PATTERNS

MON/WED: NINE STEPS TO SOLVE A PROBLEM

1. Understand the Problem
2. Write your Signature and Purpose
3. Identify Examples/Test Cases
4. Solve your Examples by Hand
5. Identify Your Data Types
6. Write your Outline
7. Fill in the Gaps
8. Code!
9. Test

Wow! Tests are for understanding! Functions emerge naturally as organizational problem solving tools!

TODAY: GOING FROM OUTLINE TO CODE

1. Understand the Problem
2. Write your Signature and Purpose
3. Identify Examples/Test Cases
4. Solve your Examples by Hand
5. Identify Your Data Types
6. **Write your Outline**
7. **Fill in the Gaps**
8. **Code!**
9. Test

*You get "good" at coding when you find yourself working harder at the **first** steps rather than the last ones.*

Going from this to code becomes easy when you **spot & apply patterns.**

> *First, I look at the first word in the list*
> *Then, I look at the last letter in that word (let's call it LAST)*
> *I look at the second word in the list*
> *Then, I look at the first letter in that word (let's call it FIRST)*
> *Then I check if LAST is equal to FIRST*
>
> *I look at the next (second) word in the list*
> *Then, I look at the last letter in that word (let's call it LAST)*
> *I look at the third word in the list*
> *Then, I look at the first letter in that word (let's call it FIRST)*
> *Then I check if LAST is equal to FIRST*
>
> *I look at the next (third) word in the list...*

SYNTAX PATTERNS

THE QUOTE-UNQUOTE EASY STUFF

We'll call something a **syntax pattern** when there's a simple one- or few-line solution to a common problem that always works the same way.

- Memorize these in order to get fast at the atomic operations
- Mostly these rely on basic rules about the data types & operations involved
- Using these will usually look the same every time.

Really simple one to start: **"write `something` down"**

- Use when you need to save some information for later
- `saved_for_later = something` — it's just putting `something` in a variable!

SYNTAX PATTERNS: PARTS OF SEQUENCES

WHAT SHOULD HAPPEN?	HOW TO DO IT?	E.G.
Take one element from a sequence	`seq[idx]`	"Take the third thing..."
Take the first `n` elements from a sequence	`seq[:n]`	"For the first five names..."
Skip the first `n` elements from a sequence	`seq[n:]`	"For all classes after the first..."
Take the elements numbered from `i` up to `j`	`seq[i:j]`	"For all years between 2020 and 2026..."

SYNTAX PATTERNS: DATA STRUCTURES

WHAT SHOULD HAPPEN?	HOW TO DO IT?	E.G.
Check if `this` is in `that`	`this in that`	"Is this name in the banned list?"
Remove all duplicates/get unique values from a sequence	`set(seq)`	"How many different people signed up?"
Associate `this` info with `that` that entity	`d[that] = this`	"Player 2's score should be 17"

Also useful: set operations like intersection, union, difference, symmetric difference. But those are maybe not "core" for now.

LET'S TRY IT

Write a function that takes a list of all coffee orders at a cafe in a day and a list of drinks. We want to find for each drink how many times it was ordered.

Together: steps 1-6 (with an **L1** intermission for you to write a test case.)

FINALLY:

```
def count_orders(orders: list[str], to_count: list[str]) -> dict[str,
                                                                 int]:
    counts = dict()
    for drink in to_count:
        count = 0
        for order in orders:
            if order == drink:
                count += 1
        counts[drink] = count
    return counts
```

FUNCTION PATTERNS

PATTERNS FOR WRITING FUNCTIONS

A surprising number of the problems we encounter can be solved with the same few patterns

- These patterns have **repeated structure** with **varying components**
- Basically: put some conditionals and loops in the same order but swap out the conditions/bodies
- Composing different patterns together will get you very, very far...
- It can be helpful to think of solutions as a function pattern with a particular **helper function**, which are short functions we can write to help solve a small part of the problem.

FILTER PATTERN

"Filter" is a pattern for collections of things that creates a new sequence containing only those elements for which the predicate function evaluates to `True`.

```
res = []
for elem in collection:
    if predicate(elem):
        res.append(elem)
```

`predicate(elem)` is a placeholder of any function that returns a boolean.

(Also, `res` could be a `set` or a `str` or a `dict`—the same idea works every time.)

Given a list of words and a particular threshold, find me all the words with lengths longer than that threshold.

```
def find_long_words(words: list[str], min_length: int):
    ...
```

> *Given a list of words and a particular threshold, find me all the words with lengths longer than that threshold.*

"Find all that..." ➡️ that's a filter!

```python
def find_long_words(words: list[str], min_length: int):
    res = []                        # TODO: go from pattern -> solution
    for elem in collection:
        if predicate(elem):
            res.append(elem)
    return res
```

> *Given a list of words and a particular threshold, find me all the words with lengths longer than that threshold.*

"Find all that..." ➡️ that's a filter!

```
def find_long_words(words: list[str], min_length: int):
    res = []
    for word in words:
        if predicate(word):
            res.append(word)
    return res
```

Given a list of words and a particular threshold, find me all the words with lengths longer than that threshold.

```
def find_long_words(words: list[str], min_length: int):
    res = []
    for word in words:
        if is_long(word, min_length):
            res.append(word)
    return res

def is_long(word, min_length):          # Helper function
    return len(word) > min_length
```

WHEN DOES A HELPER HELP?

We could simply replace this...

```
def find_long_words(words: list[str], min_length: int):
    res = []
    for word in words:
        if is_long(word):
            res.append(word)
    return res

def is_long(word, min_length):              # Helper function
    return len(word) > min_length
```

WHEN DOES A HELPER HELP?

...with this.

```
def find_long_words(words: list[str], min_length: int):
    res = []
    for word in words:
        if len(word) > min_length:
            res.append(word)
    return res
```

It works! Both are fine. But the pattern can be a bit simpler to apply when you have a consistent inventory of steps to take: *paste in the pattern; write a helper function that returns a boolean.*

C1 Try these! Apply the "filter" pattern to solve these. Each should look basically the same, but with a different helper function written for each.

> *Given a list of numbers, find all the even ones.*

> *Given a list of grades as tuples of (name, grade), keep only the tuples that correspond to passing grades.*

```
def find_evens(numbers):
  output = []
  for num in numbers:
    if num % 2 == 0:
      output.append(num)
  return output
```

```
def find_passing_grades(grades):
  output = []
  for grade in grades:
    if grade >= 60:
      output.append(grade)
  return output
```

MAP

"Map" is a pattern to take a sequence and produce a new sequence containing elements of the input sequence after having some transformation applied to them.

```
res = []
for elem in seq:
    res.append(transformation(elem))
```

`transformation(elem)` is a placeholder of any function that returns a new value based on the input `elem`.

(Also, `res` could be a `set` or a `str` or a `dict`—the same idea works every time.)

Help! What are some ways that we could curve these scores by mapping?

```
exam_scores = [90, 89, 41, 100, 82, 71, 66]
```

```
def curve_scores(scores: list[int]) -> list[int]:
    new_scores = []
    for score in scores:
        new_scores.append(transformation(score))
    return new_scores

def transformation(score: int) -> int:
    ...
```

PRACTICE! WHAT DO EACH OF THESE EVALUATE TO?

C2 If `names = ["travis", "Harry", "talie", "adi", "Sukya", "MollY"]`, what's the result? Moreover, which patterns (filter, map) are being used in each case?

```
def perfect_names(names: list[str]) -> list[str]:
    res = []
    for name in names:
        if is_perfect(name):
            res.append(name)
    return res

def is_perfect(name: str) -> bool:
    return len(name) == 5 and 'a' in name
result = perfect_name(names)
```

```
def enlouden(names: list[str]) -> list[str]:
    res = []
    for name in names:
        if is_quiet_perfect(name):
            res.append(name.upper())
    return res

def is_quiet_perfect(name: str) -> bool:
    return is_perfect(name) and name.islower()
```

PRACTICE! IDENTIFYING THE PATTERN

L1

Many problems can boil down to a combination of filter/map
For these problems what functions do you think would be needed?
Select all that apply for each question, A for filter, B for map

- get a list of all the capital letters in a string
- given a list of numbers, get another list of numbers that has the squares of those numbers
- given a list of numbers, get another list of numbers that has the squares of those numbers that are less than 25

TOGETHER, APPLYING PATTERNS

Using map/filter patterns: Given a list of numbers, print another list of numbers that has the squares of those numbers that are less than 25. (the squares, not the original numbers) *Use the helper functions defined here.*

```
def small_enough(n):
  return n < 25
def square(n):
  return n * n
numbers = [3, -6, 8, 4, 4.1]
# TODO
print(output)
```

CAESAR CIPHERS

WHAT IS ENCRYPTION?

Caesar needs to communicate with his general who is out in the countryside. He tries to send a message, but he's worried about the messenger getting captured.

- Before leaving, the general and Caesar agree on a secret **key**
- When sending out the message, Caesar uses the key to transform the message so that it cannot be understood, **encrypting** it.
- When receiving the message, the general uses the key to undo the encryption of the message.
- If someone else sees the encrypted message, they are (theoretically) unable to interpret it without knowing which secret key was used.

✨*It's showtime, baby*✨

EVERYBODY DO THE CAESAR SHIFT

A Caesar Cipher works by picking a **key**, which will be a **single character.**
By matching each letter to a number...

LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	...	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	...	23	24	25

...we can do "arithmetic" on letters, e.g.

- B + C $\rightarrow$ 1 + 2 $\rightarrow$ 3 $\rightarrow$ D
- A + B $\rightarrow$ 0 + 1 $\rightarrow$ 1 $\rightarrow$ B

A caesar takes a message and shifts (adds) all letters in the message by the key:
`CAFE` shifted by `B` is `DBGF`. *(Why?)*

EVERYBODY DO THE CAESAR SHIFT

LETTER	A	B	C	D	E	F	G	H	I	J	K	L	M	...	X	Y	Z
Number	0	1	2	3	4	5	6	7	8	9	10	11	12	...	23	24	25

L2 If we encrypt `HI` using `D` as a key, what is the result? If `GEOI` was encrypted using `E` as a key, what was the original message?

ASCII

An important part of `caesar.py` is that we can convert a character to a integer using the `ord()` function.

`ord('A')`	returns	`65`
`ord('B')`	returns	`66`
`ord('C')`	returns	`67` etc.

We can convert an integer back to a character with `chr()`:

`chr(65)`	returns	`A`
`chr(66)`	returns	`B`
`chr(67)`	returns	`C` etc.

If we want to convert A to 0, B to 1, C to 2 etc. how can we do that?

If we want to convert 0 to A, 1 to B and 2 to C etc. how can we do that?

ORD PRACTICE C3

We have a list that contains some sequence of the letters `'A'`, `'B'`, `'C'` and `'D'`.
Write the function `get_counts()` that returns a list of four integers, containing the number of times `'A'`, `'B'`, `'C'`, and `'D'` show up in the input respectively.

```
letters = ['A', 'C', 'D', 'C', 'A', 'A', 'A', 'C']
counts = get_counts(letters)
print(counts) # prints [4, 0, 3, 1]

def get_counts(letters: list[str]) -> list[int]:
    ...
```

IF TIME (THERE WON'T BE TIME): AGGREGRATION

Aggregation is a pattern that uses an *aggregation function* and a sequence and repeatedly accumulates elements from the sequence using the accumulator function.

```
result = initial_value(seq)
for elem in seq:
    result = aggregate(result, elem)
```

Our aggregated result is not always a list like in filtering/mapping, so we need to pick an initial value that makes sense for our problem.

INITIAL VALUES

PROBLEM	INITIAL VALUE
Adding up everything in a collection	`0`
Multiplying everything in a collection	`1`
Finding the biggest thing in a collection	some arbitrary value from the collection, or something smaller than everything
Finding the smallest thing in a collection	some arbitrary value from the collection, or something bigger than everything

PRACTICE AGGREGATION: AVERAGE

```
def average(scores: list[float]) -> float:
    result = initial_value()
    for score in scores:
        result = aggregate(result, score)
    return result / len(scores)

def initial_value():
    return ______

def aggregate(result: float, score: float) -> float:
    return ____________________________________
```