Week 9 Tutorial

COMP10001 – Foundations of Computing

Semester 2, 2025

Clement Chau

- “List” Comprehensions
- Iterators and Itertools

Agenda

1. Week 9 Discussion – **Tutorial sheet** (~ 60 mins)
2. One-on-one Q&A (~ 50 mins)

9 (22/9)	Project 2 overview	Advanced Lecture	**AFL public holiday** **(no lecture)**	Week 9 tutorial sheet ↓ Week 9 tutorial solutions	• Ed worksheet 12 and 13 due (22/9 at 6 pm) • Project 2 release

Ed worksheets 12, 13 due (22/Sep, Monday at 6 pm)

No classes on Friday (26/Sep)!

Revision: List Comprehensions

```
cashier_3 = []
for item in cart:
    if item % 2 == 0:
        cashier_3.append(item)
```

Non-list comprehension

```
cashier_3 = [item for item in cart if item % 2 == 0]
```

List comprehension

Revision: (Types of) List Comprehensions

sem1-2025 > week-9 > list_comprehension.py > ...

```python
# List Comprehension
l = [i for i in range(10)]

# List Comprehension with If Condition
l = [i for i in range(10) if i % 2 == 0]

# List Comprehension with If/Else Condition
l = [i if i % 2 == 0 else i * i for i in range(10)]

# Nested Comprehension
l = [(i, j) for i in range(10) for j in range(10)]
```

Revision: (Types of) List Comprehensions

sem1-2025 > week-9 > list_comprehension.py > ...

```python
# List Comprehension
l = [i for i in range(10)]

# List Comprehension with If Condition
l = [i for i in range(10) if i % 2 == 0]

# List Comprehension with If/Else Condition
l = [i if i % 2 == 0 else i * i for i in range(10)]

# Nested Comprehension
l = [(i, j) for i in range(10) for j in range(10)]
```

Revision: Iterators

Run PYTHON

```python
list1 = [1, 5]
my_iterator = iter(list1)
print(next(my_iterator))
print(next(my_iterator))

# This will generate an error because we have reached the end.
print(next(my_iterator))
```

```
1
5
Traceback (most recent call last):
  File "/home/main.py", line 7, in <module>
    print(next(my_iterator))
          ~~~~^^^^^^^^^^^^^^
StopIteration
```

Program exited with code 1

Revision: Iterables

- **list**
- **str**
- **tuple**
- **set**
- **dict**
- file objects (We'll talk more about this next week!)

Revision: Iterators vs Sequences

Sequences:

- Have *random access* (you can access any element in the sequence, as many times as you like)
- No position tracking within the sequence
- You can use `len()` to calculate the length
- Must be finite
- You can traverse it many times

Iterators:

- No *random access*
- Remembers where you are up to
- Cannot use `len()`
- Can be infinite
- You can only traverse it once, forwards.

Revision: Itertools (Cycle)

Run PYTHON

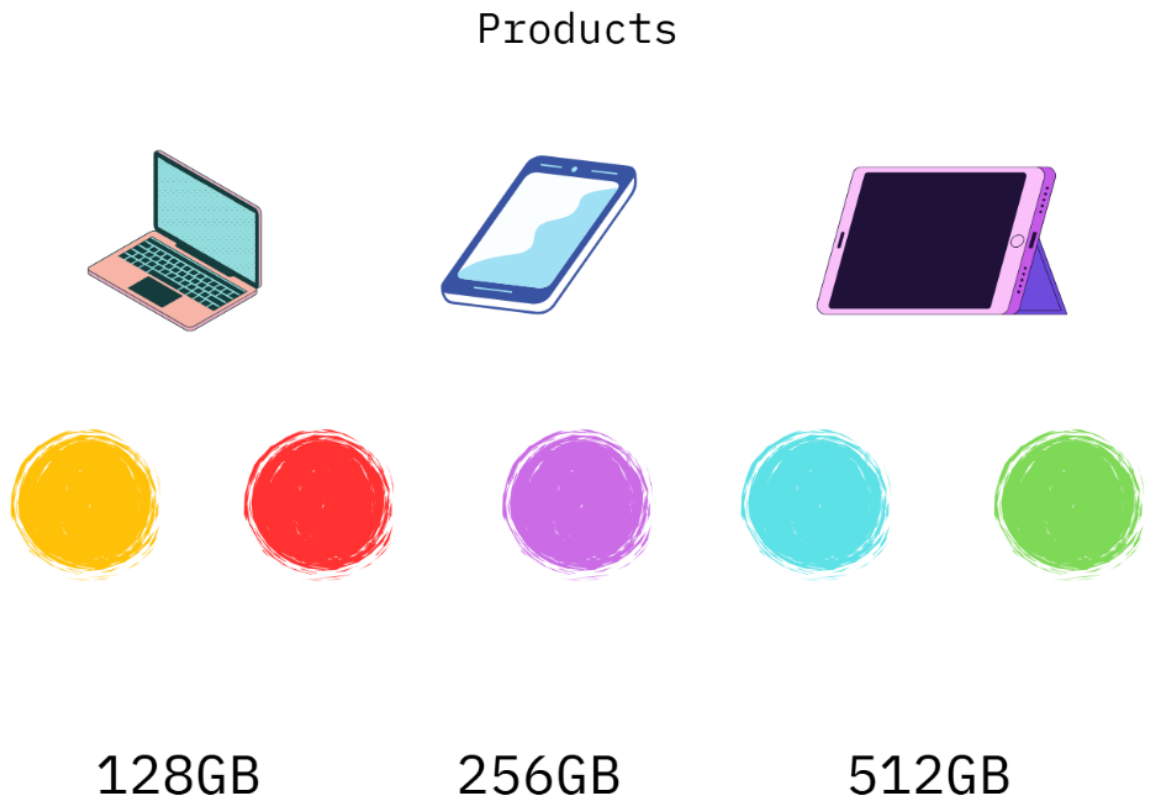
```python
from itertools import cycle
COUNT = 4
my_iterator = cycle("ABC")
for _i in range(COUNT):
    print(next(my_iterator))
```

```
A
B
C
A
```

✓ Program exited with code 0

Revision: Itertools (Product)

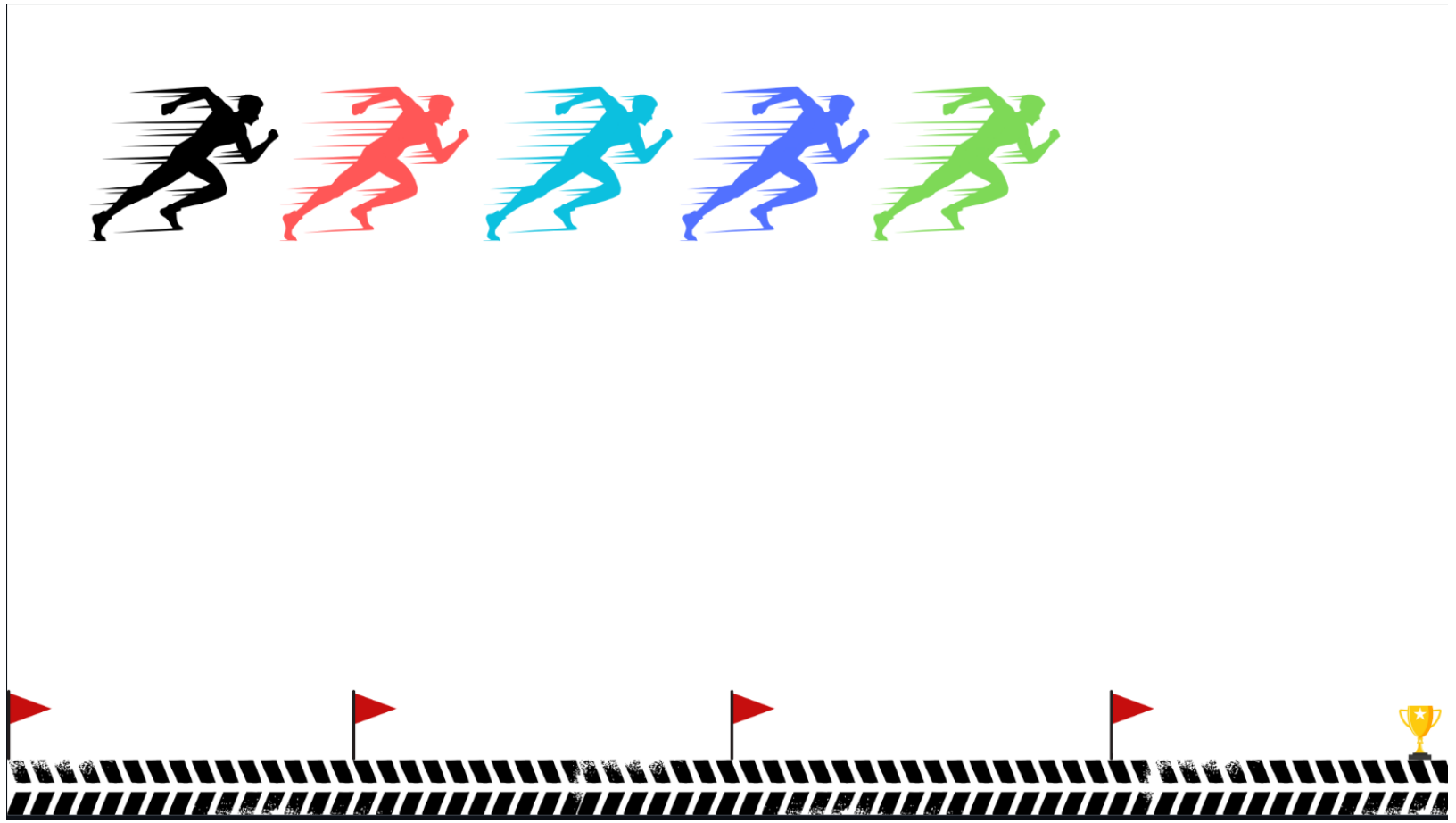
I have these options for devices, colors, and storage size. How many apple products can I come up with?



Credit: Daksh Agrawal

Revision: Itertools (Permutations)

I want to know all possible orderings for a 4-man relay sprint from 5 candidates



Credit: Daksh Agrawal

Revision: Itertools (Permutations)

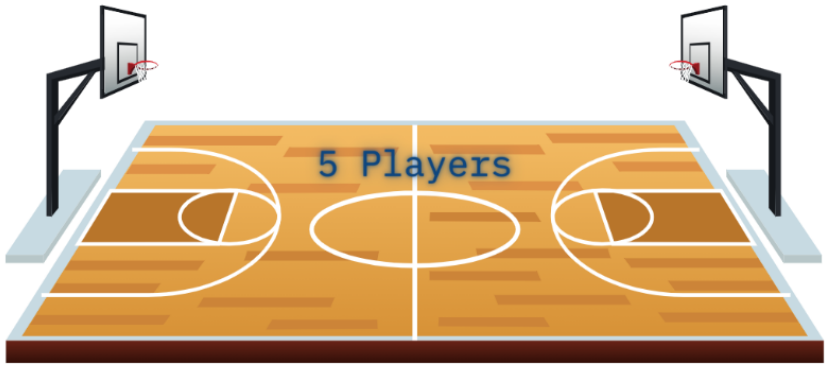
```python
import itertools

p_teams = itertools.permutations(
    iterable: ["A", "B", "C", "D", "E"],
    r: 4
)

print(list(p_teams))
```

```
[('A', 'B', 'C', 'D'), ('A', 'B', 'C', 'E'), ('A', 'B', 'D', 'C'), ('A', 'B', 'D', 'E'), ('A', 'B', 'E', 'C'), ('A', 'B', 'E', 'D'), ('A', 'C', 'B', 'D'), ('A', 'C', 'B',
'E'), ('A', 'C', 'D', 'B'), ('A', 'C', 'D', 'E'), ('A', 'C', 'E', 'B'), ('A', 'C', 'E', 'D'), ('A', 'D', 'B', 'C'), ('A', 'D', 'B', 'E'), ('A', 'D', 'C', 'B'), ('A', 'D',
'C', 'E'), ('A', 'D', 'E', 'B'), ('A', 'D', 'E', 'C'), ('A', 'E', 'B', 'C'), ('A', 'E', 'B', 'D'), ('A', 'E', 'C', 'B'), ('A', 'E', 'C', 'D'), ('A', 'E', 'D', 'B'), ('A',
'E', 'D', 'C'), ('B', 'A', 'C', 'D'), ('B', 'A', 'C', 'E'), ('B', 'A', 'D', 'C'), ('B', 'A', 'D', 'E'), ('B', 'A', 'E', 'C'), ('B', 'A', 'E', 'D'), ('B', 'C', 'A', 'D'),
('B', 'C', 'A', 'E'), ('B', 'C', 'D', 'A'), ('B', 'C', 'D', 'E'), ('B', 'C', 'E', 'A'), ('B', 'C', 'E', 'D'), ('B', 'D', 'A', 'C'), ('B', 'D', 'A', 'E'), ('B', 'D', 'C',
'A'), ('B', 'D', 'C', 'E'), ('B', 'D', 'E', 'A'), ('B', 'D', 'E', 'C'), ('B', 'E', 'A', 'C'), ('B', 'E', 'A', 'D'), ('B', 'E', 'C', 'A'), ('B', 'E', 'C', 'D'), ('B', 'E',
'D', 'A'), ('B', 'E', 'D', 'C'), ('C', 'A', 'B', 'D'), ('C', 'A', 'B', 'E'), ('C', 'A', 'D', 'B'), ('C', 'A', 'D', 'E'), ('C', 'A', 'E', 'B'), ('C', 'A', 'E', 'D'), ('C',
'B', 'A', 'D'), ('C', 'B', 'A', 'E'), ('C', 'B', 'D', 'A'), ('C', 'B', 'D', 'E'), ('C', 'B', 'E', 'A'), ('C', 'B', 'E', 'D'), ('C', 'D', 'A', 'B'), ('C', 'D', 'A', 'E'),
('C', 'D', 'B', 'A'), ('C', 'D', 'B', 'E'), ('C', 'D', 'E', 'A'), ('C', 'D', 'E', 'B'), ('C', 'E', 'A', 'B'), ('C', 'E', 'A', 'D'), ('C', 'E', 'B', 'A'), ('C', 'E', 'B',
'D'), ('C', 'E', 'D', 'A'), ('C', 'E', 'D', 'B'), ('D', 'A', 'B', 'C'), ('D', 'A', 'B', 'E'), ('D', 'A', 'C', 'B'), ('D', 'A', 'C', 'E'), ('D', 'A', 'E', 'B'), ('D', 'A',
'E', 'C'), ('D', 'B', 'A', 'C'), ('D', 'B', 'A', 'E'), ('D', 'B', 'C', 'A'), ('D', 'B', 'C', 'E'), ('D', 'B', 'E', 'A'), ('D', 'B', 'E', 'C'), ('D', 'C', 'A', 'B'), ('D',
'C', 'A', 'E'), ('D', 'C', 'B', 'A'), ('D', 'C', 'B', 'E'), ('D', 'C', 'E', 'A'), ('D', 'C', 'E', 'B'), ('D', 'E', 'A', 'B'), ('D', 'E', 'A', 'C'), ('D', 'E', 'B', 'A'),
('D', 'E', 'B', 'C'), ('D', 'E', 'C', 'A'), ('D', 'E', 'C', 'B'), ('E', 'A', 'B', 'C'), ('E', 'A', 'B', 'D'), ('E', 'A', 'C', 'B'), ('E', 'A', 'C', 'D'), ('E', 'A', 'D',
'B'), ('E', 'A', 'D', 'C'), ('E', 'B', 'A', 'C'), ('E', 'B', 'A', 'D'), ('E', 'B', 'C', 'A'), ('E', 'B', 'C', 'D'), ('E', 'B', 'D', 'A'), ('E', 'B', 'D', 'C'), ('E', 'C',
'A', 'B'), ('E', 'C', 'A', 'D'), ('E', 'C', 'B', 'A'), ('E', 'C', 'B', 'D'), ('E', 'C', 'D', 'A'), ('E', 'C', 'D', 'B'), ('E', 'D', 'A', 'B'), ('E', 'D', 'A', 'C'), ('E',
'D', 'B', 'A'), ('E', 'D', 'B', 'C'), ('E', 'D', 'C', 'A'), ('E', 'D', 'C', 'B')]
```

Credit: Daksh Agrawal

Revision: Itertools (Combinations)

How many teams of 5 can I form with 7 basketball players?



Credit: Daksh Agrawal

Revision: Itertools (Combinations)

```python
import itertools

p_teams = itertools.combinations(
    iterable: ["A", "B", "C", "D", "E", "F", "G"],
    r: 5
)

print(list(p_teams))
```

```
[('A', 'B', 'C', 'D', 'E'), ('A', 'B', 'C', 'D', 'F'), ('A', 'B', 'C', 'D', 'G'), ('A',
 'B', 'C', 'E', 'F'), ('A', 'B', 'C', 'E', 'G'), ('A', 'B', 'C', 'F', 'G'), ('A', 'B',
 'D', 'E', 'F'), ('A', 'B', 'D', 'E', 'G'), ('A', 'B', 'D', 'F', 'G'), ('A', 'B', 'E',
 'F', 'G'), ('A', 'C', 'D', 'E', 'F'), ('A', 'C', 'D', 'E', 'G'), ('A', 'C', 'D', 'F',
'G'), ('A', 'C', 'E', 'F', 'G'), ('A', 'D', 'E', 'F', 'G'), ('B', 'C', 'D', 'E', 'F'),
('B', 'C', 'D', 'E', 'G'), ('B', 'C', 'D', 'F', 'G'), ('B', 'C', 'E', 'F', 'G'), ('B',
                'D', 'E', 'F', 'G'), ('C', 'D', 'E', 'F', 'G')]
```

Credit: Daksh Agrawal

Revision: Groupby

Run PYTHON

```python
from itertools import groupby

def get_first_letter(x):
    return x[0]

my_iterable = groupby(("AB", "AD", "BA", "BC", "BD", "DD"), get_first_letter)
for category, contents in my_iterable:
    # contents is an iterable, so needs to be converted into a list
    print(category, list(contents))
```

```
A ['AB', 'AD']
B ['BA', 'BC', 'BD']
D ['DD']
```

✓ Program exited with code 0

TuteSheet W9 – Exercises Q1(a)

1. A list comprehension is a shortcut notation used to accomplish simple iteration tasks that construct a list in one line of code. List comprehensions are formed by wrapping a pair of brackets around `<expression> <for iteration statement> <optional if filter condition>`. The iteration statement will be run and for each iteration, the result of the expression will be added to the list. If a filter condition is included, the object will only be added if that condition evaluates to `True`.

   Evaluate the following list comprehensions. For each one, also write some python code to generate the same list without using a comprehension.

   (a) `[(name, 0) for name in ("evelyn", "alex", "sam")]`

   Output: `[('evelyn', 0), ('alex', 0), ('sam', 0)]`

   w/o using a list comprehension:

```
my_list = []
for name in ("evelyn", "alex", "sam"):
    my_list.append((name, 0))
```

TuteSheet W9 – Exercises Q1(b), (c)

Evaluate the following list comprehensions. For each one, also write some python code to generate the same list without using a comprehension.

(b) `[i**2 for i in range(5) if i % 2 == 1]`

Output:
```
[1, 9]
```

w/o using a list comprehension:
```
my_list = []
for i in range(5):
    if i % 2 == 1:
        my_list.append(i**2)
```

(c) `"".join([letter.upper() for letter in "python"])`

Output:
```
'PYTHON'
```

w/o using a list comprehension:
```
my_list = []
for letter in "python":
    my_list.append(letter.upper())
my_str = "".join(my_list)
```

if we don't require the list:
```
my_str = "python".upper()
```

TuteSheet W9 – Exercises Q1(d)

Evaluate the following list comprehensions. For each one, also write some python code to generate the same list without using a comprehension.

(d) `[(row, col) for row in range(3, 5) for col in range(2)]`

Output:

```
[(3, 0), (3, 1), (4, 0), (4, 1)]
```

w/o using a list comprehension:

```
my_list = []
for row in range(3, 5):
    for col in range(2):
        my_list.append((row, col))
```

TuteSheet W9 – Exercises Q2

2. What happens if we use curly brackets instead of square brackets around a "list" comprehension? What happens if we use parentheses?

If we use curly brackets `{}` around a comprehension syntax, it will evaluate into a **set**.

(e.g.)

```
{n**2 for n in range(1, 10)}
```

```
{1, 4, 9, 16, 25, 36, 49, 64, 81}
```

If we use the `key:value` syntax for the expression part of a comprehension with curly brackets `{}`, it will evaluate into a **dictionary**.

(e.g.)

```
{word:word.upper() for word in ["apple", "banana"]}
```

```
{'apple': 'APPLE', 'banana': 'BANANA'}
```

If we use parentheses `()`, it creates an object called a **generator, NOT** a **tuple**.

A **generator object** is an iterator that produces values one at a time, only when you ask for them, usually with a loop or by calling `next()`.

https://www.geeksforgeeks.org/generators-in-python/

TuteSheet W9 – Exercises Q3

3. For a list such as `words = ['pencil', 'highlighter', 'paper-clip', 'ruler', 'pen']`, write a comprehension that gives the following:

   (a) A list containing only the words that start with 'p'

```
[word for word in words if word.startswith('p')]
```

output: `['pencil', 'paper-clip', 'pen']`

   (b) A dictionary mapping each word to their length

```
{word: len(word) for word in words}
```

output: `{'pencil':6, 'highlighter':11, 'paper-clip':10, 'ruler':5, 'pen':3}`

   (c) A set of every character used in `words`

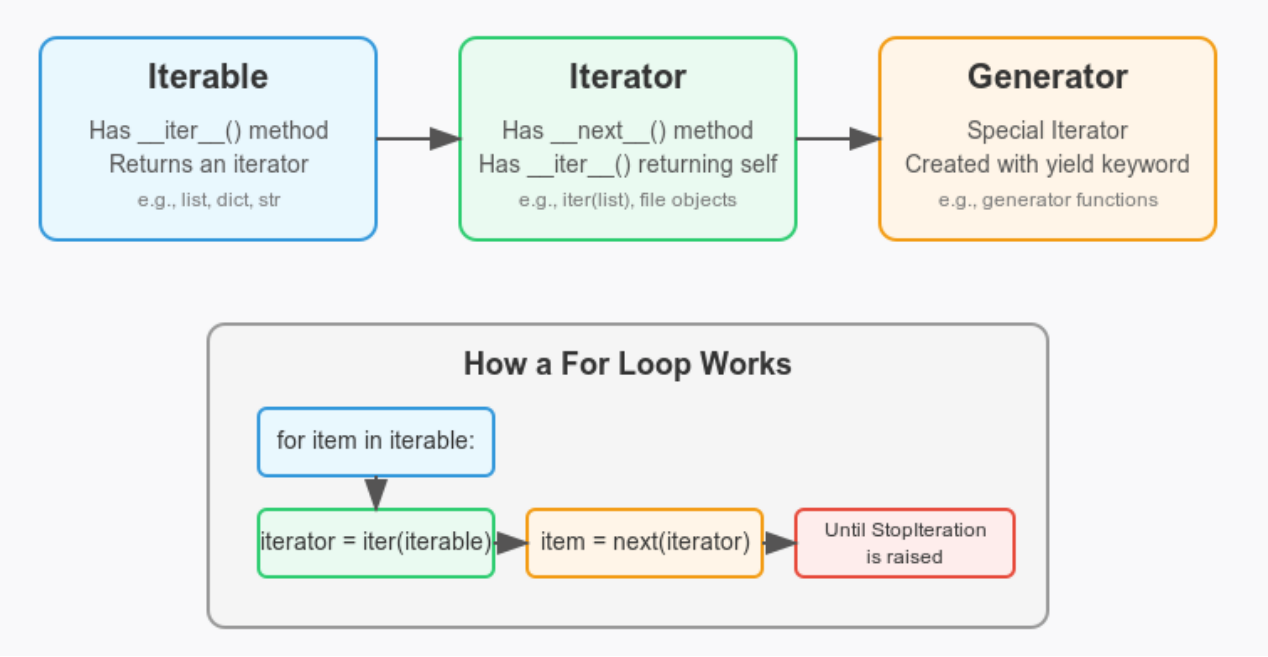
```
{letter for word in words for letter in word}
```

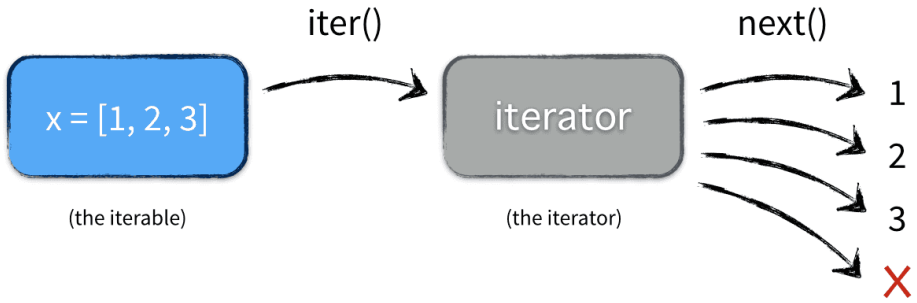
output: `{'-', 'a', 'c', 'e', 'g', 'h', 'i', 'l', 'n', 'p', 'r', 't', 'u'}`

TuteSheet W9 – Exercises Q4(a)

4.

- An *iterable* is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `set`, and file objects.
- An *iterator* is an object representing a stream of data. Repeated calls of `next(iterator)` return successive items in the stream. When no more items are available a `StopIteration` exception is raised. We can convert an iterable into an iterator using the `iter()` function.





TuteSheet W9 – Exercises Q4(a)

4.

- An *iterable* is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `set`, and file objects.
- An *iterator* is an object representing a stream of data. Repeated calls of `next(iterator)` return successive items in the stream. When no more items are available a `StopIteration` exception is raised. We can convert an iterable into an iterator using the `iter()` function.

Convert these iterable objects into iterators and extract two elements into `first` and `second` variables:

(a)
```
iterable = "ABCDEFGH"
```

A:

```
iterable = "ABCDEFGH"
iterator = iter(iterable)
first = next(iterator)  # 'A'
second = next(iterator) # 'B'
```

- ***iterable*** : a **`str`**.
- **`iter()`** on a **string**: returns **one character at a time**, starting from index 0.
- **`next()`** : the next character from the iterator

TuteSheet W9 – Exercises Q4(b)

4.
- An *iterable* is an object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `set`, and file objects.
- An *iterator* is an object representing a stream of data. Repeated calls of `next(iterator)` return successive items in the stream. When no more items are available a `StopIteration` exception is raised. We can convert an iterable into an iterator using the `iter()` function.

Convert these iterable objects into iterators and extract two elements into `first` and `second` variables:

(b)
```
iterable = {(0, 0), (0, 1), (1, 0), (1, 1)}
```

A:

```
iterable = {(0, 0), (0, 1), (1, 0), (1, 1)}
iterator = iter(iterable)
first = next(iterator)  #e.g., (0, 1)
second = next(iterator) #e.g., (1, 0)
```

- ***iterable*** : a **`set`** **of tuples**.
- Since sets are **unordered**, the iteration order is **not guaranteed**.
- **`iter()`** on a **set**: returns **the elements one by one.**
- **`next()`** : the next tuple from the iterator

TuteSheet W9 – Exercises Q5

5. The `itertools` library provides functions that create memory-efficient iterators for a variety of sequences and combinatoric sets. Have a look at the Python documentation page for the `itertools` library and the itertools session in Ed Worksheet 14. The ones we will focus on are `cycle`, `product`, `permutations`, `combinations`, and `groupby`.

Q6 **`cycle()`** returns an iterator that **repeats its elements indefinitely.**
(e.g. *cycle('ABC') → A B C A B C A B C ...*)

Q7 **`product()`** returns an iterator that generates **all possible ordered pairs, as tuples (similar to nested for loops)** (e.g. *product('ABC', 'xy') → Ax Ay Bx By Cx Cy*)

Q8 **`permutations()`** returns an iterator of **all possible ordered permutations of length r.**
(e.g. *permutations('ABC', 2) → AB AC BA BC CA CB*)

Q8 **`combinations()`** returns an iterator of **all possible unordered combinations of length r, without repetition**.
(e.g. *combinations('ABC', 2) → AB AC BC*)

Q9 **`groupby()`** returns an iterator that generates **consecutive keys and groups** from the iterable.
(e.g. *groupby(['A','B','DEF'], len) → (1, A B) (3, DEF)*)

https://docs.python.org/3/library/itertools.html

TuteSheet W9 – Exercises Q6

6. What output does the following code print? Try changing the `while` loop to get the same output.

```
import itertools
beatboxer = itertools.cycle(['boots', 'and', 'cats', 'and'])

for count in range(9):
    print(next(beatboxer))
```

output: will print two iterations of `boots and cats and`, and will end with `boots`

```
boots
and
cats
and
boots
and
cats
and
boots
```

Using `while` loop

```
import itertools
beatboxer = itertools.cycle(['boots', 'and', 'cats', 'and'])

COUNT = 9
while COUNT:
    print(next(beatboxer))
    COUNT -= 1
```

TuteSheet W9 – Exercises Q7

7. A comedy series has episode names in an `<animal> in <place>` format, for example, "Elephants in Melbourne". Using a single loop, write some code to print out every possible episode name, given:

```
animals = ['cats', 'dogs', 'hamsters', 'elephants']
places = ['Melbourne', 'space', 'the supermarket']
```

```
import itertools

animals = ['cats', 'dogs', 'hamsters', 'elephants']
places = ['Melbourne', 'space', 'the supermarket']

for animal, place in itertools.product(animals, places):
    print(f"{animal.title()} in {place}")
```

```
Cats in Melbourne
Cats in space
Cats in the supermarket
Dogs in Melbourne
Dogs in space
Dogs in the supermarket
Hamsters in Melbourne
Hamsters in space
Hamsters in the supermarket
Elephants in Melbourne
Elephants in space
Elephants in the supermarket
```

TuteSheet W9 – Exercises Q8

8. Compare the output of this code. What do you notice about the difference between combinations and permutations?

```
import itertools

numbers = [1, 2, 3]
print("combinations:", list(itertools.combinations(numbers, 2)))
print("permutations:", list(itertools.permutations(numbers, 2)))
```

```
combinations: [(1, 2), (1, 3), (2, 3)]
permutations: [(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
```

`combinations()` returns an iterator of **all possible unordered combinations of length r, without repetition**.

*In this example, combinations show the ways of **choosing 2 elements** from the numbers list, but **the order** they are selected **isn't taken in to account**. Choosing 1 then 2 is the **same** as choosing 2 then 1, so only the first is included as a tuple (1, 2).*

`permutations()` returns an iterator of **all possible ordered permutations of length r.**

*In this example, Permutations show the different ways of choosing the 2 elements as a sequence, so the **order** that they are chosen **matters**. Choosing 1 then 2 is **different** to choosing 2 then 1, so both (1, 2) and (2, 1) are included in the result.*

TuteSheet W9 – Exercises Q9

9. What output does the following code print? What happens if we don't sort the `aussie_animals` list before doing `groupby`?

```
import itertools

aussie_animals = ["Possum", "Echidna", "Emu", "Koala", "Platypus", "Wombat"]

for key, group in itertools.groupby(sorted(aussie_animals), lambda x: x[0]):
    print(key, list(group))
```

iterable ***key***

```
E ['Echidna', 'Emu']
K ['Koala']
P ['Platypus', 'Possum']
W ['Wombat']
```

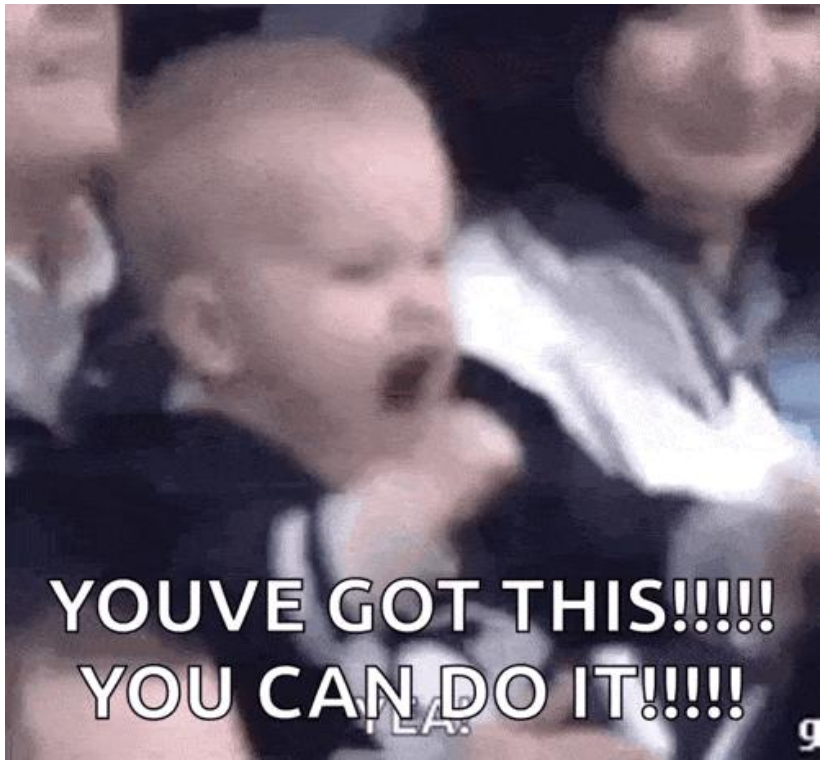
If we don't sort the aussie_animals list before doing groupby, then the output is:

```
P ['Possum']
E ['Echidna', 'Emu']
K ['Koala']
P ['Platypus']
W ['Wombat']
```

It generates a break or new group every time the value of the key function changes (which is why it is usually necessary to have sorted the data using the same key function)

https://docs.python.org/3/library/itertools.html#itertools.groupby

TuteSheet W9 – Problems Q1-Q4



Independent Work

- **Next due dates:**
  - Your **Project 2** will be released **on Thursday, September 25th, 1pm**.
    - For any questions, please go to the **First Year Centre 12pm-2pm every weekday** in Level 3, Melbourne Connect or ask in the **Ed Discussion** Forums!
    - We can only provide **very limited**, general guidance.
  - Ed Worksheets **14** and **15** is **due next next Monday, October 6th, 6pm**.
- **Raise your hand** if you have any questions!

Scan here for annotated slides