SETS!

FUNCTIONS: WHAT WE DISCUSSED

TOPIC	COVERED WHERE?	EMPHASIS IN HW	SIGNIFICANCE FOR EXAMS
Signatures	Videos + Lecture	Very High	Very High
Calling Functions	Videos + Lecture	Very High	Very High
The `return` statement	Videos + Lecture	Very High	Very High
Using returned values	Videos + Lecture	Very High	Very High
Type Annotations	Videos + Lecture	Very High	Very High
`main()`	Lecture	Matters a bit	so-so
Keyword Args	Videos	High, later	so-so

SO: PICK AN EXERCISE.

Both work as *exam practice*—representative of some of the shorter questions we might ask to test your understanding of fundamental Python behavior.

1. Understanding how `return` behaves within the body of a function
2. Calling functions & using the returned values.

(Answers to both at the end of this PDF, but definitely do not skip to the end.)

S7 What gets printed?

```
def double(x):
    return x * 2

def square(x):
    return x * x

a = double(3)
b = square(a)
print(a)
print(b)
```

S7 What gets printed?

```
def mystery(n):
    answer = n * 3
    return answer
    answer = answer + 10

result = mystery(4)
print(result)
```

REVIEW: SETS

Sets are an **unordered** container for data.
Unordered means:

- We can still use `for` to loop over every element
- can still use `in` to see if something is in it
- can **not** access into it with an index and `[]` or slice

REVIEW: SETS

Sets look similar to lists, but with `{}` instead of `[]`

- `{"this"}`
- `{"howdy", "partner"}`

Cannot store lists, dicts or other sets within a set

Most important part of a set: it enforces uniquness of its elements. An element can only be in the set once or not at all.

MORE SET REVIEW

There are a few common features of a set:

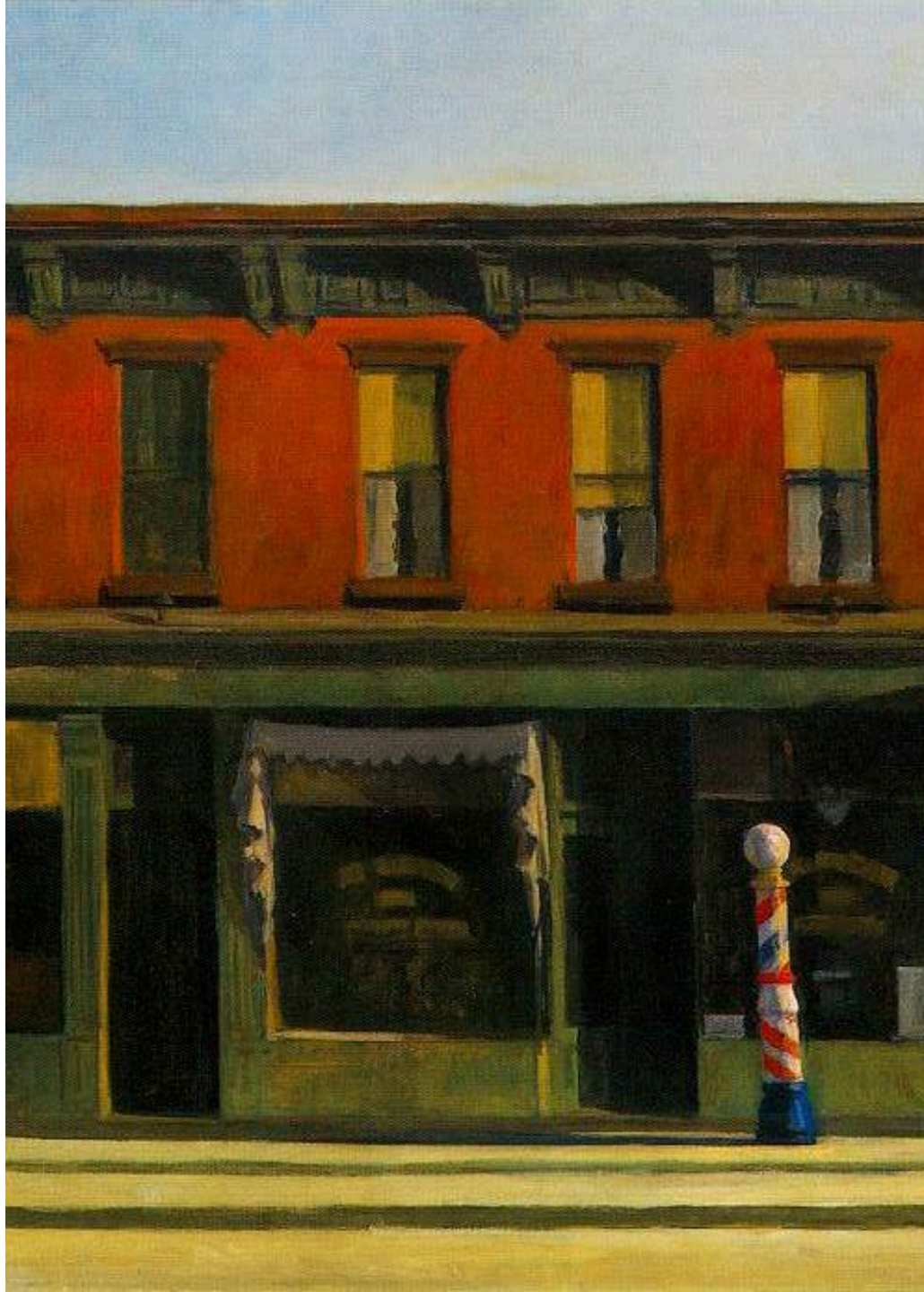
- `.add()` adds an item to the set
- `.remove()` removes an item from a set, if the item is not in the set then crash
- `.discard()` removes an item from a set, ignore if the item is already not in the set
- set comprehension work like list comprehensions, use {} instead of []

ART THIEF!

Someone walked into the gallery, cut the power, and absconded with this priceless piece of 20th century American art while the lights were off!

The museum has a report of everyone who was observed on video *walking in and out* before the power was cut.

Let's use *sets* to determine who might still have been in the gallery at the time the painting was stolen.



THE REPORT

"*Dunne walked in* through door one. *Calvino walked in* through door two shortly after. Several minutes passed before *Cole and Wallace walked in* through door two, passing *Calvino on his way out*. A loud altercation between Cole and Wallace could be heard shortly before *Wallace walked out* of door two. Finally, the lights went off."

C12 Using `.add()` to model a person walking in and `.remove()` to model a person walking out, determine who was in the gallery when the lights went out!

```
suspects = set()
suspects.add("Dunne")
# TODO: model the rest! Then:
print(suspects)
```

```
suspects = set()
suspects.add("Dunne")
suspects.add("Calvino")
suspects.add("Cole")
suspects.add("Wallace")
suspects.remove("Calvino")
suspects.remove("Wallace")
print(suspects)
```

AND SO TWO REMAIN...

All visitors to the gallery were required to share their email address. We'll want to write some code to see if we can match Cole and Dunne to their email addresses.

First, let's just try for Cole. Assume we have `emails`, a set of email addresses as strings.

```
emails = {"rooftoptrias@aol.com", "eb-white-nyc@me.com",
          "tremorscole@gmail.com", "narcolemons@aol.com",
          "whodunneit@me.com","ioawnatcalvino@gmail.com",
          "sharry@seas.upenn.edu", "sneakysuspect@aol.com",
          "dfwallace@gmail.com"}
```

Do any addresses seem like they might belong to Cole? Why?

S8 Fill in the missing line to help us decide when to print out an address because it looks like it might be Cole's!

```
emails = {"rooftoptrias@aol.com", "eb-white-nyc@me.com",
          "tremorscole@gmail.com", "narcolemons@aol.com",
          "whodunneit@me.com","ioawnatcalvino@gmail.com",
          "sharry@seas.upenn.edu", "sneakysuspect@aol.com",
          "dfwallace@gmail.com"}
for addr in emails:
  if ________________:
    print(f"{addr} could be Cole's address.")
```

To do the same for Dunne, we'd need to add the following:

```
emails = {"rooftoptrias@aol.com", "eb-white-nyc@me.com",
          "tremorscole@gmail.com", "narcolemons@aol.com",
          "whodunneit@me.com","ioawnatcalvino@gmail.com",
          "sharry@seas.upenn.edu", "sneakysuspect@aol.com",
          "dfwallace@gmail.com"}
for addr in emails:
  if "Cole".lower() in addr:
    print(f"{addr} could be Cole's address.")
  if "Dunne".lower() in addr:
    print(f"{addr} could be Dunne's address.")
```

But it's usually good to avoid repeating ourselves. What if we had 50 suspects! Write 50 conditionals?!

C14 Write a function `find_emails_for_suspect` that accepts two arguments.

- The first argument is a set of email addresses as strings.
- The second argument is the name of a single suspect.

Iterate through the email addresses, *printing* out any that contain the suspect's name.

Essentially, you should **refactor** the code below so that it is the body of a function `find_emails_for_suspect` that can be called with any set of `emails` and any `name` instead of just `"Cole"`!

```
for addr in emails:
  if "Cole".lower() in addr:
    print(f"{addr} could be Cole's address.")
```

```
def print_emails_for_suspect(emails: set[str], name: str):
  for addr in emails:
    if name.lower() in addr:
      print(f"{addr} could be {name}'s address.")
```

We can build a version that works over a list of names, too.

```
def print_all_matches(emails: set[str], names: list[str]):
  for name in names:
    print_emails_for_suspect(emails, names)
```

Full Version of the email finder.

```
emails = {"rooftoptrias@aol.com", "eb-white-nyc@me.com",
          "tremorscole@gmail.com", "narcolemons@aol.com",
          "whodunneit@me.com","ioawnatcalvino@gmail.com",
          "sharry@seas.upenn.edu", "sneakysuspect@aol.com",
          "dfwallace@gmail.com"}
names = ["Cole", "Dunne"]
def print_emails_for_suspect(emails: set[str], name: str):
  for addr in emails:
    if name.lower() in addr:
      print(f"{addr} could be {name}'s address.")

def print_all_matches(emails: set[str], names: list[str]):
  for name in names:
    print_emails_for_suspect(emails, name)

print_all_matches(emails, names)
```

SET OPERATIONS

NAME	MEANING	METHOD	OPERATOR
Union	Create a new set with all elements from both	`s.union(t)`	`s | t`
Intersection	Create a new set with only elements that appear in both sets	`s.intersection(t)`	`s & t`
Difference	Create a new set with only elements in `s` that don't appear in `t`	`s.difference(t)`	`s - t`
Symmetric Difference	Create a new set with elements that appear in only one set *but not both*	`s.symmetric_difference(t)`	`s ^ t`

The Art Protection Agency maintains a list of "suspicious" Substack accounts. Some of the accounts have public follower lists.

C14 Fill in the function below that takes in a set of accounts *followed by a suspect* and a set of *suspicious* accounts to follow. Return a new set that contains the **intersection** of these sets; that is, the names of all of the accounts that are both *suspicious* and *followed by our suspect.*

```
def find_sus_follows(followed: set[str], suspicious: set[str]) -> set[str]:
  sus_followed = set()
  # TODO!


  return sus_followed
```

```
find_sus_follows({"How2Steal", "Cats!"}, {"EZCrime", "How2Steal", "Bad Stuff"})
```

(for example)

Can we run `find_sus_follows` on these three sets to determine if Cole or Dunne seem to be the more likely culprits?

```
suspicious_accounts = {"EZCrime", "Uh oh, did you get caught?","How2Steal",
                       "Bad Stuff", "Lying: it's good, actually",
                       "I'm a bad guy, basically: my life story.",
                       "Getting Art For Free (real)"}

cole_follows = {"art history monthly", "theater/theatre",
                "appreciating the finer things", "cats monthly",
                 "cats weekly", "cats daily", "silver bulletin"}

dunne_follows = {"how to steal a painting", "dogs: good",
                 "bad food that looks good", "EZCrime",
                  "the politics understander",
                  "here is how to fix the world. i mean it.",
                  "how to come up with fake newsletter names",
                  "Bad Stuff", "Good stuff",
                  "finding your inner self with python"}
```

SET OPERATIONS PRACTICE

(C16), maybe
Implement both a union function without using the built-in intesection or union operators or functions.

```
def set_union(s1, s2):
    # given two sets, return a new set that has all elements of both input sets
    # set_union({"hi","ho"}, {"bad","hi"}) -> {"hi", "ho", "bad"}
```

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.

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", "Talie" : "taliem"}`
  - 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)

(S9)

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

(S10)

```
last_year_movies = {"Anora", "Conclave", "A Real Pain"}
```

NEXT TIME

- More on dictionaries!
  - I only just barely started talking about them
  - These are a really important data structure in Python and Programming in general

```
def double(x):     # called as double(3), so x <- 3
    return x * 2   # returns 3 * 2, which is 6

def square(x):     # called as square(a) when a = 6, so x <- 6
    return x * x   # returns 6 * 6, which is 36

a = double(3)      # a gets the returned value, which was 6
b = square(a)      # b gets the returned value, which was 36. a is UNCHANGED.
print(a)           # a still stores 6, so print 6
print(b)           # b stores 36, so print 36
```

```
def mystery(n):              # called as mystery(4), so n <- 4
    answer = n * 3           # answer = 4 * 3, so 12
    return answer            #  ! ! RETURN with value 12  ! !
    answer = answer + 10     # this line is not reached; prev. line had return

result = mystery(4)          # result gets the returned value, which is 12
print(result)                # result stores 12, so print 12
```