Module 5: Sorting and Dictionaries

Exercise

If you have not already, get prepared for class by downloading the start code:

```
!wget https://student.cs.uwaterloo.ca/~cs114/src/module-05-start.ipynb
```

Discuss the previous module with your neighbour.

- What are the differences between a `list` and a `tuple`?
- What can we do with a `range` object?

Sorting

Quick! Is 7256 in this list?

```
[5421, 4448, 8635, 2444, 3711, 3477, 4367, 1793, 5484, 2508, 9668, 3643, 4257, 9226,
 6525, 2511, 6087, 6259, 3256, 1205, 7471, 4749, 7247, 7699, 5423, 4845, 4860, 6055]
```

Now I have similar data, but sorted in increasing order. Is 7256 in this list?

```
[1041, 1952, 2385, 4743, 4896, 5008, 5081, 5417, 5555, 5612, 5896, 5960, 6278, 6294,
 6391, 6864, 7196, 7339, 7428, 7451, 7624, 7741, 8240, 8461, 9098, 9164, 9408, 9607]
```

In the first case, you have to look at each item, one by one.
You can't be sure until you look at every item.

In the second case, as soon as you look at the `6391` you can see that it can't be in first row.
You can then quickly discard the second half of the second row, and so on.
After looking at only a handful of values we can be confident it's not there.

> Data can be much easier to work with if it ordered in some sensible way.
> We want to learn to use tools that sort data.

Sorting Algorithms

There are many interesting sorting algorithms.

To see some antique algorithms, with 1981 computer graphics, I encourage you to watch Sorting Out Sorting on YouTube.

It's fun to think about sorting algorithms, and creating new ones is an active area of research.

By default, Python uses a fairly new (2002) algorithm, Timsort, by Tim Peters.

But we don't want to think about clever algorithms; we want to Do Science.
So we'll let people like Tim create clever algorithms, and we'll just use them.

Sorting tools: the `sorted` function

The built-in function `sorted` takes any iterable (a `list`, a `str`, or something else), and returns a new list containing the same values, in sorted order.

```
## A list:
sorted([2,4,6,0,1]) ⇒ [0, 1, 2, 4, 6]
## A tuple gets turned into a list:
sorted(('In', 'a', 'hole', 'in', 'the', 'ground', 'there', 'lived', 'a', 'hobbit'))
       ⇒ ['In', 'a', 'a', 'ground', 'hobbit', 'hole', 'in', 'lived', 'the', 'there']
## A str is an iterable containing single characters; it is turned into a list:
sorted("Gandalf") ⇒ ['G', 'a', 'a', 'd', 'f', 'l', 'n']
## (The uppercase letters come before lowercase.)
```

! It's important to note: `sorted` never mutates a list. It returns a **new list** with the values.

The `list.sort` method

On an existing list, we can call the `list.sort` method. This mutates the list and returns `None`.

```
jvj = [2, 4, 6, 0, 1]
jvj.sort() ⇒ None
jvj ⇒ [0, 1, 2, 4, 6]
gloin = ['He', 'looks', 'more', 'like', 'a', 'grocer', 'than', 'a', 'burglar!']
gloin.sort() ⇒ None
gloin ⇒ ['He', 'a', 'a', 'burglar!', 'grocer', 'like', 'looks', 'more', 'than']
```

You can only call the `list.sort` method on a list; it won't work on a `str`, `tuple`, etc, etc.

! It's important to note: `list.sort` always returns `None`. It only mutates the list!

What kinds of things can we sort?

Both tools, `sorted` and `list.sort`, work only when the items can be compared using the `<` operator (or other similar operators). If we try: `sorted([3, "Bilbo"])` we get an error:

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

Which is smaller: `3`, or `"Bilbo"`? It's not clear what that would mean, so it's an error.

But we can compare a lot of things using `<`.

We can compare two lists that contain comparable values: `[2,6,5] < [3]` ⇒ `True`.
So we can sort a `list[list[int]]`:

```
lol = [[3,7,4], [3,6], [1,8,5], [6], []]
sorted(lol) ⇒ [[], [1, 8, 5], [3, 6], [3, 7, 4], [6]]
```

Read The Fine Manual

Let's read the documentation:

```
>>> help(sorted)
Help on built-in function sorted in module builtins:

sorted(iterable, /, *, key=None, reverse=False)
    Return a new list containing all items from the iterable in ascending order.

    A custom key function can be supplied to customize the sort order, and the
    reverse flag can be set to request the result in descending order.
```

A **flag** is a parameter that is a `bool`.

If we set `reverse=True`, it will sort backwards:

```
sorted([2,4,6,0,1]) ⇒ [0, 1, 2, 4, 6]
sorted([2,4,6,0,1], reverse=True) ⇒ [6, 4, 2, 1, 0]
sorted("Gandalf") ⇒ ['G', 'a', 'a', 'd', 'f', 'l', 'n']
sorted("Gandalf", reverse=True) ⇒ ['n', 'l', 'f', 'd', 'a', 'a', 'G']
```

Some `str` methods

There are many methods defined on `str`.

```
s = "a man a plan a canal Panama"
```

- `str.split` finds whitespace in a `str`, and returns a new `list[str]` of the words:
  `s.split()` ⇒ `['a', 'man', 'a', 'plan', 'a', 'canal', 'Panama']`
- if we give it an argument, it splits on that instead of spaces:
  `s.split('n')` ⇒ `['a ma', ' a pla', ' a ca', 'al Pa', 'ama']`
- `str.join` takes a `list[str]`, and returns a `str`, joining the values from the list using the `str`.
  `'*'.join(['M', 'A', 'S', 'H'])` ⇒ `'M*A*S*H'`
  `''.join(['ma', 'ple', 'sy', 'rup'])` ⇒ `'maplesyrup'`
- `str.find` takes a `str`, and returns an `int` indicating **at what index** the argument appears in the original `str`, or `-1` if it does not appear:

Read `help(str)` to see more.

See that we **do not** call these like `str.split` and `str.join`.
We have a variable that **is** a `str`, and call the method on that variable.

Sort keys

The help says: “A custom key function can be supplied to customize the sort order.”

This means we can write something like `sorted(mylist, key=f)`, or `mylist.sort(key=f)`.

`key` is an optional named parameter that is a `callable`.

The system transforms each item using `key`, then puts the **original values** in order so that these **transformed values** are sorted. E.g.: sort a `list[str]` using `len` as the key:

```
sorted(['cabbage', 'pear', 'avocado',  'dulse', 'mango', 'banana'], key=len)
```

'cabbage'	'pear'	'avocado'	'dulse'	'mango'	'banana'
7	4	7	5	5	6

Rearrange so lengths are ordered:

'pear'	'dulse'	'mango'	'banana'	'cabbage'	'avocado'
4	5	5	6	7	7

⇒ `['pear', 'dulse', 'mango', 'banana', 'cabbage', 'avocado']`

Sort keys

Suppose we want to sort by **units digit**. For example, in 245, the units digit is 5; in 24601, the units digit is 1. Since $1 < 5$, once sorted 24601 should come somewhere before 245.

First step: write the key function.

Exercise

Write a function `units_digit(n: int) -> int`, that returns the units digit of `n`.

```
check.expect("UD245", units_digit(245), 5)
check.expect("UD24601", units_digit(24601), 1)
check.expect("UD42", units_digit(42), 2)
```

(Work with non-negative integers for now, or test negatives carefully.)

That's all we need. Now write:

```
sorted([42, 245, 12, 7, 24601], key=units_digit)
⇒ [24601, 42, 12, 245, 7]

## Or...
mylist = [42, 245, 12, 7, 24601]
mylist.sort(key=units_digit) ⇒ None
mylist ⇒ [24601, 42, 12, 245, 7]
```

Example: sorting `tuple`s

Here are some values:

```
harry = ("Potter, Harry", 1980, ["Elder Wand", "Resurrection Stone", "Invis. Cloak"])
hermione = ("Granger, Hermione", 1979, ["Time Turner"])
frodo = ("Baggins, Frodo", 2968, ["One Ring", "Sting"])
sam   = ("Gamgee, Samwise", 2980, [])
heroes = [harry, hermione, frodo, sam]
```

Exercise

Write a function `sort_by_item_count(characters: list[tuple[str, int, list[str]]])`. It mutates `characters`, so it is in increasing order by number of magical items.

For example, `sort_by_item_count(heroes)` mutates so that `heroes`

```
⇒ [('Gamgee, Samwise', 2980, []),
    ('Granger, Hermione', 1979, ['Time Turner']),
    ('Baggins, Frodo', 2968, ['One Ring', 'Sting']),
    ('Potter, Harry', 1980, ['Elder Wand', 'Resurrection Stone', 'Invis. Cloak'])]
```

Hint

Remember, you just need to write a helper function that turns `harry` into `3`, `hermione` into `1`, and so on. Then use `list.sort`.

Stable sorting

Earlier we used `key=len` to sort foods:

```
sorted(['cabbage', 'pear', 'avocado',  'dulse', 'mango', 'banana'], key=len)
⇒ ['pear', 'dulse', 'mango', 'banana', 'cabbage', 'avocado']
```

Q: why does `'dulse'` come before `'mango'`? Why does `'cabbage'` come before `'avocado'`?

A: our sorting is **stable**. That means that items that are "equal" stay **in the same order**.

Since `len('dulse')` and `len('mango')` are equal, they stay in the same order after sorting.

This means that if we first sort by "X", and then sort by "Y":

- "Y" will define the groups
- "X" will define the sorting within the groups.

```
half = sorted(['cabbage', 'pear', 'avocado',  'dulse', 'mango', 'banana'])
half ⇒ ['avocado', 'banana', 'cabbage', 'dulse', 'mango', 'pear']  # alphabetically
sorted(half, key=len)
⇒ ['pear', 'dulse', 'mango', 'banana', 'avocado', 'cabbage']       # by length
```

They are grouped by size, and each group is sorted alphabetically!

Stable sorting

The `str.count` method can be used to determine how many times a letter appears:

```
w = 'constitutionality'; w.count('t') ⇒ 4
w = 'floccinaucinihilipilifications'; w.count('i') ⇒ 9
```

Exercise

Write a function `sort_q_count` that takes a `list[str]` and returns a new list containing the same words, categorized by how many times the letter `'q'` appears, and sorted alphabetically within each category.

```
sort_q_count(['quote', 'dog', 'cat', 'albuquerque','saqqara','elephant',
              'quinquereme', 'unique', 'clique'])

⇒ ['cat', 'dog', 'elephant',
   'clique', 'quote', 'unique',
   'albuquerque', 'quinquereme', 'saqqara']
```

Associating values with keys

We have two ways to usefully extract information from lists:

1. Use a `for` loop to walk through the list, one item at a time;
2. use an **index** to extract the value in a particular location.

I want to store a certain amount of information, indexed by a key that might not be an `int`.

We want to associate a `str` describing a job with another `str` that is the name of the person in that job:

- `"President"` with `"Volodymyr Zelensky"`
- `"Minister of Defence"` with `"Rustem Umerov"`
- `"Minister for Veterans Affairs"` with `"Yulia Laputina"`

Lists of pairs?

We could do this using a list, storing a `tuple[str, str]` for each job:

```
govt = [("President", "Volodymyr Zelensky"),
        ("Minister of Defence", "Rustem Umerov"),
        ("Minister for Veterans Affairs", "Yulia Laputina")]
```

I can write code to make this work:

```
def look_up(job: str, data: list[tuple[str, str]]) -> str:
    """Find the name associated with job in data."""
    for row in data:
        if row[0] == job:
            return row[1]

look_up("President", govt) ⇒ "Volodymyr Zelensky"
```

This is OK, but inelegant. With more pairs in the list, it gets slower and slower.

! This is **not** a good way to store this kind of information.
There must be a better way, and there is: dictionaries.

Creating Dictionaries

A dictionary is a way to associate keys and values.

To **create** a dictionary: inside curly brackets, write `key: value` pairs, separated by commas.

For example:

```
govt = {
    "President": "Volodymyr Zelensky",
    "Minister of Defence": "Rustem Umerov",
    "Minister for Veterans Affairs": "Yulia Laputina"
}
```

The empty dictionary is expressed as `{}`.

Extracting Data from Dictionaries

There is some similarity to a list:

1. Get a single item by index, using a key as index:
   ```
   govt["President"] ⇒ "Volodymyr Zelensky"
   govt["Minister for Veterans Affairs"] ⇒ "Yulia Laputina"
   ```
2. Use a `for` loop to iterate through the **keys** only:
   ```
   for job in govt:
       print(job)
   ## This prints:
   President
   Minister of Defence
   Minister for Veterans Affairs
   ```

To see the associated `values`, use indexing on the keys:

```
for job in govt:
    print(govt[job])
```

```
## This prints:
Volodymyr Zelensky
Rustem Umerov
Yulia Laputina
```

Type annotations for Dictionaries

With a list, the values in the list could have any type, but the index was always an `int`.

Using a dictionary, the values can still be any type, but the **keys don't have to be ints**.

To describe the type of a dictionary we write `dict[KeyType, ValueType]`, where `KeyType` and `ValueType` represent the types of the keys and values.

Examples:

- `a = {3: 'trois', 4: 'quatre', 5: 'cinq'}` is a `dict[int, str]`. Use: `a[3]` ⇒ `'trois'`
- `b = {'trois': 3, 'quatre': 4, 'cinq': 5}` is a `dict[str, int]`. Use: `b['trois']` ⇒ `3`
- ```
  c = {(3, 4): 5.0,
       (1, 1): 1.4141,
       (2, 3): 3.606
  }
  ```
  Each key is a `tuple[int, int]`, and each value is a `float`.
  So this is a `dict[tuple[int, int], float]`. Use: `c[(2,3)]` ⇒ `3.606`
  ```

Key types

Many types can be used as keys, including `int`, but also `float`, and `str`. We can even use a `tuple` as a key, provided it contains only types that can themselves be used as keys.

Using only `int` and `str` as keys will be enough for the majority of our code. **Be aware that other types can be used.**

We can't use lists as keys; what can we use? Technically the only restriction is that the type be **hashable**. We're not going to go into what that means. If we want to say "the keys can be anything, as far as possible", we will use `any`, even though it's slightly imprecise.

Exercise

Consider:

```
crazydict = {math.cos: "cosine",
             math.sin: "sine",
             abs: "absolute value"
}
```

What type is `crazydict` ?

Keys and Values

Here is a example `dict[int, int]`:

```
data = {4: 41,
        9: 39,
        3: 32,
        2: 25}
```

We will write two functions that take such a value and return an `int`.

Exercise

Write a function `sum_keys` that returns the sum of the **keys**:

```
check.expect("SK", sum_keys(data),
             4 + 9 + 3 + 2)
```

Exercise

Write a function `sum_values` that returns the sum of the **values**:

```
check.expect("SV", sum_values(data),
             41 + 39 + 32 + 25)
```

Mutating dictionaries

Like lists, we can add, change, and remove items from a dictionary.
Suppose we have:
`a = {3: 'trois', 4: 'quatre', 5: 'cinq'}`.

- To **add** a new item, assign a new value using a new key as index:
  `a[24601] = 'Jean Valjean'`
  Now `a` is: `{3: 'trois', 4: 'quatre', 5: 'cinq', 24601: 'Jean Valjean'}`.
- To **change** an item, assign a new value, using a key that is already in the dictionary:
  `a[24601] = 'Monsieur Madeleine'`
  Now `a` is: `{3: 'trois', 4: 'quatre', 5: 'cinq', 24601: 'Monsieur Madeleine'}`.
- To remove a `key : value` pair, use the **`dict`**`.pop` method, using an existing key:
  `a.pop(4)` $\Rightarrow$ `'quatre'`
  Now `a` is: `{3: 'trois', 5: 'cinq', 24601: 'Monsieur Madeleine'}`.

Example: Reversing a dictionary

Exercise

Write a function `reverse_dictionary` that takes a `dict[int, str]` and returns a new `dict[str, int]` that has keys and values reversed.

For example:

```
reverse_dict({ 3: 'trois', 4: 'quatre', 5: 'cinq' })
⇒ { 'trois': 3, 'quatre': 4, 'cinq': 5 }
```

Ex.

Can you find a dictionary `d` such that `reverse_dict(reverse_dict(d)) != d` ?

Notice: it's impossible for a dictionary to have the same **key** more than once.

But the same **value** can appear as many times as you like:

```
nats = {
    0: 'zero',
    1: 'one',
    2: 'prime',
    3: 'prime',
    4: 'composite',
    5: 'prime',
    6: 'composite',
}
```

Sorting keys in a dictionary

When we introduced `sorted` on Slide 4, we said:

"The built-in function `sorted` takes **any iterable**".

Exercise

If we call `sorted` on a `dict[str, int]`,

```
mydict = {'trois': 3, 'quatre': 4, 'cinq': 5, 'six': 6}
sorted(mydict) ⇒ ?
```

Discuss: what do you expect it to return?

Sorting keys in a dictionary

When we iterate through a dictionary, we see the keys.

```
mydict = {'trois': 3, 'quatre': 4, 'cinq': 5, 'six': 6}
```

```
for item in mydict:
    print(item)
```

```
## We see:
trois
quatre
cinq
six
```

So the keys are the “iterable” part of a dictionary.

The function `sorted` always returns a `list[...]`, and here it will be list **containing the keys**.

```
sorted(mydict) ⇒ ['cinq', 'quatre', 'six', 'trois']
```

So we can loop through the keys in sorted order. We still get the values by indexing:

```
for item in sorted(mydict):
    print(item, mydict[item])
```

Checking if a value is a key

With a list, we can us the `in` operator to check if something appears:

```
4 in [2, 4, 6, 0, 1] ⇒ True
```

```
3 in [2, 4, 6, 0, 1] ⇒ False
```

We can do the same with a dictionary, but it only checks the keys:

```
nats = {
    0: 'zero',
    1: 'one',
    2: 'prime',
    3: 'prime',
    4: 'composite',
    5: 'prime',
    6: 'composite',
}
```

```
3 in nats ⇒ True
'prime' in nats ⇒ False
```

Exercise

Write a function `contains(d: dict[any, any], target: any) -> bool` that determines if any value in `d` is equal to `target`. E.g.

```
contains(nats, 'prime') ⇒ True
```

```
contains(nats, 'Optimus Prime') ⇒ False
```

Missing keys

It is an error to try to get the access the value using a key that is not in the dictionary:

```
d = { 3: 'trois', 4: 'quatre', 5: 'cinq' }

d[3] ⇒ 'trois'
d[4] ⇒ 'quatre'
d[5] ⇒ 'cinq'
d[6] # raises KeyError: 6
```

Example: a Histogram

To count how many times each value appears in an iterable, consider this function:

```
def histogram(s: str) -> dict[str, int]:
    """Return a dictionary that counts
    how often each character appears in s.
    """
    d = {}
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c] + 1
    return d

h = histogram('brontosaurus')
```

Exercise

Consider a trace of this call to `histogram`.

Exercise

Write a function `plot_histogram` that takes the value returned by `histogram`, and prints it, in alphabetic order, indicating how many of each item there are. E.g. `plot_histogram(h)` prints:

```
a
b
n
oo
rr
ss
t
uu
```

Module summary

- Use the built-in sorting tools (`sorted` and `list.sort`) to sort values, including the `reverse=` flag and `key=` parameter.
- Create and mutate dictionaries, describe their types using `dict[..., ...]`, iterate through them, check if items are present as keys.

Before we begin the next module:

- Read and complete the exercises in module 5 of the online textbook, at https://online.cs.uwaterloo.ca/
- Complete the module 5 Review Quiz, due on Monday.

Exercise

Write a function `divisors(n)`. It returns a list containing all the positive integers that `n` is divisible by. For example,

```
check.expect("D10", divisors(10), [1,2,5,10])
check.expect("D7", divisors(7), [1,7])
```

(Hint: do not use a dictionary for this exercise. It won't help.)

Exercise

Write a function `divisor_dict(n)`. It takes positive `int`, and returns a `dict[int, list[int]]`, where the keys are the numbers from `1` to `n`, and the associated value is a list containing all the numbers that divide that number.

Extra Practice

Recall the Collatz sequence: if n is a number in the sequence, the next number is given by $n/2$ when n is even, and by $3n+1$ when n is odd. Recall that this sequence always seems to reach 1.

Exercise

Write a function `collatz_dict(n)`. It returns a `dict[int,int]` where each key is a value in the Collatz sequence, and the value is the **next** value in the sequence. It should contain all the values encountered when starting at `n`. For example:

```
check.expect("C8", collatz_dict(8), {8:4, 4: 2, 2: 1})
check.expect("C3", collatz_dict(3),
             {3: 10, 10: 5, 5: 16, 16: 8, 8: 4, 4: 2, 2: 1})
```