INF1-IP

Introduction to Programming

Lecture 17 – Collections 3: Searching and Sorting

> Recap: equality and identity

Remember that there is a difference between:

two objects having the same value

two names referring to the same object

Expression	Meaning
`a == b`	Do the objects have **equal values**?
`a is b`	Do the names refer to the **same object**?
`a is not b`	Do the names refer to **different objects**?

> Recap: Collections

Data structures are ways to organise and store data so your program can use it easily.

Python has four main built-in data structures:

- **List:** An ordered collection you can change. It allows duplicates.
- **Tuple:** An ordered collection you cannot change after creating it.
- **Dictionary:** Stores pairs of keys and values, so you can look up a value using its key.
- **Set:** A collection of unique items, with no guaranteed order.

> Sorting

There are some built-in ways of sorting data within collections (some we have seen already):

Operation	Result
`items.index(value)`	Returns the first index of `value`; raises `ValueError` if not found.
`mapping.keys()`	Returns a view of the dictionary's keys.
`mapping.items()`	Returns a view of the dictionary's `(key, value)` pairs.
`mapping.get(key, default)`	Returns the value for `key`, or `default` if the key is missing.
`members.discard(value)`	Removes `value` from the set; does nothing if it is missing.

> Recap: Sorting

We also looked at sorting items using `sorted()` and `.sort()`

Operation	Result
`sorted(items)`	Returns a **new list** in ascending order; leaves the original unchanged.
`items.sort()`	Sorts the **original list** in ascending order; returns `None`.

`.sort()` is a list method – you can't use it on any other collection

`sorted()` takes in any collection* and returns a sorted list.

* It only works on dictionaries if the keys are comparable

> Sorting – key functions

With a dictionary, there can be lots of different ways to sort.

```
products = [
    {"name": "Notebook", "price": 4},
    {"name": "Backpack", "price": 25},
    {"name": "Pen", "price": 2},
]
```

We could sort this alphabetically by name, or we could sort it by price.

sorted() can take an extra argument to tell it how to sort. This is called a key function.

> Sorting – key functions

With a dictionary, there can be lots of different ways to sort.

```
products = [
    {"name": "Notebook", "price": 4},
    {"name": "Backpack", "price": 25},
    {"name": "Pen", "price": 2},
]

def price_of(product):
    return product["price"]

ordered_products = sorted(products, key=price_of, reverse=True)
print(ordered_products)
```

We create a key function called price_of which takes in a dictionary and returns the key to search on.

When we call sorted() we use the key function to tell it how to search.

`reverse=True` tells Python to sort in reverse order.

> Sorting – key functions

With a dictionary, there can be lots of different ways to sort.

```
products = [
    {"name": "Notebook", "price": 4},
    {"name": "Backpack", "price": 25},
    {"name": "Pen", "price": 2},
]

def price_of(product):
    return product["price"]

ordered_products = sorted(products, key=price_of, reverse=True)
print(ordered_products)
```

We create a key function called price_of which takes in a dictionary and returns the key to search on.

When we call sorted() we use the key function to tell it how to search.

`reverse=True` tells Python to sort in reverse order.

```
[{'name': 'Backpack', 'price': 25}, {'name': 'Notebook', 'price': 4}, {'name': 'Pen', 'price': 2}
```

> Deep and shallow copying

A *shallow copy* of a list creates a new list, but the items in that list do not exist independently – they are references to the items in the original list.

A *deep copy* of a list creates an entirely new list, with items unrelated to the original list.

> Deep and shallow copying

.copy() creates a shallow copy – a new outer copy that references the original items
.deepcopy() creates a deep copy – a completely new collection

This matters if you want your collection contains mutable objects that you might want to change.

If your collectable that contains only numbers or strings, there is no practical difference between these two.

> Deep and shallow copying

When you are copying, always keep in mind whether you are creating something new or creating a copy.

➔ example

> Membership and linear search

We can use `in` to determine whether a collection contains something.

```
fruits = ["apple", "banana", "orange"]
prices = {"apple": 50, "banana": 30}

print("banana" in fruits)        # True
print("apple" in prices)         # True
print(50 in prices)              # False
print(50 in prices.values())     # True
```

> Membership and linear search

But membership only allows to know if something is in a collection or not.

Linear search goes through collections one item at a time to retrieve more complex information (like position) or to alter a particular item.

We can do this through a loop.

```
def find_position(values, wanted):
    for position, value in enumerate(values):
        if value == wanted:
            return position
    return None


colours = ["blue", "green", "red", "yellow"]
print(find_position(colours, "yellow"))  # 3
print(find_position(colours, "purple"))  # None
```

! Watch out for index 0

We might want to create code that does something if an item is found in a collection and doesn't if it isn't. But if we do this by looking at the position of an object, remember:

The index of the first item in a list is 0

0 can be equated to false.

```
colours = ["blue", "green", "red", "yellow"]
result = find_position(colours, "blue")  # Returns index 0

# Incorrect: 0 is falsy, so this reports "Not found".
if result:
    print("Found at index", result)
else:
    print("Not found")

# Correct: 0 is a valid index; only None means "not found".
if result is not None:
    print("Found at index", result)
else:
    print("Not found")
```

Let's practice that

Write `find_by_id(books, target)` for the dictionaries below. Return the **first dictionary** whose "id" matches target, or None if no match is found. Test both outcomes. The list does not need to be sorted.

```
books = [
    {"id": "B17", "title": "Learning Python"},
    {"id": "B04", "title": "Web Design Basics"},
    {"id": "B23", "title": "Introduction to Databases"}
]
```

Let's practice that

```python
def find_by_id(books, target):
    for book in books:
        if book["id"] == target:
            return book
    return None


books = [
    {"id": "B17", "title": "Learning Python"},
    {"id": "B04", "title": "Web Design Basics"},
    {"id": "B23", "title": "Introduction to Databases"}
]

# Test a matching ID.
print(find_by_id(books, "B04"))

# Test an ID that is absent.
print(find_by_id(books, "B99"))
```

> Search strategies

Linear search is straightforward and effective, but it is not very efficient. We have to look through every member of the collection until we find what we are looking for.

This doesn't really matter for small collections, but large collections efficiency is really important.

There are many ways to search – today we will look at linear search and binary search.

There are also many ways to sort. We will look at bubble sort and insertion sort.

> Searching and sorting

For most search strategies, we first need to sort the collections so we know what order things are in.

> Binary Search

Binary search cuts a collection in two by looking at the middle of the collection and asking:

is this before or after what we are looking at?

Remember that the collection has already been sorted.

e.g.,
A list of students signed up for IP. We are looking for a student called Morris, Jane.
Split the list of students in two
a) if the middle student is called North, Claire - cut the second half because we know Miles must be in the first half.
b) If the middle student is called Jones, Paul - cut the first half.

Let's practice that

```python
def binary_search(items, target):
    low = 0                             # the first index of the list
    high = len(items) - 1               # the last index of the list

    while low <= high:                  # we have not passed half way (low is lower than high)
        middle = (low + high) // 2      # find the middle of the two values
        candidate = items[middle]
        if candidate == target:         # if that is what we are looking for, return it
            return middle
        if candidate < target:          # if it is too low, ditch the first half of the list and make low the smallest of the second half
            low = middle + 1
        else:
            high = middle - 1           # otherwise set high to the highest of the first half (ditching the second half)
    return None


values = [3, 8, 12, 17, 25, 31, 44]     # remember these have to be sorted or it doesn't work!
print(binary_search(values, 25))
print(binary_search(values, 26))
```

> Bubble sort

We have looked at an efficient way of search a sorted list. We don't need to do the sorting ourselves because this is built in. But learning about sorting is helpful to understand how it works and why efficiency is important.

Bubble sort allows larger values to 'bubble up' to the end of the list.
How?

- Look at a given two neighbouring values (usually the first two).
- Is the first one large than the second? If so, swap them?
- If not, leave it as is and go on to the next pair (usually the second and third).

> Bubble sort

```
# Start with the unsorted list.
["Edinburgh Castle", "Royal Mile", "Arthur's Seat", "Calton Hill", "Informatics Forum"]
```

```
# Pass 1: Edinburgh Castle and Royal Mile are in order; swap Royal Mile and Arthur's Seat.
["Edinburgh Castle", "Arthur's Seat", "Royal Mile", "Calton Hill", "Informatics Forum"]
```

```
# Pass 1: Swap Royal Mile and Calton Hill.
["Edinburgh Castle", "Arthur's Seat", "Calton Hill", "Royal Mile", "Informatics Forum"]
```

```
# Pass 1: Swap Royal Mile and Informatics Forum; Royal Mile reaches its final position.
["Edinburgh Castle", "Arthur's Seat", "Calton Hill", "Informatics Forum", "Royal Mile"]
```

```
# Pass 2: Swap Edinburgh Castle and Arthur's Seat.
["Arthur's Seat", "Edinburgh Castle", "Calton Hill", "Informatics Forum", "Royal Mile"]
```

```
# Pass 2: Swap Edinburgh Castle and Calton Hill. Remaining comparisons need no swaps.
["Arthur's Seat", "Calton Hill", "Edinburgh Castle", "Informatics Forum", "Royal Mile"]
```

```
# Pass 3: No swaps occur, confirming the list is sorted.
["Arthur's Seat", "Calton Hill", "Edinburgh Castle", "Informatics Forum", "Royal Mile"]
```

> Bubble sort

How does this work?

1. Make a copy of the list so the original stays unchanged
2. Starting at the beginning, compare each pair of neighbouring items:
   - If the left item is greater than the right item, swap them.
   - Record whether any swaps occurred.
3. At the end of the pass, the largest item in the unsorted section is in its final position.
4. On the next pass, exclude that item from the comparisons.
5. Repeat until either:
   - A complete pass makes **no swaps**, meaning the list is sorted.
   - All necessary passes have been completed.
6. Return the sorted copy.

> Insertion sort

In this approach, we start with a list of length one and then insert other items in one by one to the correct position.

1. Treat the first item as an already-sorted section.
2. Take the next item as the **current item**.
3. Compare it with the items in the sorted section, working from right to left:
   - Shift each item greater than the current item one position to the right.
   - Stop when you reach a smaller or equal item, or the beginning of the list.
4. Insert the current item into the gap.
5. Repeat for each remaining item, growing the sorted section one item at a time.
6. Once all items have been inserted, the list is sorted.

> Insertion sort

```
# Start: treat Edinburgh Castle as the sorted section.
["Edinburgh Castle", "Royal Mile", "Arthur's Seat", "Calton Hill", "Informatics Forum"]
```

```
# Royal Mile already belongs after Edinburgh Castle; no change is needed.
["Edinburgh Castle", "Royal Mile", "Arthur's Seat", "Calton Hill", "Informatics Forum"]
```

```
# Insert Arthur's Seat at the beginning, shifting Edinburgh Castle and Royal Mile right.
["Arthur's Seat", "Edinburgh Castle", "Royal Mile", "Calton Hill", "Informatics Forum"]
```

```
# Insert Calton Hill after Arthur's Seat, shifting Edinburgh Castle and Royal Mile right.
["Arthur's Seat", "Calton Hill", "Edinburgh Castle", "Royal Mile", "Informatics Forum"]
```

```
# Insert Informatics Forum before Royal Mile, shifting Royal Mile right. The list is sorted.
["Arthur's Seat", "Calton Hill", "Edinburgh Castle", "Informatics Forum", "Royal Mile"]
```

{ Let's code that

Let's code that

```
def insertion_sort(values):
    result = values.copy()

    for index in range(1, len(result)):
        value_to_insert = result[index]
        position = index

        while position > 0 and result[position - 1] > value_to_insert:
            result[position] = result[position - 1]
            position -= 1

        result[position] = value_to_insert

    return result


sample = [5, 2, 4, 6, 1, 3]
print(insertion_sort(sample))
```

> Efficiency

Linear search vs binary search:

	Linear search	**Binary search**
Approach	Checks items one at a time	Checks the middle item, then halves the remaining search area
Requires sorted data?	No	**Yes**
Best case	O(1) — first item matches	O(1) — middle item matches
Average / worst case	O(n)	O(log n)
For 1,000,000 items	Up to 1,000,000 items checked	At most about 20 items checked

Binary search is massively more efficient *if your list is already sorted*.

> Efficiency

How expensive is sorting?

Algorithm	Best case	Average case	Worst case
Bubble sort (with early stopping)	O(n)	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Python built-in sort (`sorted()` / `.sort()`)	O(n)	O(n log n)	O(n log n)

Built-in sort is much more efficient – bubble sort is usually used for teaching. Insertion sort can be efficient if the list is already nearly sorted.

> Efficiency

So what should you do?

- If you have a small collection like the ones we have seen in our examples, it doesn't really matter – it will be really fast anyway.
- If you have a much bigger collection it makes a big difference:
  - If you are only going to search a collection once, use linear search – it is quicker to do linear search than sorting plus binary search
  - If you are going to search a collection many times, it is always worth sorting it and then using efficient searching methods.

> Summary

- Difference between equality and identity
- Shallow and deep copying
- Two kinds of search:
  - Linear
  - Binary
- Two kinds of sorting:
  - Bubble
  - Insertion