collections: tuples, lists, dictionaries

❭ **go over solutions to homework 1**

❭ **collections**

- data types for storing multiple values (objects) together in a single object with a single name
- choosing the right way to store your data depends on what you want to do with it, and directly affects how efficient and readable your code will be. Choose wisely...

1. sequences - only integer indices allowed
    - tuples
    - lists
    - numpy arrays (next class)
2. mapping - allows for non-integer indices, or "keys", e.g. strings
    - dictionary
3. hybrid of sequence and mapping
    - Pandas `DataFrame` (class 09)

❭ **sequences: tuples and lists**

- tuples

    "A tuple is a finite ordered list of elements" -- Wikipedia

    - comes from words like "quadruple, quintuple, etc"
    - denoted by **parentheses** `()`, contain comma separated list of objects
    - can hold mix of any objects: integers, floats, strings, booleans, Dogs, Cats, whatever
    - by design, once declared, **cannot** be modified: "immutable"
    - e.g. `t = (1, 2, 3)` or `t = ('a', True, 3.14)`
        - parentheses are often optional: `t = 1, 2, 3`
        - tuple expansion/unpacking allows for multiple simultaneous assignment:
            - `a, b, c = (1, 2, 3)` or simply `a, b, c = 1, 2, 3`
        - tuples are often used to `return` multiple values from a function

            ```
            def mult123(x):
                return x, 2*x, 3*x


            a, b, c = mult123(2)
            ```

            - `return (x, 2*x, 3*x)` works just as well, but is more cluttered, takes extra typing, so less common
    - as with strings, get length of a tuple (or any other sequence) with the `len()` function
        - `len(t)` gives 3
    - indexing and slicing of tuples works as it does with strings:
        - `t[0]` gives 1
        - `t[-1]` gives 3
        - `t[:2]` gives `(1, 2)`
        - `t[::2]` gives `(1, 3)`

- - what happens if you try to assign to a particular entry in an existing tuple?
    - `t[0] = 4` - gives `TypeError: 'tuple' object does not support item assignment`
    - tuples are immutable!
  - methods:
    - `t.count(val)` returns number of occurrences of val
    - `t.index(val)` returns 0-based index of first occurence of val

- lists

  - denoted by **square brackets** `[]` , contain comma separated list of objects
  - can also hold mix of anything: integers, floats, strings, etc.
  - once declared, **can** be modified: "mutable"
  - e.g. `l = [1, 2, 3]` or `l = ['a', True, 3.14]`
  - initialize empty list with `l = []` or `l = list()`
  - same methods as tuple, plus these ones that can modify the list:
    - `l.append(val)`
    - `l.extend(anotherlist)` , or `l += [4, 5, 6]`
    - `l.reverse()`
    - `l.sort()`
      - does `.sort()` work for lists of objects of different types?
    - `l.clear()`
    - all the above methods operate *in place*, i.e. they modify the list, but don't return anything. This is different from string operations, that generally *don't* modify the string, but *do* return something, typically a new string
  - typical way to build a list is start with an empty one, use a `for` loop to `append` stuff to it:

  ```
  l = []
  for i in range(10):
      l.append(i)
  ```

  - if you just want a list of regularly spaced numbers, use range directly:
    - `l = list(range(10))`
  - convert a tuple to a list with the `list()` function
    - `list((1, 2, 3))`
  - convert a list to a tuple with the `tuple()` function
    - `tuple(l)`
  - indexing for lists is the same as for tuples and strings:
    - `l[0]` returns the first index, `l[n-1]` or `l[-1]` returns the last
  - delete entries from a list with `del` keyword by specifying the *index* of the entry to delete: `del l[2]`
  - slicing for lists is the same as for tuples and strings:
    - `l[::3]` gives every 3rd entry in the list, `l[::-3]` gives the reverse

- check contents of tuples and lists using `in` , same as for strings `3 in t` returns `True` , `5 in l` returns `False`

- when iterating over sequences, use `for val in sequence` , just like `for i in range(n)`

```
sequence = 5, True, 'blah'
for val in sequence:
    print(val)
```

- when iterating over a sequence using `enumerate()`, you get both the *index* and *value* of each entry, sometimes you might want to use both inside the loop

```
for i, val in enumerate(sequence):
    print(i, val)
```

gives:

```
0 5
1 True
2 blah
```

- use `zip()` to iterate over multiple sequences simultaneously:

```
for a, b in zip([1, 3, 5], [2, 4, 6]):
    print(a, b)
```

gives:

```
1 2
3 4
5 6
```

- `zip()` stops once it reaches the end of the shortest sequence
- **list comprehension**: handy for doing something simple but repetitive, without the extra lines and indentation of a normal for loop

  - build up a list in a single line of code:
  - `seqx2 = [ val*2 for val in sequence ]`

- common functions for use on sequences: `min()`, `max()`, `sum()`, `sorted()`, `tuple()`, `list()`

  - `sorted()` also works on strings

❯ sequences exercises:

1. Create a tuple with the following entries: `3, 5, 1.7, -2.7, 1e2, -50`
2. In a single line, make a new tuple that only contains every 2nd entry
3. Convert the original tuple in 1. to a list, assign it a name `l`
4. Sort the list in-place. Prove to yourself that it really is sorted. What happens if you sort it in-place again? What happens if you call `sorted()` on it?

5. Append the value `'blah'` to the list. What do you expect will happen if you try sorting it again? Try it!

6. Remove `'blah'` from the list, and sort the list in reverse order (multiple ways to do this)

7. Now make a new list by doubling the value of each entry in the tuple in 1. First do this using a normal `for` loop. Then redo it in a single line using list comprehension

8. Convert your code in 7. into a function called `multseq(seq, x)` that takes a sequence (tuple or list) `seq` and a multiplication factor `x` and returns a new list of `x` times the value of every entry. Ideally, the body of the function should only be a single line

› dictionaries

- what if you want to store and retrieve your values by name, instead of by numerical index?

    ○ e.g., you have an animal ID that is a mix of letters and numbers

- a "mapping" maps keys (names) to values

- dictionaries are the main mapping object in Python

    ○ denoted by **curly brackets** `{}` , contain comma separated list of key:value pairs
    ○ init an empty dictionary with `d = {}` or `d = dict()`
    ○ init a dict with some predefined key:value pairs:
    ○ `names2ages = {'Alice':25, 'Bob':20, 'Carol':32}`
    ○ keys don't have to be strings, they can be int, float, bool, etc. Same goes for values:
    ○ `ages2names = {25:'Alice', 20.4:'Bob', 32:'Carol'}`
    ○ as with lists and tuples, use square brackets `[]` to access an entry
    ○ access existing key:value pairs with `d[key]` , i.e. `names2ages['Alice']`
        ▪ what happens if key doesn't exist in d? `KeyError`
    ○ add new key:value pairs with `d[key] = value` , e.g. `d['a'] = 1`
        ▪ what happens if a key already exists? Its value is overwritten!
    ○ remove an existing key:value pair with `del d[key]`
        ▪ what happens if key doesn't exist in d? `KeyError`
    ○ dictionary methods
        ▪ `list(d)` or `list(d.keys())` returns a list of d's keys
        ▪ `list(d.values())` returns a list of d's values
        ▪ `list(d.items())` returns a list of tuples of d's `(key, value)` pairs
        ▪ `d[key].pop()` returns the value of `d[key]` and also removes the key and its val from d
    ○ iterating over dicts
        ▪ `for key in d:` or `for key in d.keys():`
        ▪ `for key, val in d.items():`
        ▪ `for val in d.values():`

- **dictionary comprehension**, analogous to list comprehension, typically need both key and val, so often need to iterate over `.items()` :

    ○ `dx2 = { key:2*val for (key, val) in d.items() }`

- NOTE: as of Python 3.6, order of `key:value` pairs in a dictionary is preserved - this means that the order of insertion is the same as the order of extraction

- - `print(names2ages)` will now always return `{'Alice':25, 'Bob':20, 'Carol':32}`, previously it was random (by design) and might return e.g. `{'Bob':20, 'Alice':25, 'Carol':32}`

- can combine tuples, lists, dicts in any combination, can be nested as deeply as you want

- common ones:

  - list of tuples: `[(1, 2), (3, 4), (5, 6)]`
  - dict of lists: `{'a':[1, 2, 3], 'b':[4, 5, 6]}`

› dictionaries exercises:

1. Describe this nested data structure in words: `[{'a':1, 'b':2}, {'c':3, 'd':4}]`
2. Assign the above structure to the name `l`. Index into `l` to print out only the second dictionary
3. Add a 3rd key:value pair `'e':5` to the second dictionary
4. Delete the key `'a'` from the first dictionary in `l`

› Gotcha: compare by reference vs. value

- for mutable sequences (like lists), be aware of difference between a "reference" and a copy:

1. `a = [1, 2, 3]; b = a`
   - `a` and `b` point (refer) to the same object in memory, the list `[1, 2, 3]`
2. `a = [1, 2, 3]; b = a.copy()`
   - `a` and `b` have the same value, but point to different objects in memory that happen to have the same value

- if we set `b[2] = 666`, what's the value of `a` in the above two cases?
- `is` and `is not` operators vs. `==` and `!=`
  - `a = [1, 2, 3]; b = a.copy()`
  - `a == [1, 2, 3]` returns True
  - `b == [1, 2, 3]` returns True
  - `a is b` returns False
  - `a is [1, 2, 3]` also returns False
  - `is` and `is not` operators check for identity, i.e., whether two variables point to the same object stored in memory
  - `==` and `!=` check for value, i.e. whether two variables have the same value
  - generally, it's safer and less confusing to use `==` than `is`, but good to know the difference

› Homework 2! Due before next class (class 04) on May 31