PHASE 2 · DATA AND FUNCTIONS

Day 7 Tuples, sets and unpacking

Self-Study Guide · About 80 minutes

By the end you will be able to

- Make tuples, including one-element tuples, and explain why a tuple cannot be changed.
- Unpack a tuple or list into names, swap two variables in one line, and collect leftovers with a starred name.
- Use a set to remove duplicates and test membership quickly, and combine sets with union, intersection, difference and symmetric difference.
- Pick a list, a tuple or a set for a job and justify the choice.

What to remember from earlier

- Square brackets read one item by index, counting from 0, and a slice [start:stop:step] gives a new piece that stops before stop. (Day 6: Lists, 6.2 Indexing and Slicing Lists)
- A string is immutable, so you never change it in place; you build a new one and assign it to the name. (Day 3: Strings, 3.5 Strings Are Immutable)
- text.split() breaks a string into a list of words, and separator.join(pieces) glues pieces back together. (Day 3: Strings, 3.7 Splitting and Joining)
- enumerate() gives each item together with its index, and the for loop takes them with two names separated by a comma. (Day 5: Loops, 5.11 enumerate(): Index and Value Together)
- An empty string or an empty list is falsy, so an if on an empty list skips its block. (Day 4: Making decisions, 4.7 Truthy and Falsy Values)
- A list keeps items in order and can change, with append(), pop(), remove() and the other list methods. (Day 6: Lists)
- len(), min(), max() and sum() work on a list, and sorted() gives back a new sorted list. (Day 6: Lists)

Make a `day07` folder for today's files, open it in your editor, and run each script from its terminal, as on the earlier days.

On Day 6 you met the **list**, which holds many values in order and can change. Some groups of values should never change, such as the year, month and day of a birthday. And some jobs care only about **which** values are present, not their order or how many times each appears, such as "which students have handed in their homework?" Today you meet two more collections built for those jobs, the **tuple** and the **set**, and a short way to take a group of values apart into separate names, called **unpacking**.

7.1 Tuples: Groups That Do Not Change

Think about a point on a map, written as (3, 4), or a date, written as 14 July 2009. Each is one thing made of a few parts, and the parts belong together in a fixed order. You could store a date in a list, `[2009, 7, 14]`, but a list invites change: any line of the program could `append` a fourth number or `pop` the month, and then it is no longer a date. You want a group that is built once and then stays exactly as it was.

A **tuple** is an ordered sequence of values that cannot be changed after it is made. You write one by putting values between **parentheses**, separated by commas:

```
point = (3, 4)
birthday = (2009, 7, 14)
```

- The parentheses `(` and `)` hold the tuple's items. (A list uses square brackets; a tuple uses round ones.)
- Commas separate the items, exactly as in a list.
- The items can be any values, and of different types: `("Ana", 15, True)` is a fine tuple.

Most of what you know about strings (Day 3) and lists (Day 6) works the same way on a tuple, because all three are **sequences**: values in a fixed order, each at an index that starts from 0.

```
>>> point = (3, 4)
>>> point
(3, 4)
>>> type(point)
<class 'tuple'>
>>> point[0]
3
>>> point[1]
4
>>> len(point)
2
>>> 4 in point
True
```

You write a tuple with round brackets but **index** it with square ones, as for strings and lists. `len` counts the items, and `in` asks whether a value is one of them.

WORKED EXAMPLE **Reading a date**

Each line reads one part of a birthday stored as a tuple of year, month and day:

birthday.py

```
birthday = (2009, 7, 14)
print("Year:", birthday[0])
print("Month:", birthday[1])
print("Day:", birthday[2])
print("Last item:", birthday[-1])
print("Parts:", len(birthday))
```

```
Year: 2009
Month: 7
Day: 14
Last item: 14
Parts: 3
```

Negative indexes count from the end, as they did for strings on Day 3, so `birthday[-1]` and `birthday[2]` are the same item. The tuple has three items, so the valid indexes are 0, 1 and 2 (or -1, -2 and -3).

Now vary it: put different types in one tuple, and take a **slice**. A tuple stores information about a song: its title, composer, year and length in minutes.

```
>>> song = ("Clair de Lune", "Debussy", 1905, 5.2)
>>> song[-1]
5.2
>>> song[1:3]
('Debussy', 1905)
>>> "Debussy" in song
True
>>> tuple("abc")
('a', 'b', 'c')
>>> tuple([90, 85, 70])
(90, 85, 70)
```

- A slice of a tuple, such as `song[1:3]`, is a **new tuple**, just as a slice of a string is a new string. It starts at index 1 and stops before index 3.
- `in` checks every item, whatever its type.
- **`tuple()`** converts another sequence into a tuple, the way `int()` and `str()` convert on Day 2. `tuple("abc")` makes one item per character, and `tuple([90, 85, 70])` turns a list into a tuple with the same items in the same order. `list()` (Day 6) does the reverse: `list((3, 4))` gives the list `[3, 4]`.

Why a tuple cannot change

A tuple is **immutable**, exactly like a string (Day 3). Once it exists, you cannot replace, add or remove any of its items. Try to move a point by assigning to one position:

move_point.py

```
1 point = (3, 4)
2 point[0] = 5
3 print(point)
```

```
Traceback (most recent call last):
  File ".../day07/move_point.py", line 2, in <module>
    point[0] = 5
    ~~~~~^^^
TypeError: 'tuple' object does not support item assignment
```

It is the same error a string gives on Day 3, with `tuple` in place of `str`. And a tuple has no `append`, `insert`, `pop` or `remove`, because each of those would change it:

```
>>> point = (3, 4)
>>> point.append(5)
Traceback (most recent call last):
  ...
AttributeError: 'tuple' object has no attribute 'append'
```

To get a different point, do what you did with strings: **build a new tuple** and point the name at it.

move_point_fixed.py

```
1 point = (3, 4)
2 point = (5, point[1])
3 print(point)
```

```
(5, 4)
```

Line 2 reads `point[1]` from the old tuple, builds the new tuple `(5, 4)`, and rebinds `point` to it. The old `(3, 4)` was never changed; the name now refers to a different tuple.

Why want a collection that refuses to change? It **protects the data**: a date cannot grow a fourth part by accident, because Python stops the program instead. It **tells the reader** the size and contents are fixed, so nobody has to search the program for an `append`. And Python can **rely on it**, so a tuple can go in places a list cannot, such as inside a set (7.7).

KEY IDEA

A tuple is an ordered, immutable sequence written in parentheses: `(3, 4)`. Index it, slice it, and use `len` and `in` as with a list, but to "change" it, build a new tuple.

PAUSE AND TRY

Question 1 Tracing

A color is stored as a tuple of red, green and blue amounts. What does this program print?

rgb.py

```
orange = (255, 165, 0)
print(orange[1:])
print(orange[0] + orange[2])
print(165 in orange, "165" in orange)
print(len(orange[:1]))
```

Question 2 Debugging

This program should replace Sunday with Monday. It stops with an error. Quote the last line of the error, explain it, and write a fix that keeps `weekend` a tuple.

weekend.py

```
weekend = ("Saturday", "Sunday")
weekend[1] = "Monday"
print(weekend)
```

7.2 The Comma Makes the Tuple

A tuple usually has two or more items. But sometimes you need a tuple with exactly one: a list of sizes that happens to have only one size today, or one item to add on the end of an existing tuple. The obvious way to write it does not work, and the reason teaches you what really makes a tuple.

On Day 2, parentheses had a job already: **grouping**, as in `(2 + 3) * 4`. Python cannot let `(5)` mean "a tuple holding 5", because then `(2 + 3) * 4` would be a tuple times 4. So parentheses around one value are just grouping, and `(5)` is the number 5.

What makes a tuple is the **comma**. The parentheses are usually there for clarity (and are needed in some places, such as inside a function call), but the commas are what Python looks for.

```
>>> just_one = ("small")
>>> type(just_one)
<class 'str'>
>>> just_one = ("small",)
>>> type(just_one)
<class 'tuple'>
>>> len(just_one)
1
>>> just_one
('small',)
>>> empty = ()
>>> len(empty)
0
>>> pair = 3, 4
>>> pair
(3, 4)
>>> type(pair)
<class 'tuple'>
```

Read the session line by line:

- `("small")` is only the string `"small"` in grouping parentheses, so `type` says `str`.
- `("small",)` has a **trailing comma** after the one item, and that comma makes it a one-element tuple. Python prints it back with the comma, `('small',)`.
- `()`, empty parentheses with nothing inside, is the **empty tuple**, with length 0. It is the one tuple written with no comma.
- `pair = 3, 4` has no parentheses at all, and it is still a tuple. Building a tuple from comma-separated values like this is called **packing** the values into a tuple. You will use it in 7.3 and 7.4.

> **WORKED EXAMPLE** **Joining and repeating tuples**
>
> Tuples join with `+` and repeat with `*`, just like strings on Day 3. Both give a **new** tuple; neither changes the old one.

```
>>> (5) * 3
15
>>> (5,) * 3
(5, 5, 5)
>>> (90, 85) + (70,)
(90, 85, 70)
>>> ("a", "b") * 2
('a', 'b', 'a', 'b')
```

- `(5) * 3` is `5 * 3`, the number 15, because `(5)` is not a tuple.
- `(5,) * 3` repeats a one-element tuple three times: `(5, 5, 5)`.
- `(90, 85) + (70,)` joins two tuples into a new one with three items. This is how you "add" to a tuple: make a new one that has the extra item, then assign it to the name.
- `("a", "b") * 2` repeats both items, in order, twice.

KEY IDEA

The comma makes a tuple, not the parentheses. A one-element tuple needs a trailing comma, `(70,)`; without it, `(70)` is just 70.

WATCH OUT

Forgetting the trailing comma. This program tries to add a third score to a tuple:

add_score.py

```
1 scores = (90, 85)
2 scores = scores + (70)
3 print(scores)
```

```
Traceback (most recent call last):
  File ".../day07/add_score.py", line 2, in <module>
    scores = scores + (70)
             ~~~~~~~^~~~~
TypeError: can only concatenate tuple (not "int") to tuple
```

`(70)` is the int 70, and `+` cannot join a tuple and an int, the same way `"Age: " + 15` fails on Day 3. **Concatenate** means join end to end. Add the comma so the right side is a one-element tuple:

add_score_fixed.py

```
1 scores = (90, 85)
2 scores = scores + (70,)
3 print(scores)
```

```
(90, 85, 70)
```

PAUSE AND TRY

Question 3 Tracing

Predict exactly what the REPL prints for the last four lines.

```
>>> a = (7)
>>> b = (7,)
>>> c = 7, 8
>>> type(a)
>>> type(b)
>>> len(c)
>>> b * 2
```

7.3 Unpacking: Many Names at Once

Look back at birthday.py in 7.1. To print the date as 7/14/2009, you would write `birthday[1]`, `birthday[2]` and `birthday[0]`, and the reader has to remember which index is the month. Names such as `year`, `month` and `day` would be much clearer. You could write three assignment lines, one per index, but Python has a shorter way that does all three at once.

Unpacking assigns the items of a sequence to several names in one statement. Put the names on the left of `=`, separated by commas, and the sequence on the right:

```
year, month, day = birthday
```

- Python takes the items of `birthday` in order and assigns the first to `year`, the second to `month`, and the third to `day`.
- The number of names on the left must equal the number of items on the right. (If it does not, you get an error; see the Watch Out below.)
- Unpacking is the opposite of packing (7.2): packing puts several values into one tuple, and unpacking takes them out into separate names.

WORKED EXAMPLE **Unpacking a date**

unpack_birthday.py

```
birthday = (2009, 7, 14)
year, month, day = birthday
print(f"{month}/{day}/{year}")
```

```
7/14/2009
```

Line 2 does the work of three lines (`year = birthday[0]`, `month = birthday[1]`, `day = birthday[2]`). Line 3 then uses the names in an f-string (Day 3), in whatever order the date format needs. Nothing in the tuple changes: unpacking only reads the items.

Unpacking is not only for tuples. Any sequence with the right number of items can be unpacked: a list, a string, or the list that `split()` gives back. And when the right side is several values separated by commas, Python **packs** them into a tuple first and then unpacks it, so one line can set several variables:

```
>>> width, height = 1920, 1080
>>> width
1920
>>> height
1080
>>> first, last = ["Ada", "Lovelace"]
>>> last
'Lovelace'
>>> first, last = "Grace Hopper".split()
>>> first
'Grace'
>>> a, b, c = "cat"
>>> b
'a'
```

- `width, height = 1920, 1080` packs `1920, 1080` into a tuple, then unpacks it into two names.
- `first, last = ["Ada", "Lovelace"]` unpacks a list.
- `first, last = "Grace Hopper".split()` unpacks the two-word list that `split()` gives back (Day 3). This is a handy way to take a full name apart, as long as it has exactly two words.
- `a, b, c = "cat"` unpacks a string one character per name.

KEY IDEA

`name1, name2 = sequence` assigns the items, in order, to the names. The number of names must match the number of items exactly.

WATCH OUT

The counts do not match. Unpacking a three-item tuple into two names:

unpack_too_many.py

```
1 point = (3, 4, 5)
2 x, y = point
3 print(x, y)
```

```
Traceback (most recent call last):
  File ".../day07/unpack_too_many.py", line 2, in <module>
    x, y = point
    ^^^^
ValueError: too many values to unpack (expected 2, got 3)
```

"Too many values" means the right side had more items than the left side had names. The opposite mistake gives "not enough values":

```
>>> year, month, day = (2009, 7)
Traceback (most recent call last):
  ...
ValueError: not enough values to unpack (expected 3, got 2)
```

Python refuses to guess which item you meant to drop or which name should stay empty. Count the items and the names, and make them match. (Older Python versions leave off the `got 3` part of the message.) 7.5 shows a way to unpack when you do not know the count in advance.

Unpacking in a for loop

Day 5 promised that today would explain the two-name form of a for loop from 5.11:

```
1 for index, letter in enumerate(word):
```

Now you can. On each pass, `enumerate` gives the for loop **one tuple**, holding the index and the item. `list()` shows every tuple it would give:

```
>>> list(enumerate("hi"))
[(0, 'h'), (1, 'i')]
>>> index, letter = (0, "h")
>>> letter
'h'
```

The two names after `for` unpack that tuple, exactly like `index, letter = (0, "h")`. So `for index, letter in ...` is a for loop whose loop variable is unpacked on every pass.

The same works for any list whose items are tuples. A **list of tuples** is a common way to store a table: each tuple is a row, and each position in the tuple is a column.

WORKED EXAMPLE **A table of scores**

Each tuple holds a student's name and score. The loop unpacks each row into `name` and `score`:

team_scores.py

```
1 results = [("Ana", 92), ("Ben", 78), ("Chloe", 85)]
2 total = 0
3 for name, score in results:
4     print(f"{name}: {score}")
5     total += score
6 print("Average:", total / len(results))
```

```
Ana: 92
Ben: 78
Chloe: 85
Average: 85.0
```

Pass	Tuple	`name`	`score`	`total` after
1	`("Ana", 92)`	`"Ana"`	92	92
2	`("Ben", 78)`	`"Ben"`	78	170
3	`("Chloe", 85)`	`"Chloe"`	85	255

`total` is an accumulator (Day 5): set to 0 before the loop, added to inside it, used after it. `len(results)` is 3, the number of tuples in the list (not 6: each tuple is one item of the list). 255 / 3 is 85.0.

PAUSE AND TRY

Question 4 Tracing

Each tuple is a room's name, width and length. What does this program print? Trace `area`, `biggest` and `biggest_area` on every pass.

rectangles.py

```
rooms = [("hall", 2, 5), ("kitchen", 3, 4), ("bath", 2, 2)]
biggest = ""
biggest_area = 0
for name, width, length in rooms:
    area = width * length
    if area > biggest_area:
        biggest = name
        biggest_area = area
print(biggest, biggest_area)
```

Question 5 Multiple choice

What happens when this line runs?

```
name, age = ("Ana", 15, "Leeds")
```

A. `name` is `"Ana"`, `age` is `15`, and `"Leeds"` is dropped.
B. `name` is `"Ana"` and `age` is `(15, "Leeds")`.
C. Python stops with `ValueError: too many values to unpack`.

> D. `name` and `age` both become the whole tuple.

7.4 Swapping Two Values

Sometimes two variables hold values in the wrong order. A program asks for a low and a high number, and the user types them the other way round. Or a game changes whose turn it is. You need to **swap**: make each variable hold the other's value.

The first idea most people try is two assignments:

swap_wrong.py

```
1 left = "cat"
2 right = "dog"
3 left = right
4 right = left
5 print(left, right)
```

```
dog dog
```

Trace it. Line 3 copies `"dog"` into `left`, and the `"cat"` that `left` held is gone: no name refers to it any more. Line 4 then copies `left`, which is now `"dog"`, into `right`. Both names end up with `"dog"`. The problem is that the first assignment destroys a value the second one needs.

The classic fix uses a third, **temporary** variable to hold one value while the other moves, like putting one cup down on the table so you can pour between two hands:

swap_temp.py

```
1 left = "cat"
2 right = "dog"
3 temp = left
4 left = right
5 right = temp
6 print(left, right)
```

```
dog cat
```

Line 3 saves `"cat"` in `temp` before line 4 overwrites `left`, so line 5 can still give it to `right`. It works, but it takes three lines and an extra name. Python can do it in one line by combining packing and unpacking:

swap_tuple.py

```
1 left = "cat"
2 right = "dog"
3 left, right = right, left
4 print(left, right)
```

```
dog cat
```

WORKED EXAMPLE **How a, b = b, a works**

Python always works out the **whole right side first**, before assigning anything. That is the same rule as `count = count - 1` on Day 2, applied to several values at once.

Step	What Python does	`left`	`right`
1	Before line 3	`"cat"`	`"dog"`
2	Read `right, left` and pack: `("dog", "cat")`	`"cat"`	`"dog"`
3	Unpack: first item to `left`	`"dog"`	`"dog"`
4	Unpack: second item to `right`	`"dog"`	`"cat"`

At step 3, `left` changes, but the tuple built in step 2 already holds a copy of the old `"cat"`. That packed tuple plays the role of `temp`, without your having to name it.

The same rule makes any "update several variables from their old values" line work. The **Fibonacci sequence** starts 0, 1, and each number after that is the sum of the two before it. Two variables, `a` and `b`, hold the last two numbers; each pass moves them one step along:

fibonacci.py

```
1 a, b = 0, 1
2 numbers = []
3 for step in range(8):
4     numbers.append(a)
5     a, b = b, a + b
6 print(numbers)
```

```
[0, 1, 1, 2, 3, 5, 8, 13]
```

On line 5, the right side `b, a + b` is worked out with the **old** `a` and `b` before either changes. On the first pass that is `1, 0 + 1`, so `a` becomes 1 and `b` becomes 1. On the next, `1, 1 + 1`: `a` becomes 1 and `b` becomes 2. The loop variable `step` is not used in the body; it only makes the loop run 8 times. Line 4 adds each number to the list with `append` (Day 6).

KEY IDEA

`a, b = b, a` swaps two values in one line. Python builds the whole right side from the old values first, then assigns, so no value is lost.

WATCH OUT

Splitting a "together" update into two lines. Write the Fibonacci update as two separate assignments and the sequence goes wrong:

fibonacci_wrong.py

```
1 a = 0
2 b = 1
3 numbers = []
4 for step in range(8):
5     numbers.append(a)
6     a = b
7     b = a + b
8 print(numbers)
```

```
[0, 1, 2, 4, 8, 16, 32, 64]
```

Line 6 changes `a` first, so line 7 adds the **new** `a` to `b` instead of the old one: `b` simply doubles. It is the same bug as swap_wrong.py. When every new value must come from the old values, put them in one assignment: `a, b = b, a + b`.

PAUSE AND TRY

Question 6 Tracing

What does this program print?

rotate.py

```
first, second, third = "red", "green", "blue"
first, second, third = second, third, first
print(first, second, third)
x, y = 2, 5
x, y = y, x * y
print(x, y)
```

7.5 Starred Unpacking: Collecting the Rest

Plain unpacking needs exactly one name per item. But often you want only the first item or two and do not care how many follow. A race has a winner and a runner-up, and then everyone else. A list of scores has a top score, and then the rest. You cannot write the right number of names when the length changes from day to day.

Starred unpacking puts a star, `*`, in front of one name on the left. That **starred name** collects every item the other names do not take, and it always holds them as a **list**.

```
>>> scores = [90, 85, 70, 60]
>>> first, *rest = scores
>>> first
90
>>> rest
[85, 70, 60]
>>> *others, last = scores
>>> others
[90, 85, 70]
>>> last
60
>>> top, *middle, bottom = scores
>>> middle
[85, 70]
```

- `first, *rest = scores`: `first` takes item 0, and `*rest` collects all the others, in order: `[85, 70, 60]`.
- The star can go at the end, the start or the middle. In `*others, last = scores`, `last` takes the final item and `others` gets everything before it.
- In `top, *middle, bottom = scores`, the plain names take the first and last items, and `middle` collects whatever is between.

The plain names are filled first, from both ends; the starred name gets whatever is left. That is why only **one** name can have a star: with two, Python could not tell how to divide the leftovers between them.

WORKED EXAMPLE **Medals and the rest**

race.py

```
finishers = ["Maya", "Leo", "Sam", "Ivy", "Tom"]
winner, runner_up, *others = finishers
print("Gold:", winner)
print("Silver:", runner_up)
print("Also ran:", len(others), "runners")
```

```
Gold: Maya
Silver: Leo
Also ran: 3 runners
```

`winner` takes item 0, `runner_up` item 1, and `others` the list `["Sam", "Ivy", "Tom"]`. If two more runners finish tomorrow, the same line still works: `others` just gets 5 names. That is what plain unpacking could not do.

Now a second example where the star goes in the middle. In some sports, judges' scores are counted after dropping the highest and the lowest, so one unusual judge cannot swing the result. `sorted()` (Day 6) puts the scores in order, and starred unpacking peels off both ends:

judges.py

```
scores = (7.5, 9.0, 8.0, 9.5, 6.0)
lowest, *middle, highest = sorted(scores)
print("Dropped:", lowest, "and", highest)
print("Counted:", middle)
print(f"Final score: {sum(middle) / len(middle):.2f}")
```

```
Dropped: 6.0 and 9.5
Counted: [7.5, 8.0, 9.0]
Final score: 8.17
```

`sorted(scores)` gives the list `[6.0, 7.5, 8.0, 9.0, 9.5]`. `lowest` takes the first item, `highest` the last, and `middle` the three between. `sum(middle) / len(middle)` is 24.5 / 3, about 8.1667, which `:.2f` shows as `8.17` (Day 3). Notice that `middle` is a list even though `scores` was a tuple: the starred name **always** collects into a list.

The starred name can also collect nothing at all, which is not an error:

```
>>> first, *rest = (5, 6, 7)
>>> rest
[6, 7]
>>> first, *rest = [5]
>>> first
5
>>> rest
[]
>>> first, second, *rest = [5]
Traceback (most recent call last):
  ...
ValueError: not enough values to unpack (expected at least 2, got 1)
```

- From the tuple `(5, 6, 7)`, `rest` is the **list** `[6, 7]`.
- From `[5]`, `first` takes the only item and `rest` is the empty list `[]`.
- The plain names still need an item each. `first, second, *rest = [5]` needs at least 2 items, so Python raises a `ValueError` that says "expected at least 2".

KEY IDEA

In `first, *rest = items`, the plain names take one item each and the one starred name collects all the rest as a list, which may be empty.

WATCH OUT

Two starred names. Trying to split a list into a front half and a back half with two stars:

two_stars.py

```
names = ["Ana", "Ben", "Cy", "Dee"]
*front, *back = names
print(front, back)
```

```
  File ".../day07/two_stars.py", line 2
    *front, *back = names
    ^^^^^^^^^^^^^
SyntaxError: multiple starred expressions in assignment
```

It is a `SyntaxError`, so Python refuses to run any of the file (Day 1). There is no single right way to share four names between two "take the rest" names. Use one star, and make the other name plain:

two_stars_fixed.py

```
1 names = ["Ana", "Ben", "Cy", "Dee"]
2 *front, back = names
3 print(front, back)
```

```
['Ana', 'Ben', 'Cy'] Dee
```

To split a list in half, use slices (Day 6) instead.

PAUSE AND TRY

Question 7 Tracing

What does this program print?

star_string.py

```
1 head, *tail = "hello"
2 print(head)
3 print(tail)
4 print(len(tail))
```

Question 8 Multiple choice

After this line runs, what is `b`?

```
1 a, *b, c = (1, 2)
```

A. `[]`
B. `()`
C. `None`
D. Nothing: Python stops with a `ValueError`.

7.6 Sets: Only Unique Values

A website logs a name every time someone visits: `ana`, `ben`, `ana`, `cy`, `ben`, `ana`. How many **different** people visited? Day 6's practice removed duplicates with a loop that built a new list and checked `in` before each `append`. That works, but "keep one of each" is such a common job that Python has a collection that does it for you.

A **set** is an unordered collection of unique values. It has two rules a list does not:

1. **No duplicates.** A value is either in the set or not; it can never be in it twice. Adding a value that is already there does nothing.
2. **No order.** A set does not remember the order you put things in, and it has no positions. Think of a bag of name tags, not a numbered line of people.

You write a set with **curly braces**, `{` and `}`, and commas between the items. The **`set()`** function converts any sequence into a set, keeping one of each value.

```
>>> sizes = {"S", "M", "L", "M", "S"}
>>> len(sizes)
3
>>> "M" in sizes
True
>>> "XL" in sizes
False
>>> type(sizes)
<class 'set'>
>>> visits = [4, 2, 4, 4, 2, 9]
>>> len(set(visits))
3
>>> sorted(set(visits))
[2, 4, 9]
>>> sorted(set("banana"))
['a', 'b', 'n']
```

- `{"S", "M", "L", "M", "S"}` lists five values, but the set keeps only one of each, so `len` is 3. `in` works as it does for lists.
- `set(visits)` turns a list into a set, dropping the repeats. The list had 6 numbers; the set has 3.
- `sorted()` works on a set and gives back a **list** in order (Day 6), so it is the way to show a set's contents in a predictable order.
- `set("banana")` keeps one of each character: `a`, `b` and `n`.

A set has no order

Because a set has no order, it has no indexes either. Asking for "item 0" makes no sense:

set_index.py

```
1 sizes = {"S", "M", "L"}
2 print(sizes[0])
```

```
Traceback (most recent call last):
  File ".../day07/set_index.py", line 2, in <module>
    print(sizes[0])
          ~~~~~^^^
TypeError: 'set' object is not subscriptable
```

"Subscriptable" means "can be read with square brackets". A set cannot, and it cannot be sliced either. You can still loop over a set with `for`, or test it with `in`.

When Python prints a set, it shows the items in whatever order it happens to store them, which is usually **neither** the order you typed **nor** sorted order:

```
>>> numbers = {10, 3, 8}
>>> numbers
{8, 10, 3}
>>> sorted(numbers)
[3, 8, 10]
```

`{8, 10, 3}` comes from how Python arranges numbers inside a set, a detail you should never rely on. For sets of strings, the printed order can even change from one run to the next. So never depend on a set's order, including the order a `for` loop goes through it. When order matters, use `sorted(some_set)`, which is why today's examples print `sorted(...)` rather than the set itself.

Since order does not count, two sets are equal when they hold the same values, whatever order you wrote them in. Tuples and lists are equal only if the items are in the same order too:

```
>>> {1, 2, 3} == {3, 2, 1}
True
>>> (1, 2, 3) == (3, 2, 1)
False
>>> [1, 2, 3] == [3, 2, 1]
False
```

Changing a set

A set itself is **mutable**: you can add and remove items. (Contrast a tuple, which you cannot change at all.)

```
>>> badges = {"gold"}
>>> badges.add("silver")
>>> badges.add("gold")
>>> len(badges)
2
>>> badges.remove("silver")
>>> sorted(badges)
['gold']
>>> badges.discard("bronze")
>>> badges.remove("bronze")
Traceback (most recent call last):
  ...
KeyError: 'bronze'
```

Method	What it does	If the value is missing
`s.add(value)`	Puts `value` in the set; does nothing if already there	Adds it
`s.remove(value)`	Takes `value` out of the set	Stops with a `KeyError`
`s.discard(value)`	Takes `value` out if it is there	Does nothing, no error

Adding `"gold"` a second time leaves the set unchanged, so its length is 2. `discard("bronze")` quietly does nothing, but `remove("bronze")` stops with `KeyError: 'bronze'`. Use `remove` when the value should be there (so a mistake shows up as an error), and `discard` when "already gone" is fine. A set has no `append`: that name means "add at the end", and a set has no end.

WORKED EXAMPLE **Counting unique visitors**

unique_visitors.py

```
1 visits = ["ana", "ben", "ana", "cy", "ben", "ana"]
2 visitors = set(visits)
3 print("Visits:", len(visits))
4 print("Visitors:", len(visitors))
5 print("Names:", sorted(visitors))
```

```
Visits: 6
Visitors: 3
Names: ['ana', 'ben', 'cy']
```

Line 2 keeps one of each name. Line 3 counts every visit, repeats included; line 4 counts each visitor once. Line 5 uses `sorted` because the set's own order is not reliable. The list still has all 6 names: making a set from it does not change it.

KEY IDEA

A set holds each value at most once and has no order: write it with braces, `{1, 2}`, or convert with `set()`. Use `len`, `in` and `sorted()` on it, but never an index.

WATCH OUT

Writing `{}` for an empty set. Braces with nothing inside do not make a set. They make an empty **dictionary**, a different collection you meet tomorrow on Day 8. To make an empty set, call `set()`:

```
>>> nothing = set()
>>> len(nothing)
0
>>> type(nothing)
<class 'set'>
>>> type({})
<class 'dict'>
```

If you start with `seen = {}` and then call `seen.add(...)`, Python stops with `AttributeError: 'dict' object has no attribute 'add'`.

An empty set, like an empty list on Day 4, is **falsy**, and a set with anything in it is truthy. So `if visitors:` means "if there is at least one visitor". The same goes for tuples: `()` is falsy.

PAUSE AND TRY

Question 9 Tracing

Predict exactly what the REPL prints for the last three lines.

```
>>> letters = set("mississippi")
>>> len(letters)
>>> sorted(letters)
>>> "s" in letters
```

7.7 Fast Membership Tests

Imagine checking names at a party door against a guest list of 500. If the list is in the order people were invited, you read from the top, one name at a time; for someone not invited, that is 500 comparisons. If the names were filed in labelled drawers, A to Z, you would open one drawer and look at a handful.

The `in` operator on a **list** works the first way: Python compares the value with item 0, then item 1, and so on, stopping at a match. A million items can mean a million comparisons.

A **set** works more like the drawers. When a value goes into a set, Python computes a number from the value itself, called its **hash**, and uses that number to decide where to store it. Later, `value in my_set` computes the same hash and looks straight in that one place. It does not have to look at the other items at all. So checking a set takes about the same short time whether the set has 10 items or 10 million. This is called a **membership test**: asking whether a value is a member of a collection.

WORKED EXAMPLE **Checking names at the door**

guest_list.py

```
guests = {"maya", "leo", "sam", "ivy"}
arrivals = ["leo", "tom", "ivy", "zoe"]
for name in arrivals:
    if name in guests:
        print(name, "- welcome!")
    else:
        print(name, "- not on the list")
```

```
leo - welcome!
tom - not on the list
ivy - welcome!
zoe - not on the list
```

- `guests` is a set because the program only ever asks "is this name on it?". Order does not matter, and nobody is invited twice.
- `arrivals` is a list because the order people arrive in does matter, and the loop goes through it in that order. That is why the output order is fixed, even though `guests` is a set.
- Each `name in guests` is a set membership test: one quick lookup, not a scan of all four guests.

With four guests a list would be fast enough. The difference shows when the collection is large or the check runs many times, and choosing a set costs nothing in the code.

Only unchangeable values can go in a set

The hash trick has one cost. A value's hash depends on its contents. If a value could change after it went into the set, its hash would change too, and the set would look in the wrong place for it. So Python allows only values that cannot change in a set: numbers, strings, `True`/`False`, `None`, and tuples (of such values). A value that can be used this way is called **hashable**. Lists and sets can change, so they are not hashable.

WATCH OUT

Putting a list in a set. Store seat positions as `[row, seat]` lists inside a set:

set_of_lists.py

```
seats = {[1, 2], [3, 4]}
print(len(seats))
```

```
Traceback (most recent call last):
  File ".../day07/set_of_lists.py", line 1, in <module>
    seats = {[1, 2], [3, 4]}
            ^^^^^^^^^^^^^^^^
TypeError: cannot use 'list' as a set element (unhashable type: 'list')
```

"Unhashable" means Python cannot compute a hash for it, because a list can change. (Older Python versions print only `TypeError: unhashable type: 'list'`.) A seat position is a fixed pair, which is exactly what a tuple is for:

set_of_tuples.py

```
1 seats = {(1, 2), (3, 4), (1, 2)}
2 print(len(seats))
3 print((3, 4) in seats)
```

```
2
True
```

`(1, 2)` appears twice in the braces but is kept once, so the set has 2 items. This is the third reason for tuples from 7.1: because they never change, they can go in a set.

KEY IDEA

`value in my_set` looks in one place found from the value's hash, so it stays fast however big the set gets. Only hashable (unchangeable) values, such as numbers, strings and tuples, can go in a set.

PAUSE AND TRY

Question 10 Tracing

A game stores which squares have been played as (row, column) tuples. Predict what the REPL prints for the last three lines.

```
>>> moves = {(1, 2), (2, 1), (1, 2)}
>>> len(moves)
>>> (2, 1) in moves
>>> (2, 2) in moves
```

Question 11 Short answer

A spelling checker tests each word of an essay with `word in known_words`, where `known_words` holds 100,000 correctly spelled words. Explain why `known_words` should be a set rather than a list.

7.8 Set Operations

Two school clubs have member lists. Who is in either club? Who is in both? Who is in the chess club but not robotics? You could answer each with loops and `in`, but these are questions about **whole groups**, and sets answer them with one operator each. If you have drawn two overlapping circles to compare two groups (a **Venn diagram**), these are the regions of that drawing.

Operation	Operator	Method	Gives the values that are...
union	`a | b`	`a.union(b)`	in `a`, in `b`, or in both
intersection	`a & b`	`a.intersection(b)`	in both `a` and `b`
difference	`a - b`	`a.difference(b)`	in `a` but not in `b`
symmetric difference	`a ^ b`	`a.symmetric_difference(b)`	in exactly one of them, not both

Each one gives a **new set** and leaves `a` and `b` unchanged. The operators read naturally once you know them: `|` is "or" (in this one or that one), `&` is "and" (in this one and that one), `-` "takes away" the second set's values, and `^` is "one or the other but not both". `|`, `&` and `^` also mean something on whole numbers, which this course does not use; on sets they mean only what the table says.

WORKED EXAMPLE **Comparing two clubs**

clubs.py

```
chess = {"ana", "ben", "cy", "dee"}
robotics = {"cy", "dee", "eli"}
print("Either club:", sorted(chess | robotics))
print("Both clubs: ", sorted(chess & robotics))
print("Chess only: ", sorted(chess - robotics))
print("Robots only:", sorted(robotics - chess))
print("Just one:   ", sorted(chess ^ robotics))
```

```
Either club: ['ana', 'ben', 'cy', 'dee', 'eli']
Both clubs:  ['cy', 'dee']
Chess only:  ['ana', 'ben']
Robots only: ['eli']
Just one:    ['ana', 'ben', 'eli']
```

- **Union** `|` takes every region: all five people, each once, even though cy and dee are in both clubs.
- **Intersection** `&` takes only the overlap: cy and dee.
- **Difference** `chess - robotics` takes chess's members and removes anyone in robotics: ana and ben.
- `robotics - chess` is a **different** answer, eli, because difference depends on which set comes first.
- **Symmetric difference** `^` takes everything except the overlap: ana, ben and eli.

Each result goes through `sorted()` so the names come out in alphabetical order.

Every operator also has a method with a long name, which says what it does in words. The methods give the same results, and they have one extra power: the argument does not have to be a set. Any sequence works, and Python treats it as a set of its items.

```
>>> chess = {"ana", "ben", "cy", "dee"}
>>> robotics = {"cy", "dee", "eli"}
>>> sorted(chess.union(robotics))
['ana', 'ben', 'cy', 'dee', 'eli']
>>> sorted(chess.intersection(robotics))
['cy', 'dee']
>>> sorted(chess.difference(robotics))
['ana', 'ben']
>>> sorted(chess.symmetric_difference(robotics))
['ana', 'ben', 'eli']
>>> sorted(chess.intersection(["ben", "zoe"]))
['ben']
>>> len(chess)
4
```

The last example asks which of `"ben"` and `"zoe"` (given in a list) are in the chess club. The operators are stricter: `chess & ["ben", "zoe"]` would stop with a `TypeError`, because `&` needs a set on both sides. And `len(chess)` is still 4 after all of this: none of the operations changed it.

KEY IDEA

`a | b` is everything in either, `a & b` is what both share, `a - b` is what `a` has that `b` lacks, and `a ^ b` is what only one has. Each makes a new set; order matters only for `-`.

WATCH OUT

Joining sets with `+`. Lists and tuples join with `+`, so it is natural to try it on sets:

set_plus.py

```
1 morning = {"ana", "ben"}
2 evening = {"ben", "cy"}
3 everyone = morning + evening
4 print(sorted(everyone))
```

```
Traceback (most recent call last):
  File ".../day07/set_plus.py", line 3, in <module>
    everyone = morning + evening
               ~~~~~~~~^~~~~~~~~
TypeError: unsupported operand type(s) for +: 'set' and 'set'
```

Sets do not support `+`, because "join end to end" makes no sense for a collection with no ends and no repeats. The set version of "put them together" is union:

set_plus_fixed.py

```
1 morning = {"ana", "ben"}
2 evening = {"ben", "cy"}
3 everyone = morning | evening
4 print(sorted(everyone))
```

```
['ana', 'ben', 'cy']
```

`ben` is in both sets but appears once in the union.

PAUSE AND TRY

Question 12 Tracing

Predict exactly what the REPL prints for each of the last five lines.

```
>>> a = {1, 2, 3, 4}
>>> b = {3, 4, 5}
>>> sorted(a - b)
>>> sorted(b - a)
>>> len(a | b)
>>> sorted(a & b)
>>> sorted(a ^ b)
```

Question 13 Multiple choice

`signed_up` is a set of students who signed up for a trip, and `paid` is a set of students who have paid. Which expression gives the students who signed up but have **not** paid yet?

A. `signed_up & paid`
B. `paid - signed_up`
C. `signed_up - paid`
D. `signed_up ^ paid`

7.9 Choosing a List, a Tuple or a Set

You now have three collections that can each hold several values. They overlap: all three work with `len`, `in` and a for loop. So how do you decide? Ask what the data is and what the program will do with it. Each collection makes a different promise:

Question	List	Tuple	Set
Keeps the order?	Yes	Yes	No
Allows duplicates?	Yes	Yes	No
Can be changed?	Yes	No	Yes (add and remove)
Read by index?	Yes	Yes	No
`in` is fast for big data?	No (checks each item)	No (checks each item)	Yes

Typical job	A sequence that grows or changes	A fixed record of a few parts	A group of unique values

A practical way to choose:

1. **Does each value need to appear only once, and do you mostly ask "is this in it?"** Use a **set**. Examples: the tags on a photo, the words a reader already knows, the squares already played.
2. **Is it one thing made of a fixed number of parts, each with its own meaning by position?** Use a **tuple**. Examples: a date (year, month, day), a map point (x, y), a color (red, green, blue), one row of a table (name, score).
3. **Otherwise, when you need order and the collection will grow, shrink or change**, use a **list**. Examples: a playlist, a queue of tasks, test scores typed in one by one.

The conversion functions let you move between them when a job changes. A common pattern is to remove duplicates with a set, then get an ordered result back:

```
>>> tags = ["red", "blue", "red", "green"]
>>> unique_tags = set(tags)
>>> len(unique_tags)
3
>>> sorted(unique_tags)
['blue', 'green', 'red']
>>> tuple(sorted(unique_tags))
('blue', 'green', 'red')
>>> list((3, 4))
[3, 4]
```

`set(tags)` drops the second `"red"`. `sorted` puts the unique tags in order as a list, and `tuple(...)` freezes that order into a tuple if it should not change from then on. `list((3, 4))` goes from a tuple to a list, which you would do if you needed to change it.

Real programs mix collections. The table in 7.3, `[("Ana", 92), ("Ben", 78), ("Chloe", 85)]`, is a **list of tuples**: each row is a tuple because it is always exactly a name and a score, and the table is a list because rows can be added or removed and are printed in order. If the program also asked "does this student have a score?", a set of names would answer each check quickly.

KEY IDEA

Choose a list for an ordered sequence that changes, a tuple for a fixed record whose parts are known by position, and a set for unique values you mostly test with `in` or compare as groups.

WATCH OUT

Choosing a set when order or repeats matter. A set looks like a tidy list, so it is tempting to use one for anything. But turning test scores `[90, 85, 90]` into a set gives two scores, and the average comes out wrong. Turning a playlist into a set loses the order of the songs. Before you choose a set, check that dropping duplicates and losing order are both acceptable for the job.

PAUSE AND TRY

Question 14 Multiple choice

A program stores the latitude and longitude of a school, which will never change while the program runs. Which is the best choice?

A. A list, `[51.5, -0.12]`
B. A tuple, `(51.5, -0.12)`
C. A set, `{51.5, -0.12}`
D. Two separate lists, one for each number

Question 15 Short answer

For each job, say whether you would use a list, a tuple or a set, and give the reason in one sentence: (a) the email addresses already registered on a website, checked every time someone signs up; (b) the moves in a chess game, in the order they were played; (c) one student's name, year group and homeroom.

Putting it together

Two classes each answered a survey: "What is your favorite fruit?". Each answer is stored as a (name, fruit) tuple, and each class is a list of answers. The program finds the fruits each class picked, which fruits both classes picked, and which only one class picked, then reports the first answer from the morning class:

fruit_survey.py

```
1  morning = [("Ana", "mango"), ("Ben", "apple"), ("Cy", "mango"), ("Dee", "kiwi")]
2  afternoon = [("Eli", "apple"), ("Fay", "pear"), ("Gus", "apple")]
3
4  morning_fruits = set()
5  for name, fruit in morning:
6      morning_fruits.add(fruit)
7
8  afternoon_fruits = set()
9  for name, fruit in afternoon:
10     afternoon_fruits.add(fruit)
11
12 print("Morning:", sorted(morning_fruits))
13 print("Afternoon:", sorted(afternoon_fruits))
14 print("Both classes:", sorted(morning_fruits & afternoon_fruits))
15 print("Only one class:", sorted(morning_fruits ^ afternoon_fruits))
16
17 first, *later = morning
18 first_name, first_fruit = first
19 print(f"First answer: {first_name} chose {first_fruit}, then {len(later)} more")
```

```
Morning: ['apple', 'kiwi', 'mango']
Afternoon: ['apple', 'pear']
Both classes: ['apple']
Only one class: ['kiwi', 'mango', 'pear']
First answer: Ana chose mango, then 3 more
```

Lines	What happens	Idea
1–2	Each answer is a tuple; each class is a list of tuples	7.1, 7.9
4–6	Start an empty set with `set()`; unpack each answer; `add` the fruit	7.3, 7.6
8–10	The same for the afternoon class	7.3, 7.6
12–13	Each class's fruits, once each: "mango" was picked twice but appears once	7.6
14	Intersection: picked in both classes	7.8
15	Symmetric difference: picked in exactly one class	7.8
17	Starred unpacking: `first` is Ana's tuple, `later` collects the other 3	7.5
18	Plain unpacking of that tuple into a name and a fruit	7.3

- Each class is a **list** because the order the answers came in matters for line 17, and two students can give the same answer.
- Each answer is a **tuple** because it is a fixed pair: one name, one fruit.
- The fruits are collected in **sets** because the questions are "which fruits?", not "how many votes?". If the program needed vote counts, a set would be the wrong choice, because it throws the repeats away.
- `name` is not used in the loops on lines 5 and 9, but unpacking still needs a name for each part.

Vocabulary

Term	Definition	Example
tuple	An ordered sequence that cannot be changed after it is made	`(2009, 7, 14)`
sequence	Values in a fixed order, each at an index from 0	a string, list or tuple
immutable	Cannot be changed in place; strings and tuples are immutable	`point[0] = 5` fails
`tuple()`	Converts a sequence into a tuple	`tuple("ab")`
trailing comma	A comma after the last item; makes a one-element tuple	`(70,)`
empty tuple	A tuple with no items	`()`
packing	Building a tuple from comma-separated values	`pair = 3, 4`
unpacking	Assigning a sequence's items to several names at once	`x, y = point`
swap	Exchange the values of two variables	`a, b = b, a`
starred unpacking	Unpacking where one starred name collects the leftover items as a list	`first, *rest = items`
set	An unordered collection of unique values	`{"S", "M", "L"}`
`set()`	Makes a set from a sequence, or an empty set	`set("banana")`
membership test	Asking whether a value is in a collection	`"M" in sizes`
hash	A number computed from a value that tells a set where to store and find it	used by `in` on a set
hashable	Can go in a set because it cannot change	numbers, strings, tuples
union	Values in either set	`a | b`
intersection	Values in both sets	`a & b`
difference	Values in the first set but not the second	`a - b`
symmetric difference	Values in exactly one of the two sets	`a ^ b`

Summary

- A **tuple** is an ordered, **immutable** sequence: index it, slice it, and use `len` and `in`, but build a new tuple to "change" it. The comma makes a tuple, so a one-element tuple needs a **trailing comma**: `(70,)`.
- **Unpacking** assigns a sequence's items to several names: `year, month, day = birthday`. The counts must match. It works in for loops too, which is how `for index, letter in enumerate(word):` works.
- `a, b = b, a` **swaps** two values, because Python builds the whole right side from the old values before assigning. In **starred unpacking**, `first, *rest = items`, one starred name collects the leftovers as a list.
- A **set** keeps each value once and has no order or indexes. Make one with braces or `set()` (never `{}` for an empty set), and use `sorted()` to show it in order. `in` on a set is fast at any size, and only **hashable** values such as numbers, strings and tuples can go in it.

- **Union** `|`, **intersection** `&`, **difference** `-` and **symmetric difference** `^` compare whole sets and give a new set.
- Choose a list for an ordered sequence that changes, a tuple for a fixed record, and a set for unique values and fast membership tests.

Practice

Do these in your `day07` folder. Type each program yourself, run it, and only then open the answer.

Question 16 Short answer

Minimum and maximum together. Given a list of numbers, put both its smallest and its largest value into one tuple, print the tuple, then unpack it into two names and print a sentence that uses them. (On Day 9 you will turn this into a function that **returns** the tuple; today, build the tuple directly.)

Question 17 Short answer

Words in common. Ask the user for two sentences, then print the words that appear in both, in alphabetical order. Treat `The` and `the` as the same word. If there are none, say so.

Question 18 Short answer

Unique characters. Ask the user for some text and report how many characters it has, how many **different** characters it has, and which ones they are, in order.

Review questions

Question 19 Tracing

What does this program print? Trace the two sets through the loop.

pairs_sets.py

```
1 pairs = [(1, "a"), (2, "b"), (1, "c"), (3, "a")]
2 numbers = set()
3 letters = set()
4 for number, letter in pairs:
5     numbers.add(number)
6     letters.add(letter)
7 print(len(numbers), len(letters))
8 print(sorted(numbers | {5}))
```

Question 20 Debugging

This program should count the different words in a phrase, which gives 4. It stops with an error. Quote the last line of the error, explain it, and write the fix.

seen_words.py

```
1 seen = {}
2 for word in "to be or not to be".split():
3     seen.add(word)
4 print(len(seen))
```

Question 21 Tracing

What does this program print?

order_pair.py

```
1 low, high = 9, 4
2 if low > high:
3     low, high = high, low
4 values = (low, high)
5 print(values, values[::-1])
```

Question 22 Multiple choice

What do these three lines print?

```
>>> set("listen") == set("silent")
>>> set("tool") == set("lot")
>>> len("tool") == len("lot")
```

A. `True`, `False`, `False`
B. `True`, `True`, `False`
C. `False`, `False`, `False`
D. `True`, `True`, `True`

Question 23 Short answer

Explain why `a ^ b` always gives the same set as `(a | b) - (a & b)`. Use `a = {1, 2, 3}` and `b = {2, 3, 4}` as an example.

Go further

The official Python Tutorial covers today's collections in Data Structures: Tuples and Sequences and Data Structures: Sets. Its sets section also shows set comprehensions, which you meet on Day 11.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- [ ] I can make a tuple, read its items, and explain why it cannot be changed. Reread: 7.1 Tuples: Groups That Do Not Change
- [ ] I can write a one-element tuple and explain what the comma does. Reread: 7.2 The Comma Makes the Tuple
- [ ] I can unpack a tuple or list into names, including in a for loop. Reread: 7.3 Unpacking: Many Names at Once
- [ ] I can swap two variables in one line and explain why it works. Reread: 7.4 Swapping Two Values
- [ ] I can use a starred name to collect the leftover items. Reread: 7.5 Starred Unpacking: Collecting the Rest
- [ ] I can use a set to remove duplicates, and I know a set has no order. Reread: 7.6 Sets: Only Unique Values
- [ ] I can explain why in is fast on a set and which values a set can hold. Reread: 7.7 Fast Membership Tests
- [ ] I can use union, intersection, difference and symmetric difference. Reread: 7.8 Set Operations

- [ ] I can choose a list, a tuple or a set for a job and justify the choice. Reread: 7.9 Choosing a List, a Tuple or a Set