elections are not the only ones; check your House races, too.

Reminders: Late Token

- Late tokens have been updated to account for all lectures (not including today's or Friday's)
- The counts in there is the total amount you have earned. At the end of the semester we will compare this to the amount you used (not calculated yet)
- You may want to calculate these and make use of them.
- You cannot use late tokens on the last assignment (HW9)

Reminders: Exam 2

- There will be a second midterm exam in about a week (November 20th)
- There is also the final exam
    - (and the clobber policy)

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

Yes some of these things could be also achieved from a tuple,

but consider... Which of these is easier to understand the point of:

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

Review: Class as a tool for abstraction

Yes some of this could be also achieved from a tuple, but consider... Which of these is easier to understand the point of:

```
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.

Lets say we wanted to make an object that represented a upenn course, what attributes may we want to store in that class? What types would they be? (L11)

Review: Syntax

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

```
travis_fave = Movie("Pink Floyd - The Wall", 1982, 95, "musical drama", "Vimeo", 0.00)
# not sure I would recommend it to just anyone, but it is my fave

the_name = travis_fave.name
print(travis_fave.name)
if (travis_fave.length > 120):
    print("TOO LONG")
travis.genre = "surrealist " + travis.genre
```

(NOTE: we do not use `()` when accessing fields directly.

`()` is usually used to indicate some sort of function call)

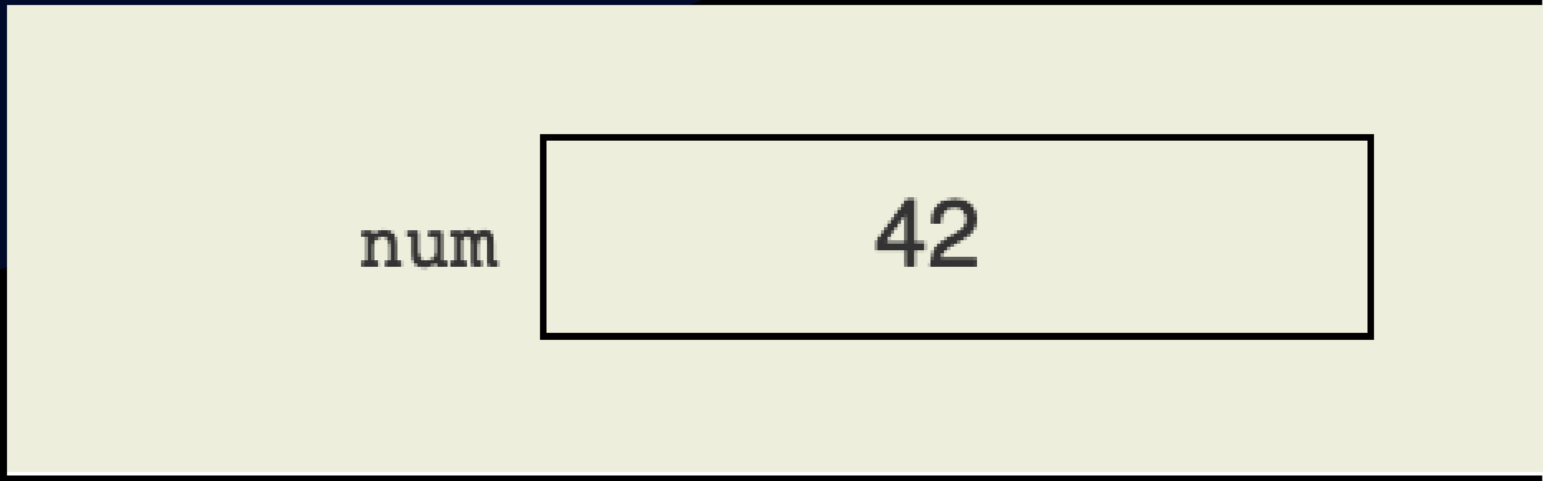
Practice:

Wich of these are (A) method calls, (B) accessing fields, 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`

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!

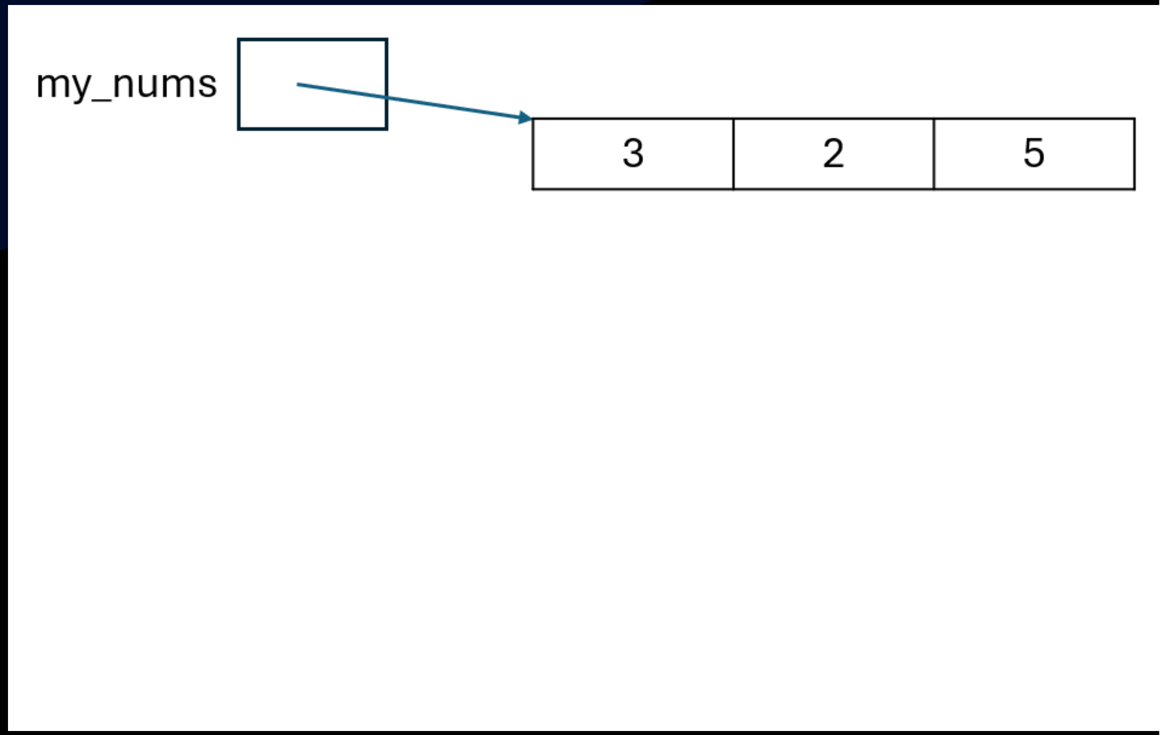
Let's drill down.

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.
- The object that the reference refers to is known as its ***pointee***

```
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.

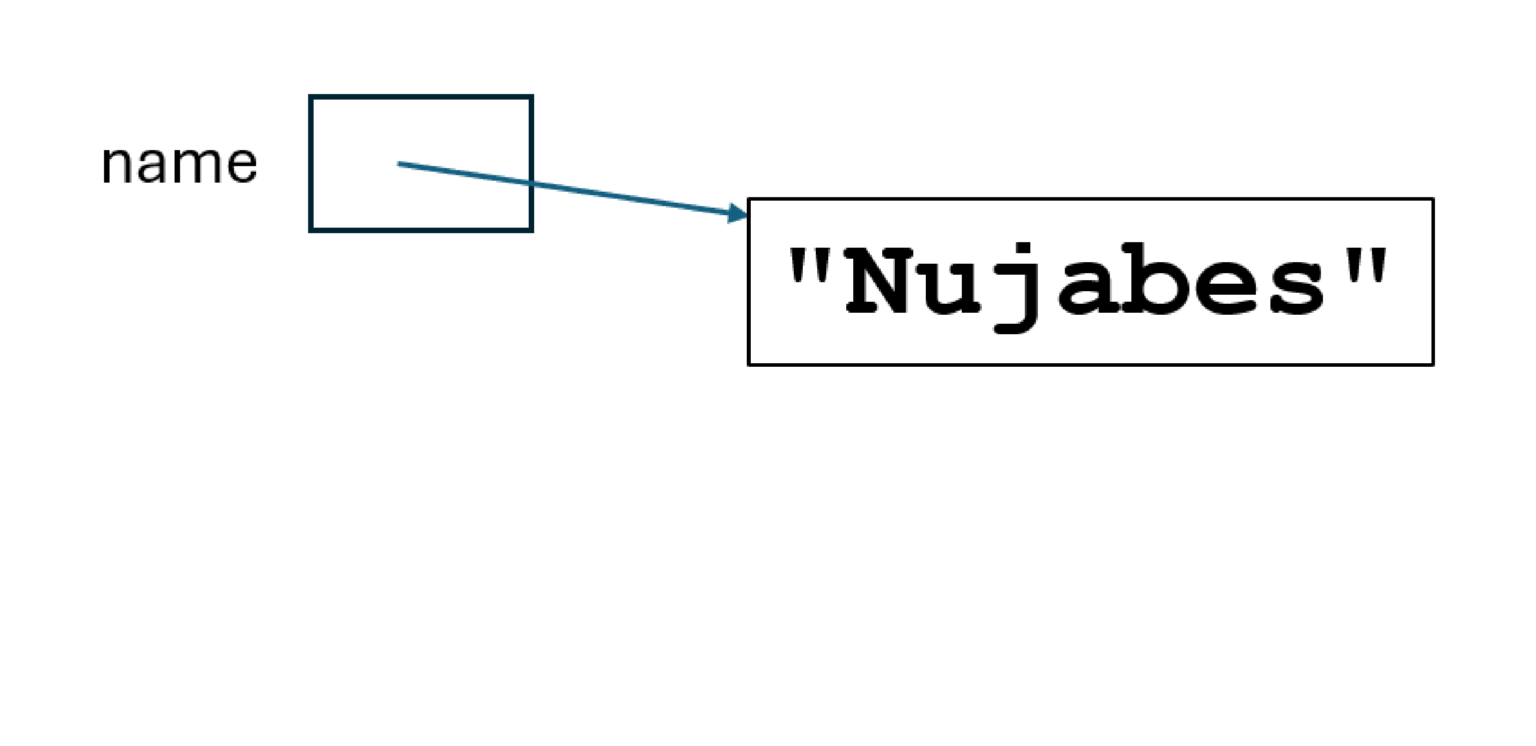
```
number = 5
x = number + 3   # number is not changed, it's value is used as part of a computation
number += 2      # equivalent to number = number + 2, similar to previous line


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

Memory Diagram: Immutable Type

Lets look at the string a little closer

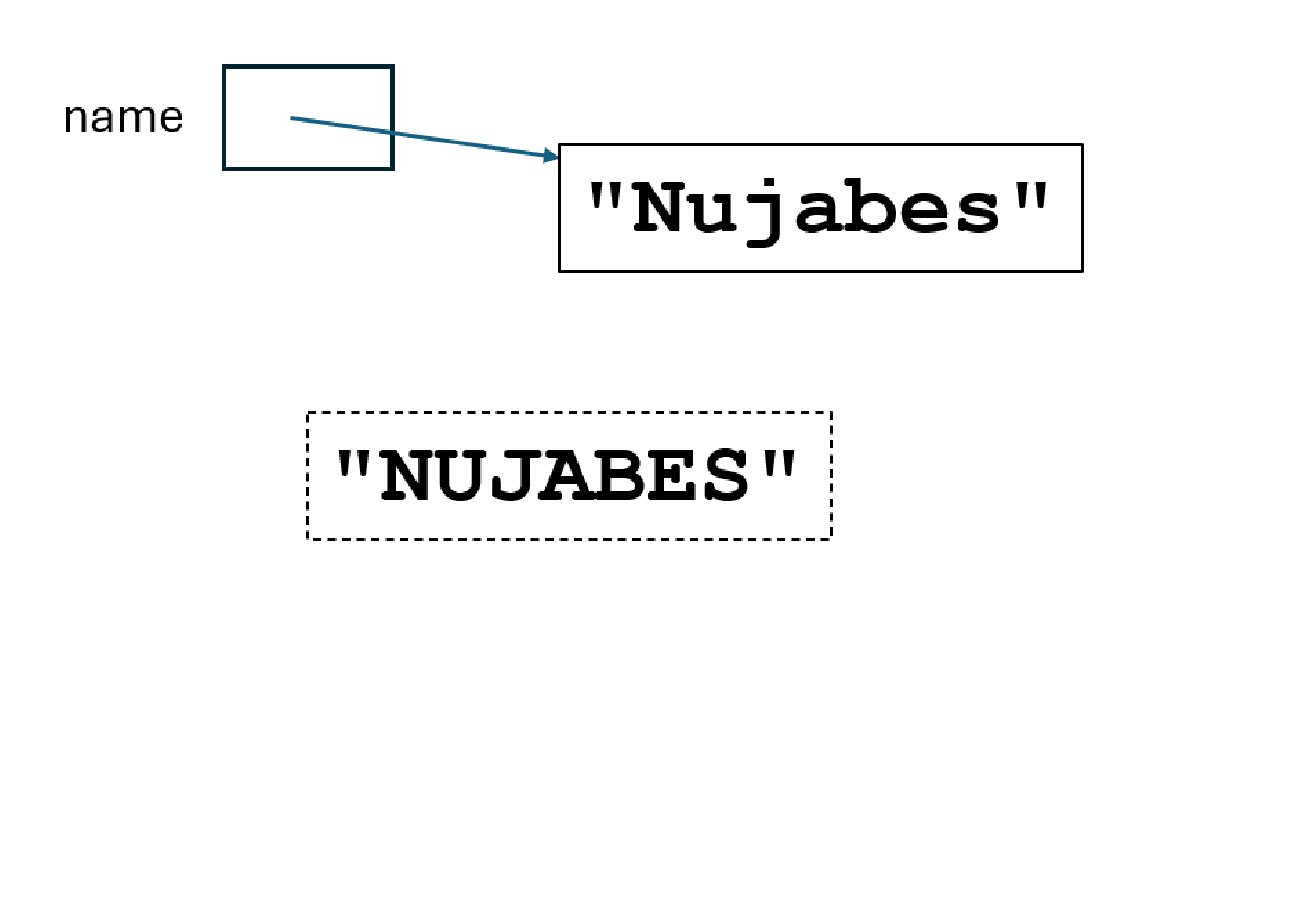
```
name = "Nujabes"    <-
name.upper()            # does nothing, returns a new string "NUJABES"
name = name.upper()     # Reassigns name to a new string
```



Memory Diagram: Immutable Type

Lets look at the string a little closer

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



name

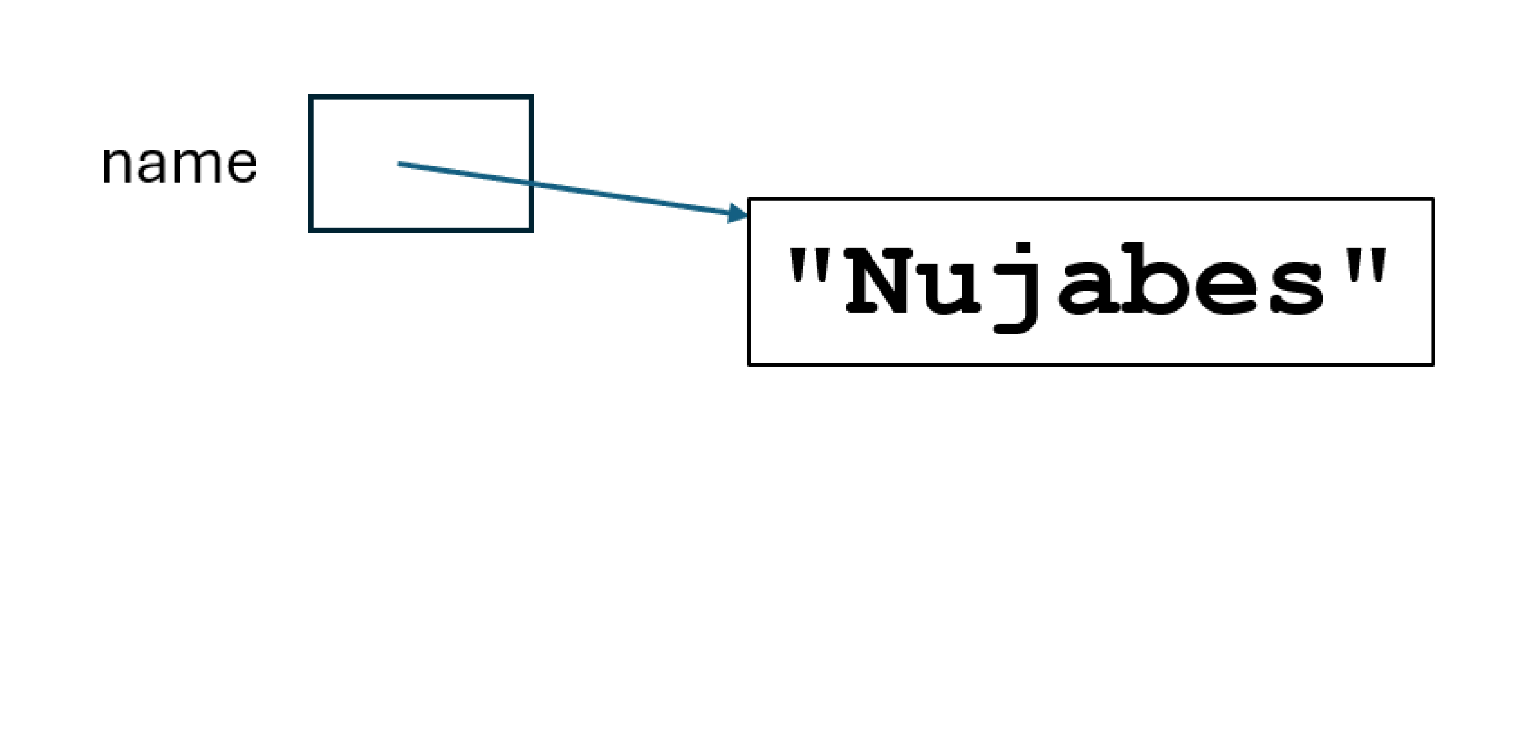
"Nujabes"

"NUJABES"

Memory Diagram: Immutable Type

Lets look at the string a little closer

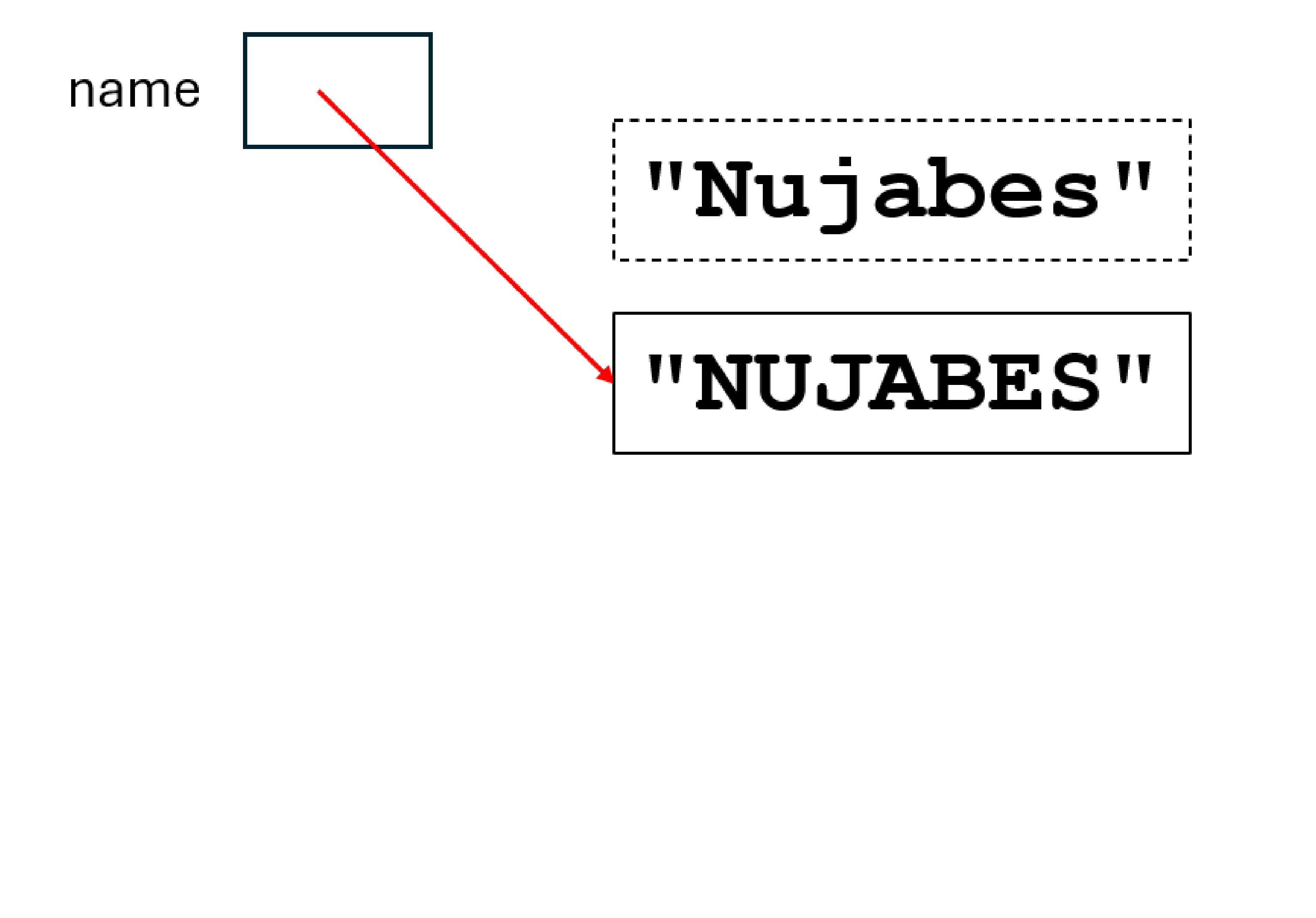
```
name = "Nujabes"
name.upper()        <-  # does nothing, returns a new string "NUJABES"
name = name.upper()     # Reassigns name to a new string
```



Memory Diagram: Immutable Type

Lets look at the string a little closer

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



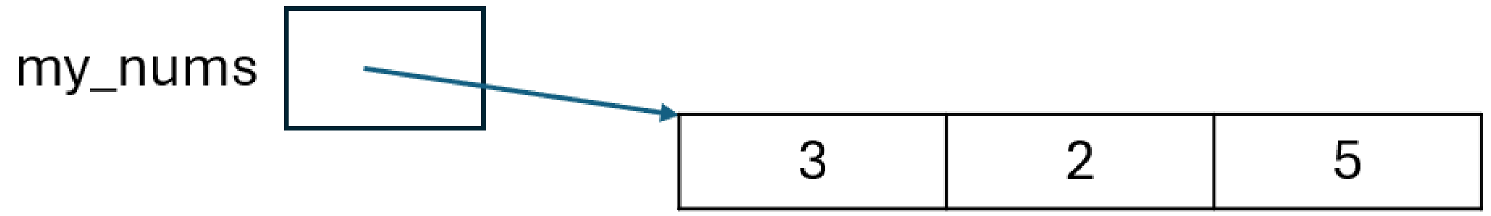
References to Mutable Types

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

Consider:

```
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)
```



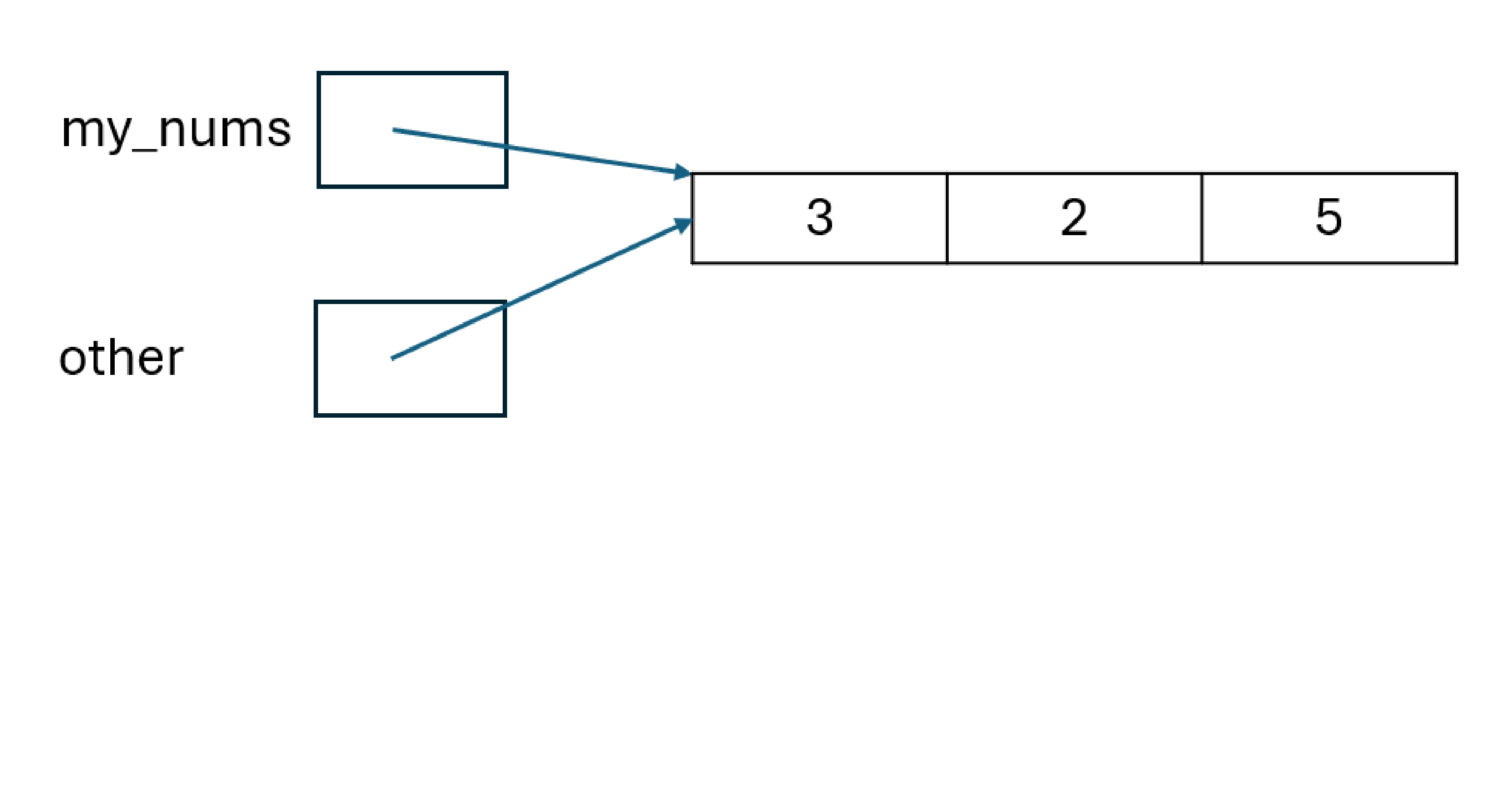
References to Mutable Types

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

Consider:

```
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)
```



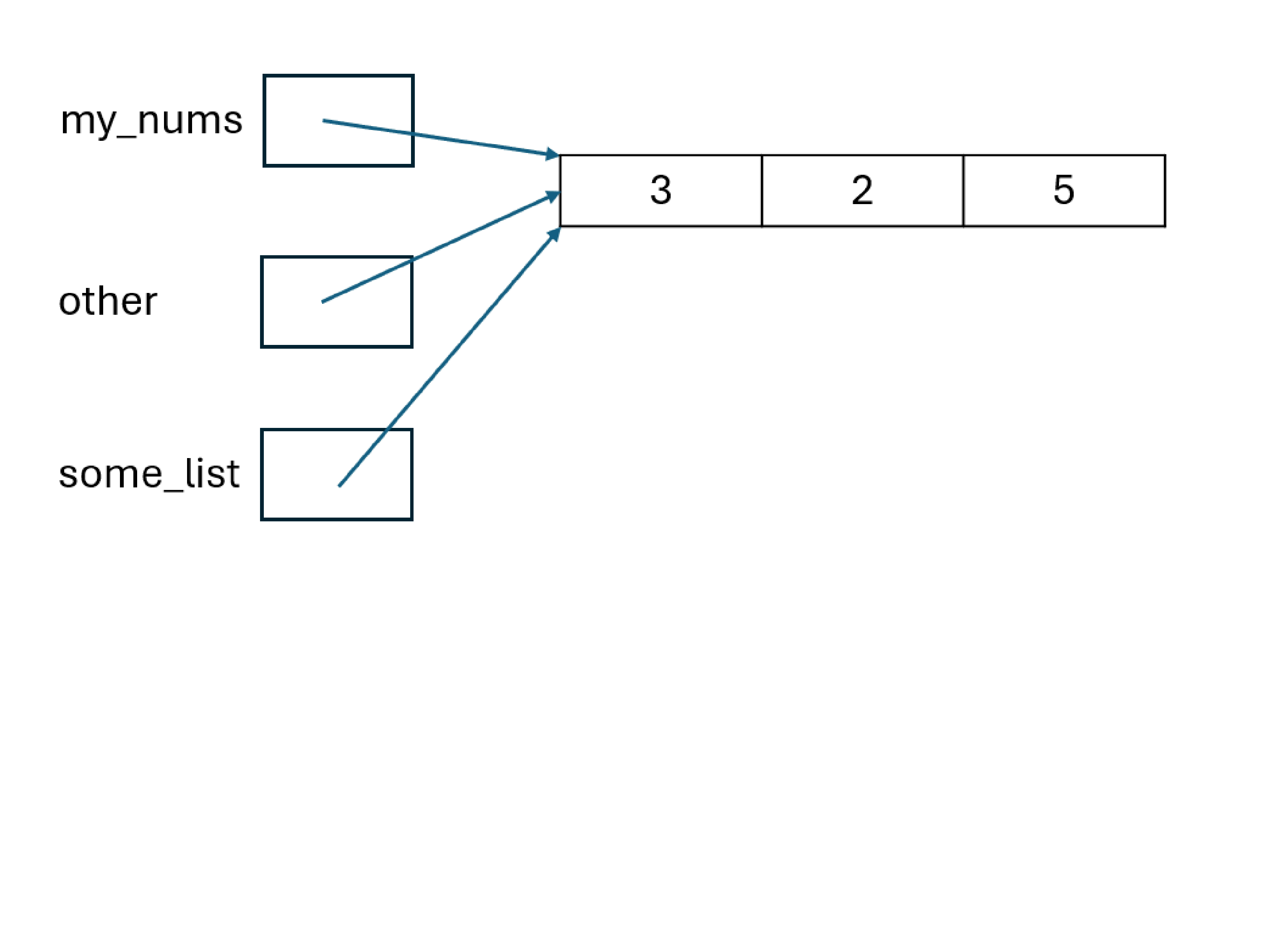
References to Mutable Types

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

Consider:

```
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)
```



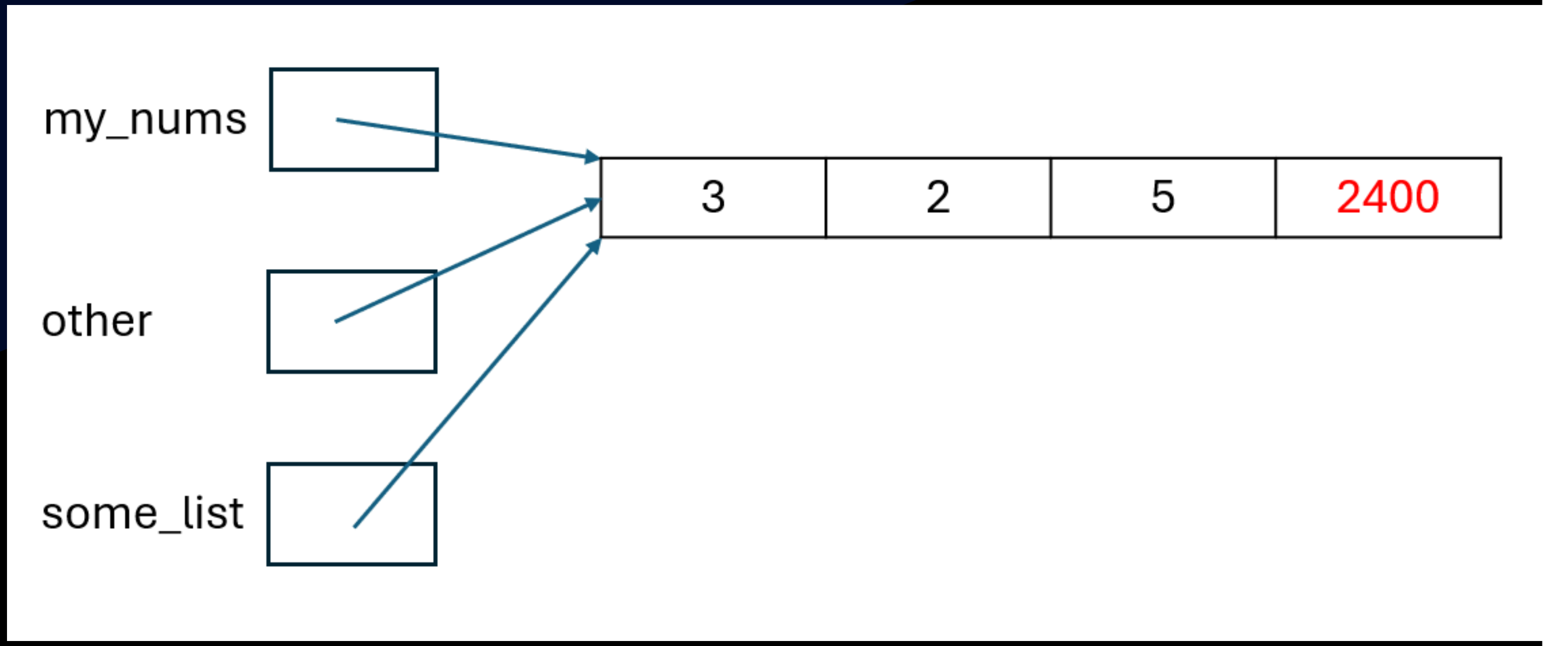
References to Mutable Types

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

Consider:

```
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)
```



my_nums

other

some_list

3	2	5	2400

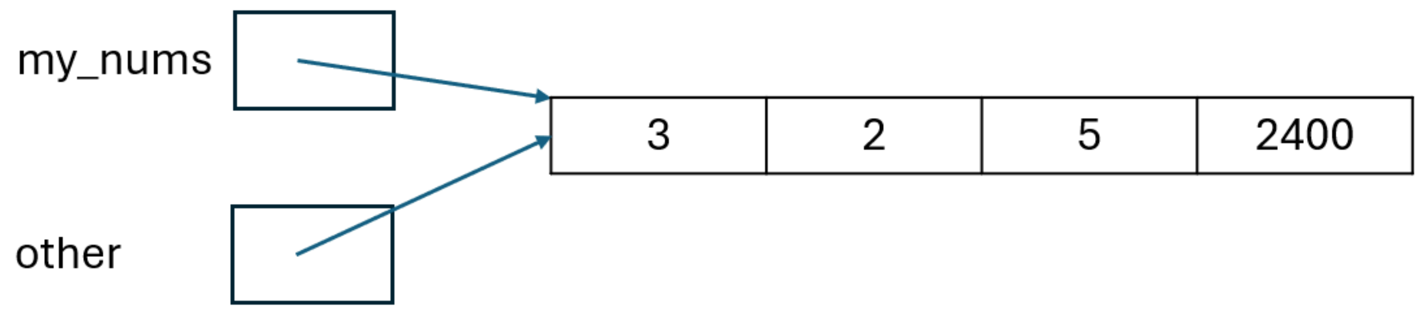
References to Mutable Types

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

Consider:

```
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)
```



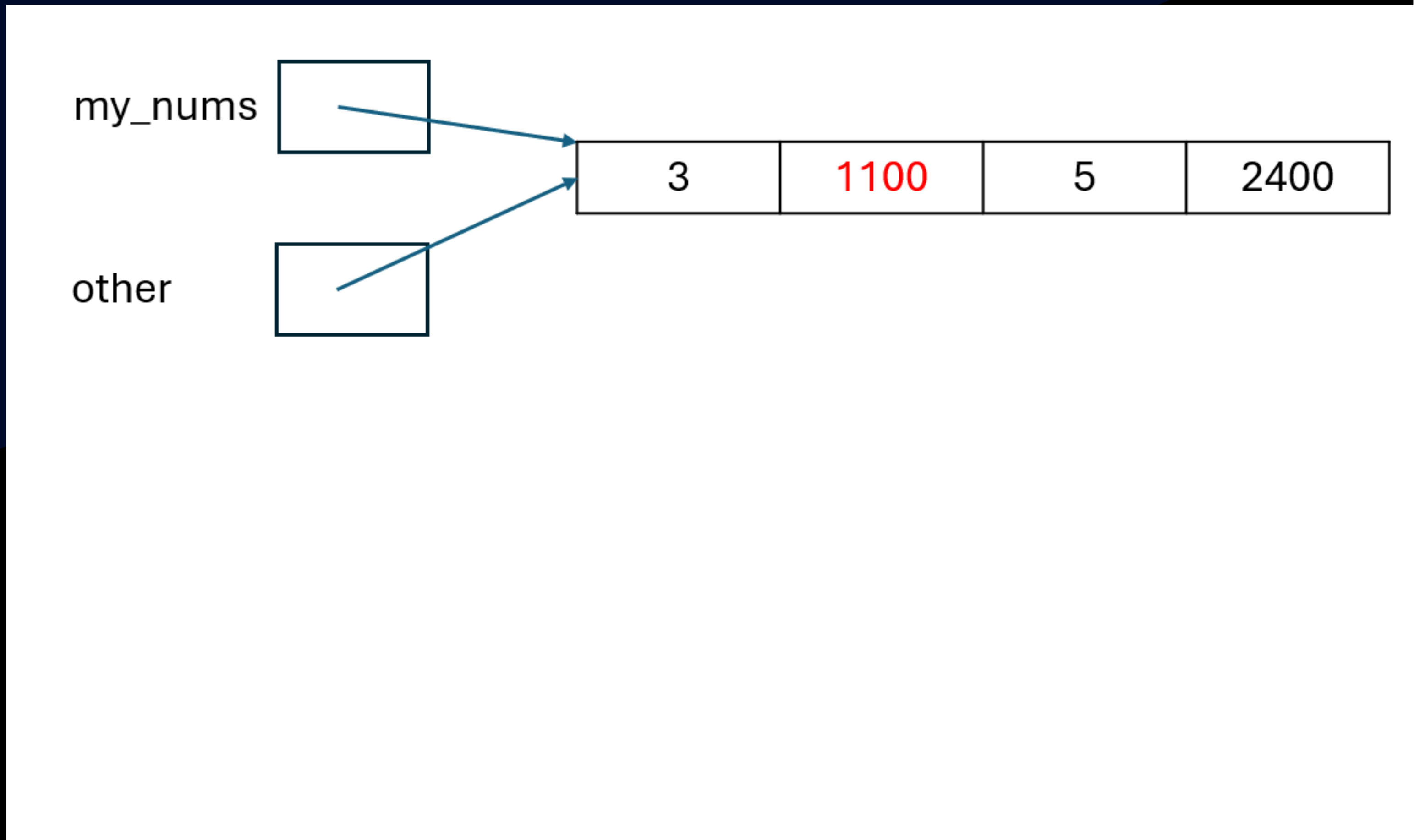
References to Mutable Types

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

Consider:

```
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)
```

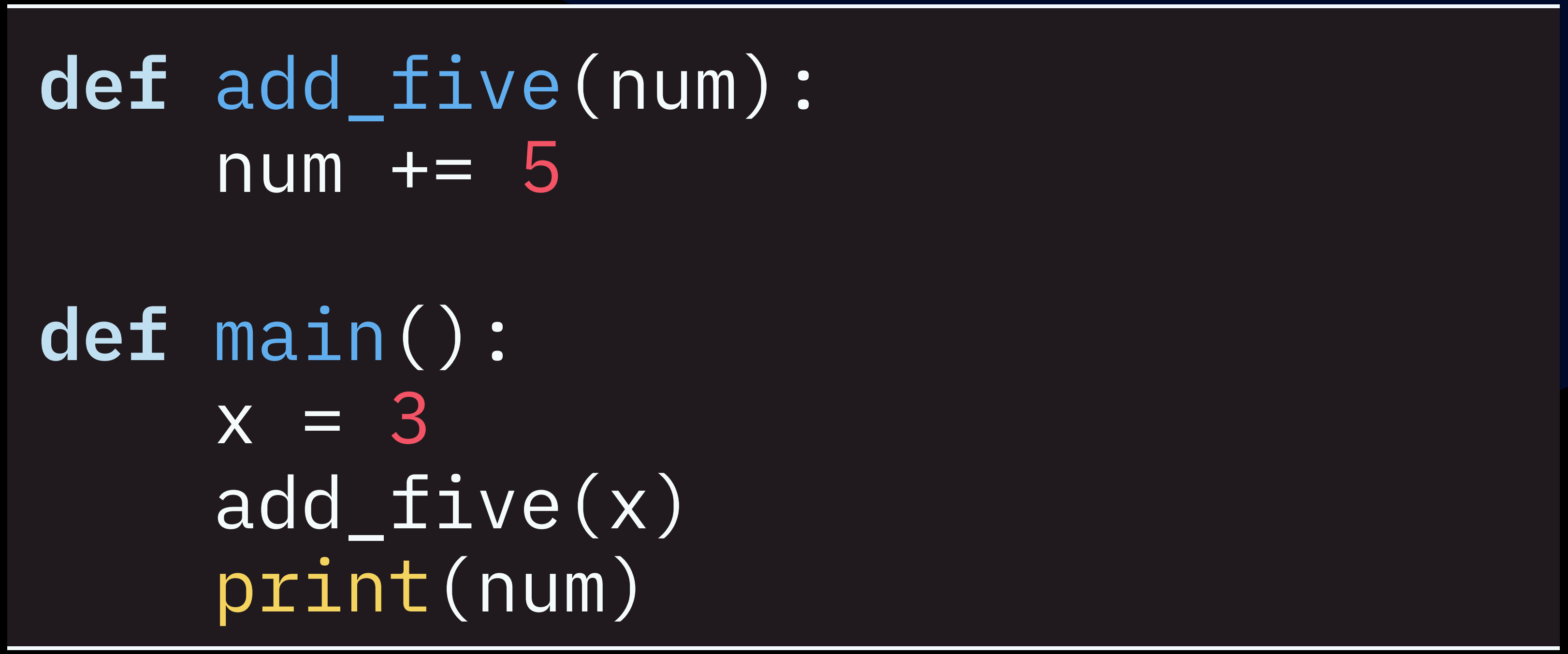


Practice

Given these snippets of code:

what do you think is printed:

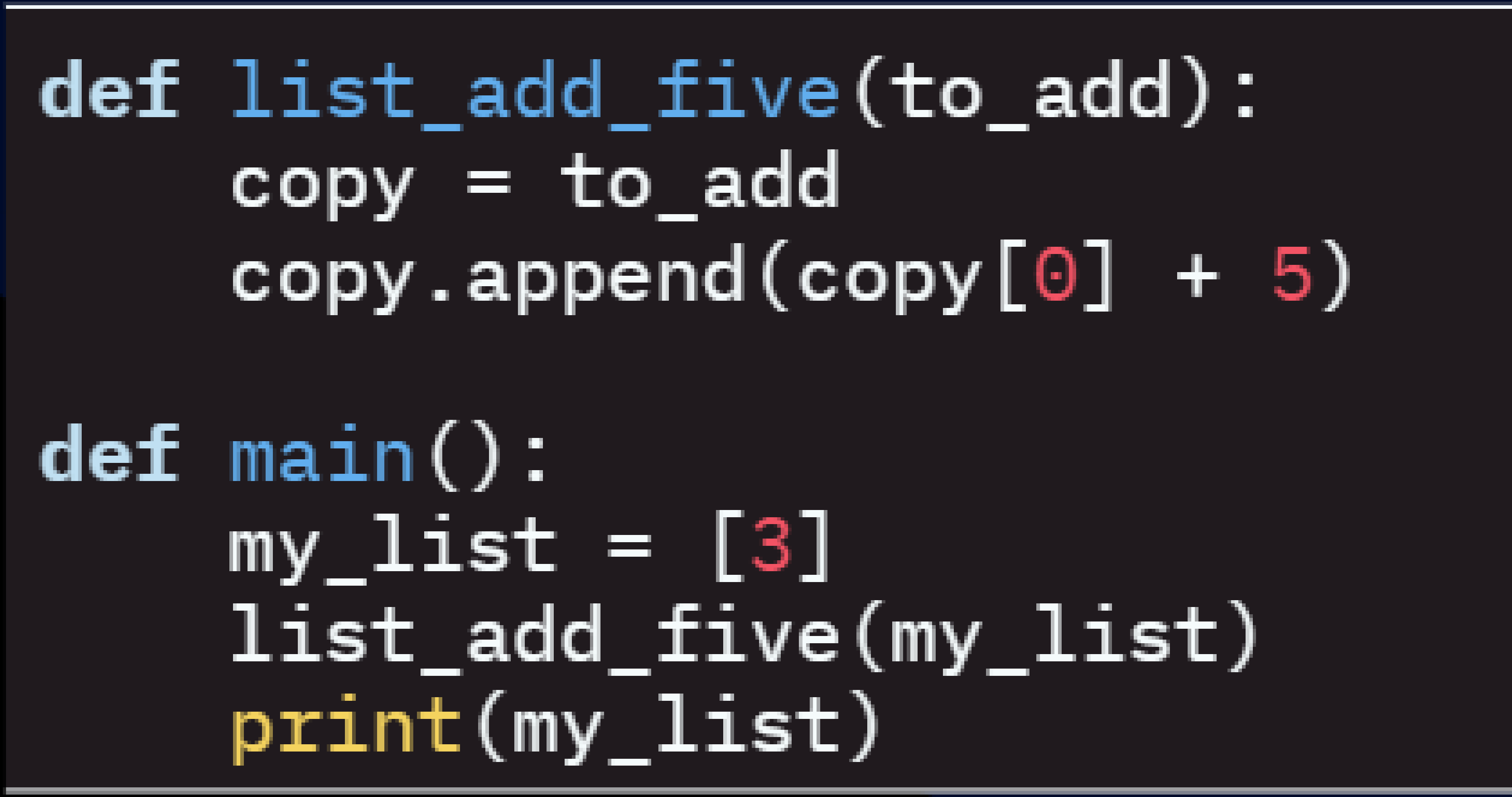
S6



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

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

S7



```
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, an `x` and a `y`, what gets printed?

(S10)

```python
def main():
    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)
```

Review Data Class

If we wanted to make the `Point` object in the previous slide we would do:

```python
from dataclasses import dataclass

@dataclass      # mark the class as a data class
class Point:    # Declare a class
    x: int      # declare the field names and their types
    y: int
```

In Python, a `dataclass` is the simplest kind of class.

- Defined (in most basic case) just by what properties that members of this class should have.

More advanced type notations

If we want to have a data class with more advanced type notations, it would look something like this:

```python
from dataclasses import dataclass

@dataclass
class Example:
    x: list[int]       # list of integers
    y: dict[str, int]  # dictionary, keys are strings, value are ints
    z: tuple[int, int, str]  # a tuple of two ints and a string
```

Practice:

(C12) Write a dataclass that represents a `Square` with three fields:

- a float to represent the `half_width`
- two more `float`s to represent the `center_x` and `center_y`
- a tuple containing three integers to represent the `color`

Next time

- More on objects and creating them!
  - we will exapnd on how to build onto our data class!
  - We will do some code that is VERY relevant for the next homework (FFF)