ECS 32A - User-Defined Classes and References

Aaron Kaloti

UC Davis - Summer Session #1 2020

Overview

- User-defined classes.
- References.
  - These don't solely relate to user-defined classes.

Concepts of This Course: Where User-Defined Classes Fit In

Fundamentals

- Variables.
- Math.
- I/O.
- Conditional statements.
- Iterative statements.

Organizing Information

- Strings.
- Lists.
- Dictionaries.
- Tuples.
- Nested collections.
- *User-defined classes*.

Review: Built-In Types

Examples

```
>>> type(5)
<class 'int'>
>>> type("abc")
<class 'str'>
>>> type(["a","b"])
<class 'list'>
>>> type((3,5))
<class 'tuple'>
>>> type({'k': 'value', 'k2': 'value2'})
<class 'dict'>
```

Defining a Class

- **class**: blueprint for how a new type (that we define) operates.
  - a.k.a. **programmer-defined type**, **user-defined class**.
  - Lets us tell our code about real-world concepts that Python doesn't know about.

Example: `Point` Class

```
>>> class Point:
        pass

>>> p = Point()
>>> p.x = 5
>>> p.y = 8
>>> print(p.x, p.y)
5 8
>>> print("({}, {})".format(p.x,p.y))
(5, 8)
>>> type(p)
<class '__main__.Point'>
```

Example: `Date` Class

- Can (and almost always will) define classes in Python files instead of on Interpreter.

```
class Date:
    pass
```
date_draft1.py

```
>>> d = Date()
>>> type(d)
<class '__main__.Date'>
>>> d.month = 1  # January
>>> d.month = 4  # April
>>> d.day = 20
>>> print(d.month, d.day)
4 20
>>> d2 = Date()
>>> d2.month = 12
>>> d2.day = 17
>>> d2.year = 1996
>>> print(d2.month, d2.day, d2.year)
12 17 1996
```

Terminology

```
class Date:
    pass
```
date_draft1.py

```
>>> d = Date()
>>> type(d)
<class '__main__.Date'>
>>> d.month = 1  # January
>>> d.month = 4  # April
>>> d.day = 20
>>> print(d.month, d.day)
4 20
>>> d2 = Date()
>>> d2.month = 12
>>> d2.day = 17
>>> d2.year = 1996
>>> print(d2.month, d2.day, d2.year)
12 17 1996
```

- `d` and `d2` are **instances** of `class Date`.
- `month` and `day` are class members / data members / fields of `d`.
- `month`, `day`, and `year` are fields of `d2`.
- `d = Date()` is **instantiation** of an instance of `class Date`.

No Enforcement of Names

```
class Date:
    pass
```
date_draft1.py

```
>>> d = Date()
>>> type(d)
<class 'date_draft1.Date'>
>>> d.month = 1
>>> d.day = 20
>>> d.mont = 16
>>> d.mont
16
>>> d.month
1
```

- Python doesn't know what a "mont" is (and doesn't know what a date is) and thus can't tell that "mont" is supposed to be "month".

Initializers: The `__init__()` Method

- An **initializer** allows one to set the fields of a class during instantiation.

Example

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

person.py

- On interpreter:

```
>>> p1 = Person("Shawn", "Thomas", 25)
>>> p2 = Person("Blake", "Stelter", 28)
>>> p3 = Person("Brianna", "Murphy", 27)
>>> print(p1.first_name)
Shawn
>>> print(p2.last_name)
Stelter
>>> print(p3.age)
27
```

Initializers

Same Example, Without Interpreter

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

# Create three Person instances.
p1 = Person("Shawn", "Thomas", 25)
p2 = Person("Blake", "Stelter", 28)
p3 = Person("Brianna", "Murphy", 27)

# Put them in a list.
people = [p1, p2, p3]

# Print.
for person in people:
    print("{} {} is {}".format(person.first_name,
                               person.last_name,
                               person.age))
```

demo_person.py

```
Shawn Thomas is 25
Blake Stelter is 28
Brianna Murphy is 27
```

Initializers

Same Example, Without Interpreter and With `append()`

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

# Create three Person instances in a list.
people = []
people.append(Person("Shawn", "Thomas", 25))
people.append(Person("Blake", "Stelter", 28))
people.append(Person("Brianna", "Murphy", 27))

# Print.
for person in people:
    print("{} {} is {}".format(person.first_name,
                               person.last_name,
                               person.age))
```

demo_person2.py

```
Shawn Thomas is 25
Blake Stelter is 28
Brianna Murphy is 27
```

File Organization / Importing

Example

- Can import a class just as you would import a variable or function from a module.

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

person.py

```
from person import Person

# Create three Person instances in a list.
people = []
people.append(Person("Shawn", "Thomas", 25))
people.append(Person("Blake", "Stelter", 28))
people.append(Person("Brianna", "Murphy", 27))

# Print.
for person in people:
    print("{} {} is {}".format(person.first_name,
                               person.last_name,
                               person.age))
```

use_person.py

```
Shawn Thomas is 25
Blake Stelter is 28
Brianna Murphy is 27
```

File Organization / Importing

- Usually better to have the class definition and associated functions (if any) in a different file than code that uses the class.
  - Someone might want to use the `Person` definition (in `person.py`) without using the code in `use_person.py`.

Example: Why Including Extra Code with Class Definition is Bad

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age

print("I will print whenever you import me!")
print("HAHAHAHA!")
```

bad_person.py

```
from bad_person import Person

p1 = Person("Scott", "Thompson", 58)
print(p1.first_name, p1.last_name, p1.age)
```

use_bad_person.py

- Output (of executing `use_bad_person.py`):

```
I will print whenever you import me!
HAHAHAHA!
Scott Thompson 58
```

Note on File Names

- Avoid putting dashes in file names

Example

- Can't import `Person` from `person-with-dash.py`.

```
class Person:
    def __init__(self, first_name, last_name,
                 age):
        self.first_name = first_name
        self.last_name = last_name
        self.age = age
```

person-with-dash.py

```
from person-with-dash import Person

p1 = Person("Scott", "Thompson", 58)
print(p1.first_name, p1.last_name, p1.age)
```

use-person-with-dash.py

- Output:

```
  File "use-person-with-dash.py", line 1
    from person-with-dash import Person
               ^
SyntaxError: invalid syntax
```

- Although the error is due to the dashes in `person-with-dash.py`, not `use-person-with-dash.py`, you should avoid dashes in all file names for consistency.

Mutability

Example

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
```

point.py

```
>>> p = Point(3,5)
>>> p.x
3
>>> p.x = 8
>>> p.x
8
```

`__str__()`

- Printing an instance of a class normally has unhelpful result.

```
>>> p = Point(3,5)
>>> print(p)
<__main__.Point object at 0x7f61ca883cf8>
```

- If you define the `__str__` method of a class, you can change the effect of printing an instance of that class.

Example

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)
```

point.py

```
>>> p = Point(3,5)
>>> print(p)
(3,5)
```

`__str__()`

Technical Details

- Defining the `__str__` method of a class affects what happens when you use `str()` (the function for converting a given value to a string) on an instance of that class. Behind the scenes, when you `print()` *any* value, `print()` first attempts to convert that value to a string.

Example

```
class Class1:
    def __init__(self,x):
        self.x = x

class Class2:
    def __init__(self,x):
        self.x = x

    def __str__(self):
        return "Class2 has x={}".format(self.x)

c1 = Class1(5)
print("c1 is", c1)
c2 = Class2(8)
print("c2 is", c2)
```

```
c1 is <__main__.Class1 object at 0x7f2bfa184860>
c2 is Class2 has x=8
```

`__str__()`

Technical Details

- Can use `str()` explicitly.
  - Akin to converting `8` to `"8"` before printing that integer, whereas `print(8)` would already convert `8` to a string.

Example

```
class Class1:
    def __init__(self,x):
        self.x = x

class Class2:
    def __init__(self,x):
        self.x = x

    def __str__(self):
        return "Class2 has x={}".format(self.x)

c1 = Class1(5)
c1_str = str(c1)
print("c1 is", c1_str)
c2 = Class2(8)
c2_str = str(c2)
print("c2 is", c2_str)
```

```
c1 is <__main__.Class1 object at 0x7f2bfa184860>
c2 is Class2 has x=8
```

`__str__()`

Technical Details: Writing to a File

- `write()` doesn't attempt to convert argument to a string like `print()` does.

Example

```
class Class2:
    def __init__(self,x):
        self.x = x

    def __str__(self):
        return "Class2 has x={}".format(self.x)

c2 = Class2(8)
f = open("foo.txt","w")
f.write(c2)
f.close()
```

```
Traceback (most recent call last):
  File "/home/aaronistheman/.11demos/write_class_to_file.py", line 24, in <module>
    f.write(c2)
TypeError: write() argument must be str, not Class2
```

`__str__()`

Technical Details: Writing to a File

- Must use `str()` explicitly.

Example (Fixed)

```
class Class1:
    def __init__(self,x):
        self.x = x

class Class2:
    def __init__(self,x):
        self.x = x

    def __str__(self):
        return "Class2 has x={}".format(self.x)

c1 = Class1(5)
c2 = Class2(8)

f = open("foo.txt","w")
f.write(str(c1) + "\n")
f.write(str(c2) + "\n")
f.close()
```

write_class_to_file.py

- `foo.txt` after:

```
<__main__.Class1 object at 0x7f6eb3f29860>
Class2 has x=8
```

Methods

- **method** (formal definition): function defined inside of a `class`.

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)
```

point.py

- `__init__()` and `__str__()` are methods, although they're special methods that needn't be called with the *`object.method_name(arguments)`* notation.

Methods

Example: `Date` Class

- Can add more methods to make our `Point` class more useful.

```
from math import sqrt

class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)

    # Compute distance from origin.
    def distance(self):
        return sqrt(self.x ** 2 + self.y ** 2)
```

point.py

```
>>> p = Point(3,4)
>>> p.distance()
5.0
>>> p2 = Point(6,8)
>>> p2.distance()
10.0
>>> p2.y = 9
>>> p2.distance()
10.816653826391969
```

self

- `self` refers to the instance on which a method is being called[1]. For example, when we do `p.distance()`, the argument `self` will refer to `p`. This may seem odd, because we didn't do `distance(p)`, but that is merely how the syntax for methods works. Your book calls `p` the **subject**.

```
from math import sqrt

class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)

    # Compute distance from origin.
    def distance(self):
        return sqrt(self.x ** 2 + self.y ** 2)
```

point.py

```
>>> p = Point(3,4)
>>> p.distance()
5.0
```

1. The name `self` is a *convention*, which means you can violate it if you want, but it would be bad form, and your coworkers won't like you as much.

Methods: Setters/Modifiers

- Functions that allow changing fields in more controlled manner.
- More important in languages like C++ or Java that permit private fields.

Example

```
from math import sqrt

class Point:
    ...

    # Compute distance from origin.
    def distance(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def add(self,delta_x,delta_y):
        self.x += delta_x
        self.y += delta_y
```

point.py

```
>>> p = Point(3,4)
>>> print(p)
(3,4)
>>> p.add(2,-1)
>>> print(p)
(5,3)
```

Methods: `other`[1]

- You can also define a method that takes as argument another instance.

```
from math import sqrt

class Point:
    ...

    def distance_between(self,other):
        x_dist2 = (self.x - other.x) ** 2
        y_dist2 = (self.y - other.y) ** 2
        return sqrt(x_dist2 + y_dist2)
```

point.py

```
>>> p1 = Point(3,2)
>>> p2 = Point(7,8)
>>> p1.distance_between(p2)
7.211102550927978
>>> p2.distance_between(p1)
7.211102550927978
```

1. Again, convention is to call it `other`.

Input Validation

- The initializer (`__init__()`) provides us with an easy way to perform input validation on each new instance of the class. The convention is to throw a `ValueError` if an invalid argument is passed to the initializer[1].

```
class Date:
    def __init__(self,year,month,day):
        self.year = year
        if month < 1 or month > 12:
            raise ValueError("Invalid month")
        self.month = month
        self.day = day
```

date.py

```
>>> d = Date(2015,5,11)
>>> d2 = Date(2099,15,98)
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    d2 = Date(2099,15,98)
  File "/home/aaronistheman/.11demos/date.py", line 5, in __init__
    raise ValueError("Invalid month")
ValueError: Invalid month
```

1. Source: https://docs.python.org/3/library/exceptions.html#ValueError

Example: Increment Day

Prompt

- Add a `increment_day` method to the `Date` class. This method should increment the day by 1 and carry on over to the next month (or even the next year) if necessary. Ignore leap years.
- Also, add input validation for the day in the initializer.

Solution

```
class Date:
    def __init__(self,year,month,day):
        self.year = year
        if month < 1 or month > 12:
            raise ValueError("Invalid month")
        self.month = month

        self.days_per_month = [31, 28, 31, 30, 31, 30,
                               31, 31, 30, 31, 30, 31]

        if day > self.days_per_month[self.month - 1] or \
            day <= 0:
            raise ValueError("Invalid day")
        self.day = day

    def __str__(self):
        return "{}/{}/{}".format(self.year, self.month, self.day)
...
```

Example: Increment Day

Solution (Continued)

```
...
    # Returns true if day is too high; otherwise, false.
    def day_too_high(self):
        return self.day > self.days_per_month[self.month - 1]
        # if day > self.days_per_month[month - 1]:
            # return True
        # else:
            # return False

    def increment_day(self):
        self.day += 1
        # If day is past end of month.
        if self.day_too_high():
            self.day = 1
            self.month += 1
            # If month is past end of year.
            if self.month > 12:
                self.month = 1
                self.year += 1
```

Example: Small "Game"

Driver Code

```
from player import Player
from species import read_species

species = read_species("species.txt")
print("=== The species ===")
for s in species:
    print(s)
print()

p1 = Player("urBadAtThisGame", species[0], 200, 50)
p2 = Player("coolguy123", species[1], 70, 100)
p3 = Player("cornpop", species[1], 215, 60)
p4 = Player("drakeIsPopMusic", species[2], 70, 130)
...
```

playground.py

```
leto 100 10 15
zordborg 70 6 40
fortwat 80 12 10
```

species.txt

- Output of above part:

```
=== The species ===
Name: leto, Base health: 100, Base damage: 10, Attack range: 15
Name: zordborg, Base health: 70, Base damage: 6, Attack range: 40
Name: fortwat, Base health: 80, Base damage: 12, Attack range: 10

...
```

Example: Small "Game"

Driver Code

```
...
def print_within_range(player1, player2):
    print("{} is within range of {}: {}".format(
        player1.username, player2.username, player1.within_range(player2)))

print_within_range(p1, p2)
print_within_range(p1, p3)
print_within_range(p1, p4)
...
```

playground.py

- Output of above part:

```
...
urBadAtThisGame is within range of coolguy123: False
urBadAtThisGame is within range of cornpop: True
urBadAtThisGame is within range of drakeIsPopMusic: False
...
```

Example: Small "Game"

Driver Code

```
...
print("Before P4 attacks P1: {} current health: {}".format(
    p1.username, p1.curr_health))
p4.attack(p1)
print("After P4 attacks P1: {} current health: {}".format(
    p1.username, p1.curr_health))

print("Before P2 attacks P4: {} current health: {}".format(
    p4.username, p4.curr_health))
p2.attack(p4)
print("After P2 attacks P4: {} current health: {}".format(
    p4.username, p4.curr_health))
...
```

- Output of above part:

```
...
Before P4 attacks P1: urBadAtThisGame current health: 100
After P4 attacks P1: urBadAtThisGame current health: 100
Before P2 attacks P4: drakeIsPopMusic current health: 80
After P2 attacks P4: drakeIsPopMusic current health: 74
...
```

Example: Small "Game"

Driver Code

```
...
# Have P3 continuously attack P1 until P1 runs out of health.
print("P3 continuously attacks P1.")
while not p3.attack(p1):
    print("{} current health: {}".format(p1.username, p1.curr_health))
print("{} has run out of health.".format(p1.username))
```

playground.py

- Output of above part:

```
P3 continuously attacks P1.
urBadAtThisGame current health: 94
urBadAtThisGame current health: 88
urBadAtThisGame current health: 82
urBadAtThisGame current health: 76
urBadAtThisGame current health: 70
urBadAtThisGame current health: 64
urBadAtThisGame current health: 58
urBadAtThisGame current health: 52
urBadAtThisGame current health: 46
urBadAtThisGame current health: 40
urBadAtThisGame current health: 34
urBadAtThisGame current health: 28
urBadAtThisGame current health: 22
urBadAtThisGame current health: 16
urBadAtThisGame current health: 10
urBadAtThisGame current health: 4
urBadAtThisGame has run out of health.
```

Example: Small "Game"

`Point` Class

```
from math import sqrt

class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y

    def __str__(self):
        return "({},{})".format(self.x,self.y)

    # Compute distance from origin.
    def distance(self):
        return sqrt(self.x ** 2 + self.y ** 2)

    def add(self,delta_x,delta_y):
        self.x += delta_x
        self.y += delta_y

    def change(self,new_x,new_y):
        self.x = new_x
        self.y = new_y

    def distance_between(self,other):
        x_dist2 = (self.x - other.x) ** 2
        y_dist2 = (self.y - other.y) ** 2
        return sqrt(x_dist2 + y_dist2)
```

Example: Small "Game"

Prompt: Finish Incomplete Portions

- Below is an incomplete version of `species.py`. On the next slide is an incomplete version of `player.py`. Complete these Python files so that the output of `playground.py` is as shown earlier.

```
# TODO: Imports?

class Species:
    # Assumes arguments are already integers.
    def __init__(self, name, base_health, base_dmg, atk_range):
        # TODO: Initialize members.

    def __str__(self):
        # TODO: Implement.

# NOT a method.
# Apparently, the plural of "species" is "species".
def read_species(filename):
    # TODO: Implement.
```

species.py

Example: Small "Game"

Prompt: Finish Incomplete Portions

```
# TODO: Imports?

class Player:
    def __init__(self, username, species, x, y):
        # TODO: Initialize members.

    # @self attacks @other, if within range.
    # Returns True if the attack results in @other running out of health.
    def attack(self, other):
        # TODO: Implement.

    def change_position(self, x, y):
        # We won't implement this method.

    # Returns player to full health.
    def return_full_health(self):
        # We won't implement this method.

    # Returns true if @self runs out of health.
    def take_damage(self, dmg):
        # TODO: Implement.

    # Returns true if @self is within @other's attack range.
    def within_range(self, other):
        # TODO: Implement.
```

player.py

Example: Small "Game"

Solution: Completed `species.py`

```
class Species:
    # Assumes arguments are already integers.
    def __init__(self, name, base_health, base_dmg, atk_range):
        self.name = name
        self.base_health = base_health
        self.base_dmg = base_dmg
        self.atk_range = atk_range

    def __str__(self):
        return "Name: {}, Base health: {}, Base damage: {},"\
               " Attack range: {}"\
               .format(self.name, self.base_health,
                       self.base_dmg, self.atk_range)

# NOT a method.
# Apparently, the plural of "species" is "species".
def read_species(filename):
    f = open(filename)
    species = []
    for line in f:
        vals = line.split()  # e.g. ["leto", "100", "10", "15"]
        new_species = Species(vals[0], int(vals[1]),
                              int(vals[2]), int(vals[3]))
        # species += [new_species]
        species.append(new_species)
        # species.extend([new_species])
    f.close()
    return species
```

species.py

Example: Small "Game"

Solution: Completed `player.py`

```
from point import Point

class Player:
    def __init__(self, username, species, x, y):
        self.username = username
        self.species = species
        self.location = Point(x, y)
        self.curr_health = self.species.base_health

    # @self attacks @other, if within range.
    # Returns True if the attack results in @other running out of health.
    def attack(self, other):
        # If @other is within range of @self.
        if other.within_range(self):
            retval = other.take_damage(self.species.base_dmg)
            return retval
        else:
            # Attack failed... not in range.
            return False

    # def change_position(self, x, y):
        # pass

    # Returns player to full health.
    # def return_full_health(self):
        # self.curr_health = self.species.base_health
...
```

Example: Small "Game"

Solution: Completed `player.py` (Continued)

```
...
    # @self takes @dmg amount of @dmg, which is an int.
    # Returns true if @self runs out of health.
    def take_damage(self, dmg):
        self.curr_health -= dmg
        return self.curr_health <= 0

    # Returns true if @self is within @other's attack range.
    def within_range(self, other):
        # TODO: Implement.
        if self.location.distance_between(other.location) \
            <= other.species.atk_range:
            return True
        else:
            return False
```

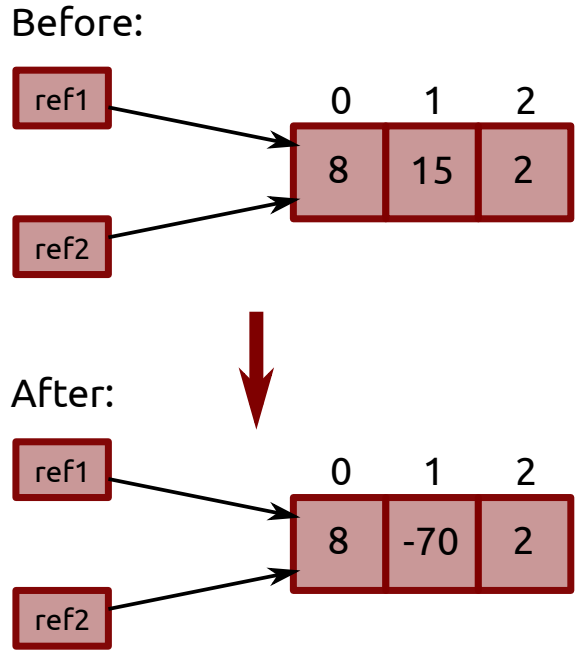
player.py

References

- Generically, we'll refer to any lists, strings, dictionaries, tuples, class instances, etc. as *objects*.
- Saying a variable's value is an object means the variable *references* that object.
  - While speaking, anyone would probably just say the variable has that value and not mention references. References help in reasoning about certain behaviors that may be unexpected to anyone lacking understanding of references.

Example

```
>>> ref1 = [8, 15, 2]
>>> ref2 = ref1
>>> ref2[1] = -70
>>> ref2
[8, -70, 2]
>>> ref1
[8, -70, 2]
```



References

`is` Operator

- Checks if two variables reference same object.

Example

```
>>> ref1 = [8, 15, 2]
>>> ref2 = ref1
>>> ref1 is ref2
True
>>> ref2 is ref1
True
>>> d1 = {'bob': 30, 'rick': 32, 'james': 40}
>>> d2 = d1
>>> d2['bob'] = 20
>>> d2['jordan'] = 35
>>> d2
{'bob': 20, 'rick': 32, 'james': 40, 'jordan': 35}
>>> d1
{'bob': 20, 'rick': 32, 'james': 40, 'jordan': 35}
>>> d1 is d2
True
```

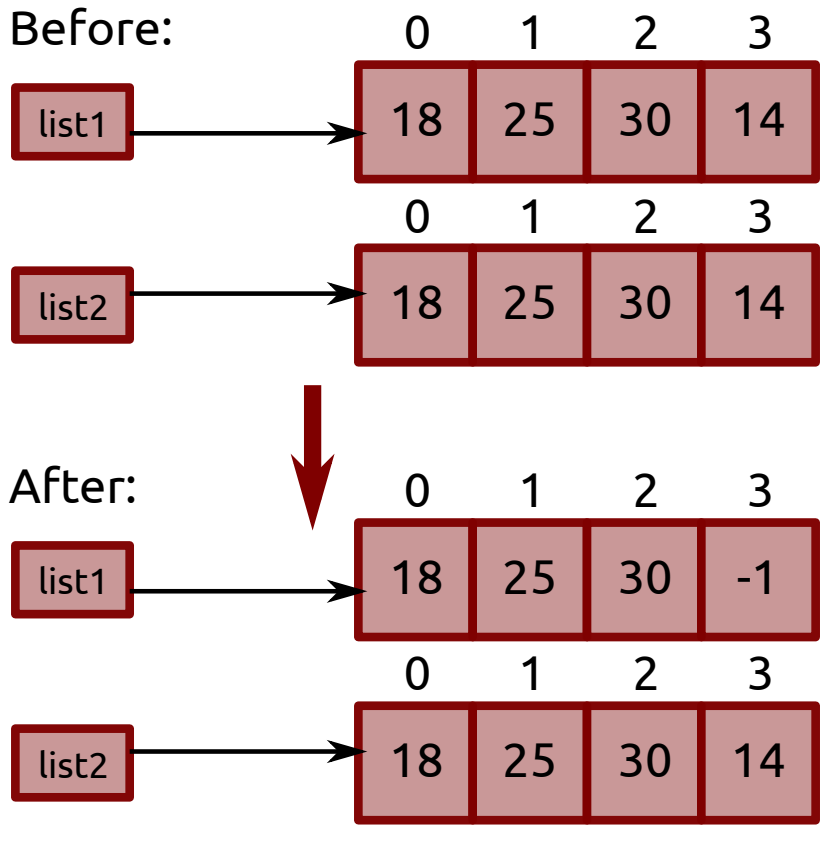
References

`is` Operator

- Two variables can reference two different objects that have the same values; this doesn't make them references to the same object.

Example

```
>>> list1 = [18, 25, 30, 14]
>>> list2 = [18, 25, 30, 14]
>>> list1[3] = -1
>>> list1
[18, 25, 30, -1]
>>> list2
[18, 25, 30, 14]
>>> list1 is list2
False
```



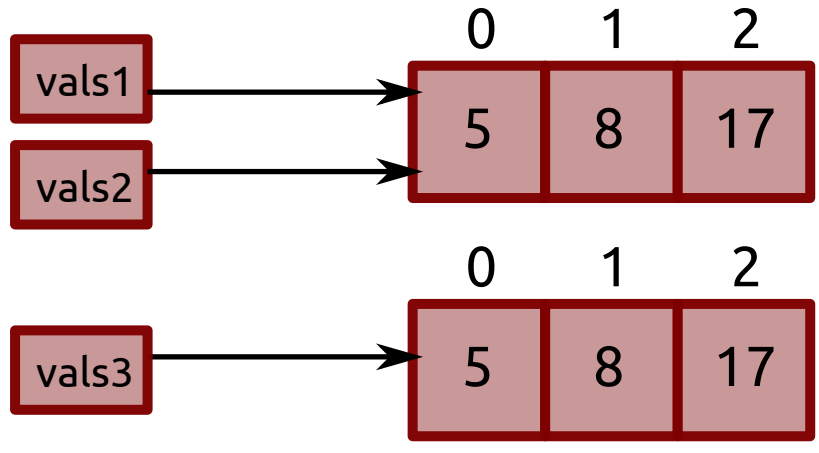
References

`is` Operator vs. `==`

- `==` will check for same values.

Example

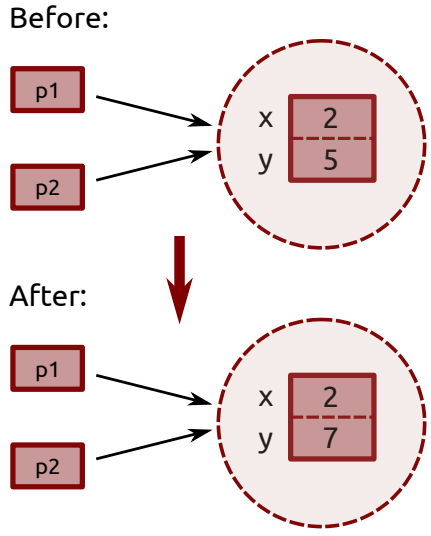
```
>>> vals1 = [5, 8, 17]
>>> vals2 = vals1
>>> vals3 = [5, 8, 17]
>>> vals1 is vals2
True
>>> vals1 == vals2
True
>>> vals1 is vals3
False
>>> vals1 == vals3
True
```



References

Example: Class Instances

```
>>> p1 = Point(2,5)
>>> p2 = p1
>>> p2.y = 7
>>> print(p1)
(2,7)
>>> print(p2)
(2,7)
>>> p1 is p2
True
```



- Again, having fields with same values doesn't mean reference same object:

```
>>> p3 = Point(2,7)
>>> print(p1)
(2,7)
>>> print(p3)
(2,7)
>>> p1 is p3
False
>>> p3.x = 15
>>> print(p1)
(2,7)
>>> print(p3)
(15,7)
```

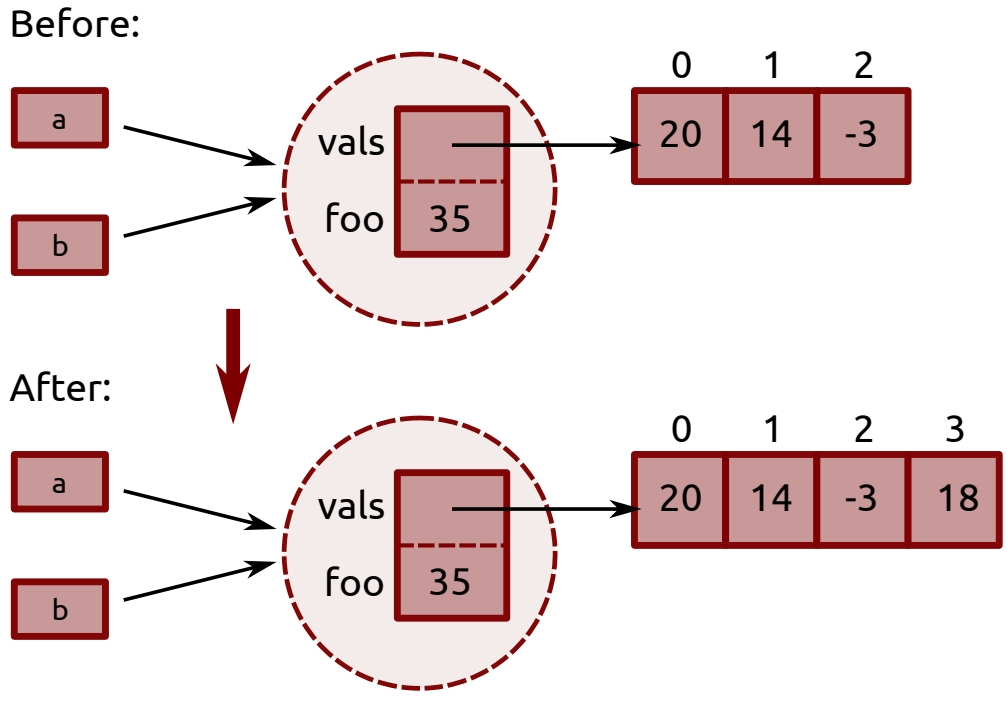
References

More Complicated Example: Class Instances

```
class X:
    def __init__(self):
        self.vals = [20, 14, -3]
        self.foo = 35
```

references_example.py

```
>>> a = X()
>>> b = a
>>> b.vals.append(18)
>>> b.vals
[20, 14, -3, 18]
>>> a.vals
[20, 14, -3, 18]
>>> b is a
True
```



References

Different Example: Class Instances

```
class Friend:
    def __init__(self, name):
        self.name = name
        self.best_friend = None

    def print_best_friend(self):
        if self.best_friend == None:
            print("{} has no best friend.".format(self.name))
        else:
            print("The best friend of {} is {}.".format(
                  self.name, self.best_friend.name))

if __name__ == "__main__":
    f1 = Friend("Ryan")
    f2 = Friend("Jenna")
    # Unreciprocated best friendship.
    f1.best_friend = f2
    f1.print_best_friend()
    f2.print_best_friend()
    # Can add random field and access it through Ryan's reference.
    f2.random_field = -87
    print(f1.best_friend.random_field)
```

friend.py

```
The best friend of Ryan is Jenna.
Jenna has no best friend.
-87
```

References

Making Actual Copies?

- What if we *did* want to make a copy of an instance with its own copies of the instance's members/fields?
- Options:
  - Create a new instance and manually assign each field.
  - Create a method that manually assigns each field.
    - This would be analogous to a "copy constructor" in C++.

References

Making Actual Copies?

Example

```
class X:
    def __init__(self):
        self.vals = [20, 14, -3]
        self.foo = 35
```
references_example.py

```
>>> c = X()
>>> c.vals.extend([14,18])
>>> c.foo = 49
>>> print(c.vals, c.foo)
[20, 14, -3, 14, 18] 49
>>> d = X()
>>> d.vals = c.vals
>>> d.foo = c.foo
>>> print(d.vals, d.foo)
[20, 14, -3, 14, 18] 49
```

- Nothing wrong so far. Let's change the members of `d`.

```
>>> d.foo = 78
>>> d.foo
78
>>> c.foo
49
```

- Still nothing wrong.

```
>>> d.vals.append(-15)
>>> d.vals
[20, 14, -3, 14, 18, -15]
>>> c.vals
[20, 14, -3, 14, 18, -15]
```

- Um...

References

Shallow Copies

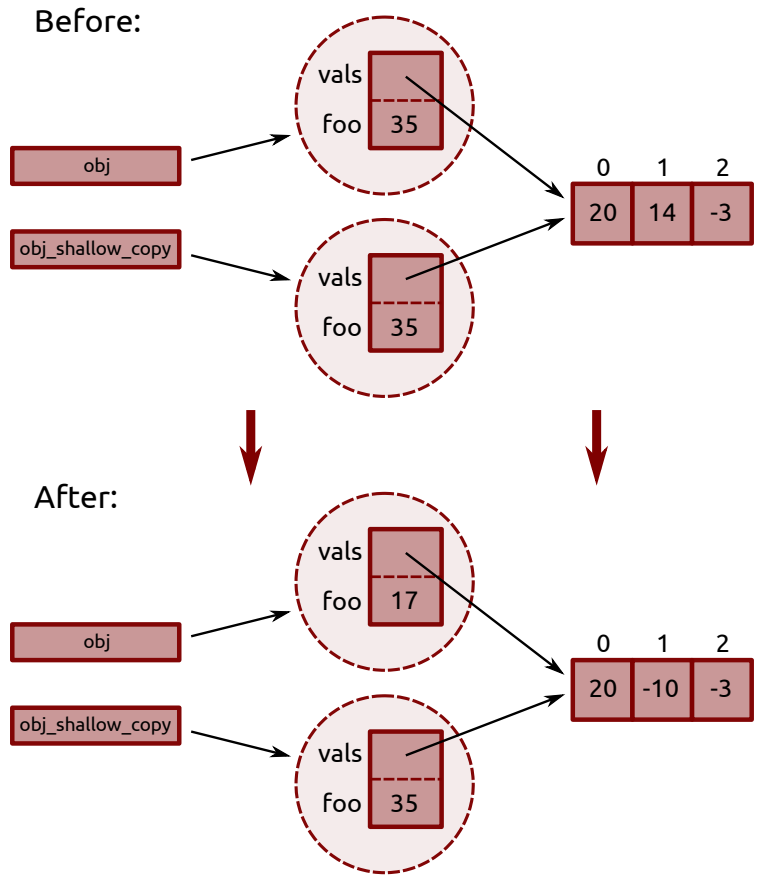
- What we just created was a shallow copy.
- You can do the same thing with `copy()` from the `copy` module.

Example

```
class X:
    def __init__(self):
        self.vals = [20, 14, -3]
        self.foo = 35
```

references_example.py

```
>>> from copy import copy
>>> obj = X()
>>> obj_shallow_copy = copy(obj)
>>> obj.foo = 17
>>> obj.foo
17
>>> obj_shallow_copy.foo
35
>>> obj.vals[1] = -10
>>> obj.vals
[20, -10, -3]
>>> obj_shallow_copy.vals
[20, -10, -3]
```



- `obj` and `obj_shallow_copy` *do not* reference the same class instance, but their own `vals` fields reference the same list.

References

Deep Copies

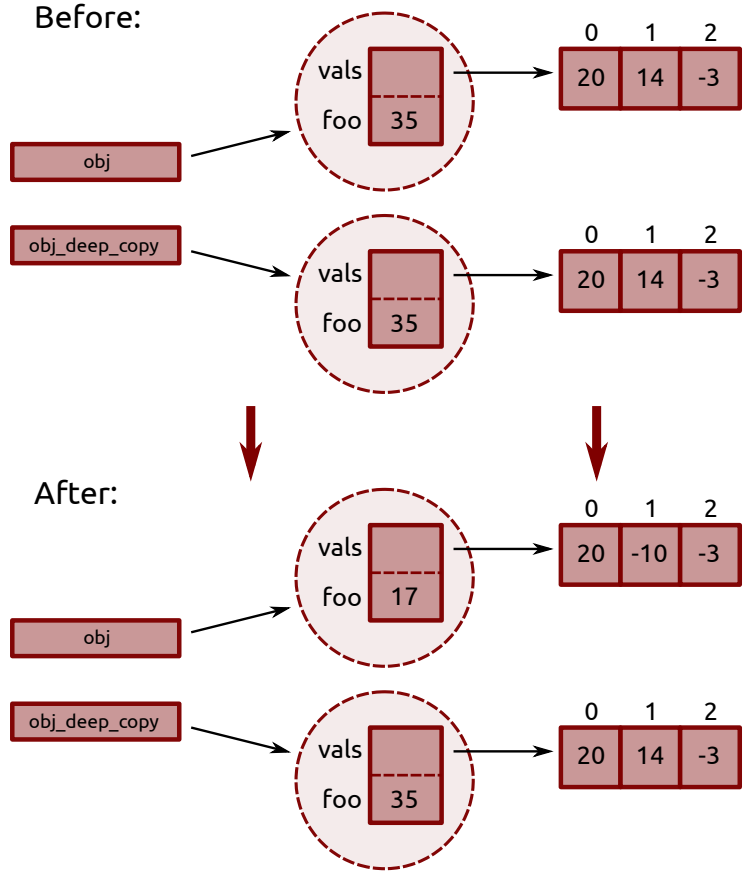
- What we truly want.

Example

```
class X:
    def __init__(self):
        self.vals = [20, 14, -3]
        self.foo = 35
```

references_example.py

```
>>> from copy import deepcopy
>>> obj = X()
>>> obj_deep_copy = deepcopy(obj)
>>> obj.foo = 17
>>> obj.foo
17
>>> obj_deep_copy.foo
35
>>> obj.vals[1] = -10
>>> obj.vals
[20, -10, -3]
>>> obj_deep_copy.vals
[20, 14, -3]
```

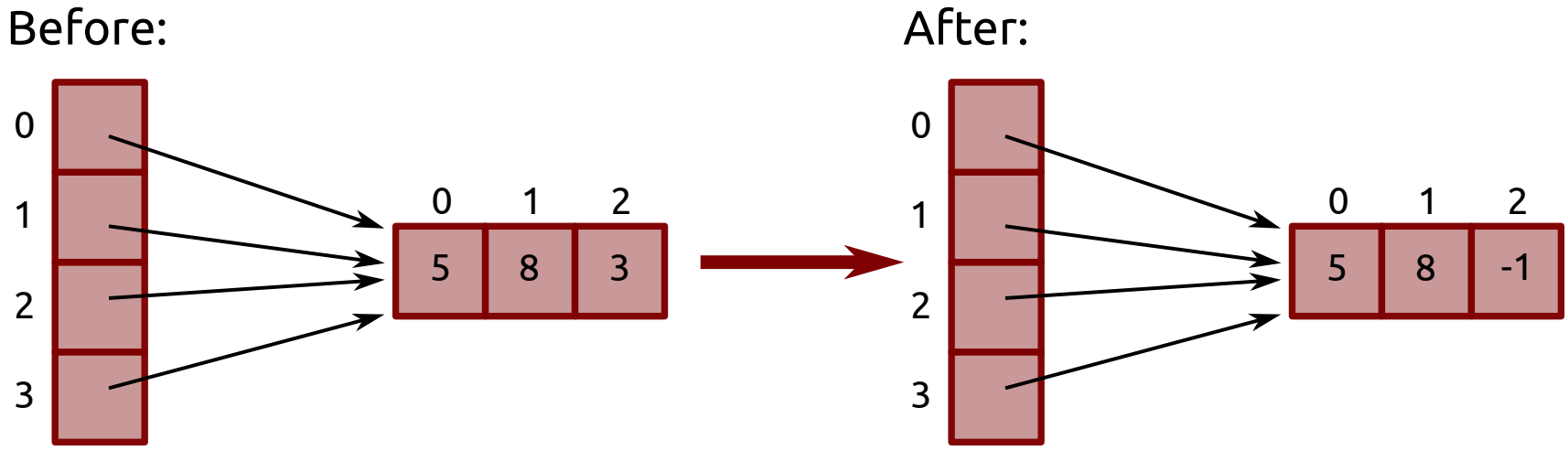


References

Initializing 2D Lists[1]

The Wrong Way

```
>>> vals = [[5, 8, 3]] * 4
>>> vals
[[5, 8, 3], [5, 8, 3], [5, 8, 3], [5, 8, 3]]
>>> vals[1][2] = -1
>>> vals
[[5, 8, -1], [5, 8, -1], [5, 8, -1], [5, 8, -1]]
```



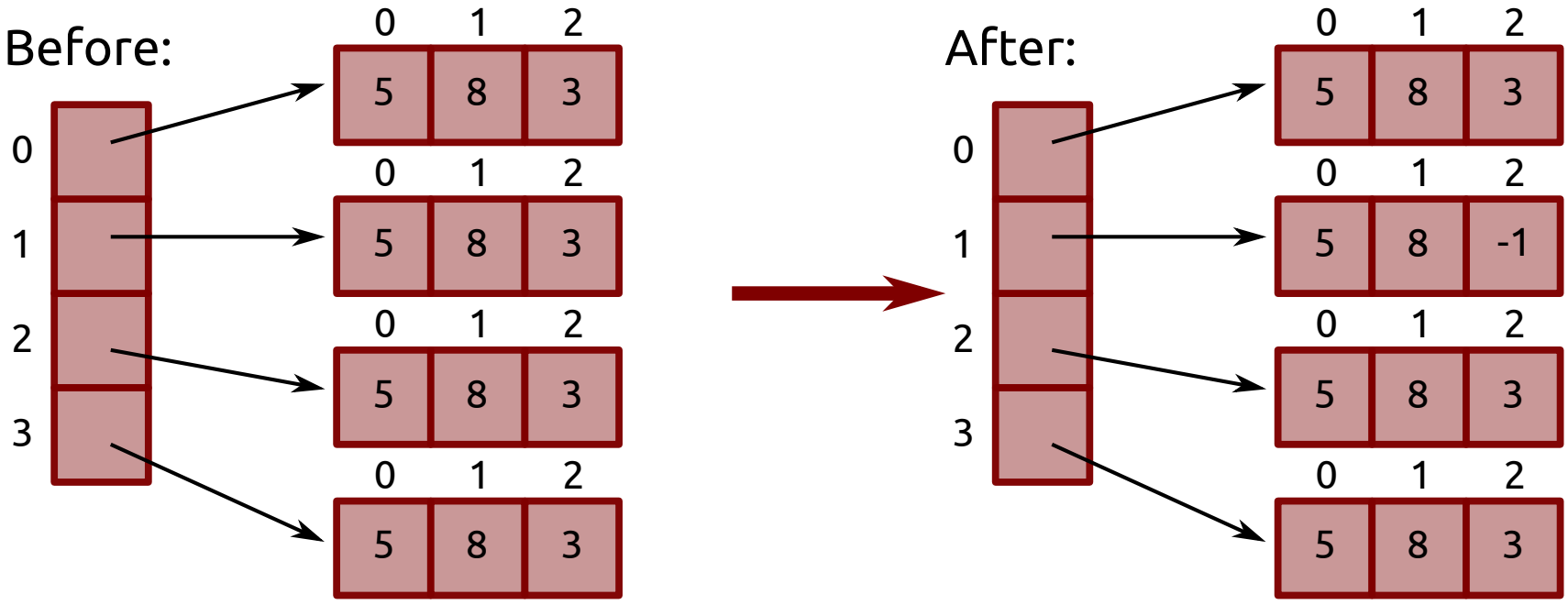
1. Nothing here should affect programming assignment #5, since on the one problem that involves *you* creating a 2D list (rather than your function being given one), it is more intuitive to create the list row-by-row and appending each row.

References

Initializing 2D Lists

Workaround[1]

```
>>> vals = [[5, 8, 3] for i in range(4)]
>>> vals
[[5, 8, 3], [5, 8, 3], [5, 8, 3], [5, 8, 3]]
>>> vals[1][2] = -1
>>> vals
[[5, 8, 3], [5, 8, -1], [5, 8, 3], [5, 8, 3]]
```



1. Source: https://stackoverflow.com/a/44382900/7020654

References

Concluding Remarks

- I've focused on details that make it seem as if references are a heavily complex concept. In reality, you are unlikely to need to think about the complex aspects of references or the differences between shallow and deep copying. However, it is possible that you may encounter a bug involving references[1], and when that happens, having exposure to these aspects of references may help.
  - In ECS 32B, you'll often see code that uses references, and you won't have to think about it; the way in which the references are used will feel intuitive.
  - In ECS 36A, pointers (the hardest concept in the C programming language) will seem similar to references.

1. In fact, the note about 2D lists on the previous slide was a bug that I encountered while coding my own solution for a homework assignment the last time I taught this course.

Other Topics Related to Classes

List of Topics

- Encapsulation.
- Design choices.
- Operating overloading.
- Inheritance.
  - Multiple inheritance.
    - Diamond of death.
  - Interfaces.
- Annotating functions with types and the typing module.

Remarks

- You won't learn about many of these until ECS 32B and ECS 34 (if you are a non-major student) or ECS 36B (if you are a major student).