6.0001 Recitation 3 - Spring 2020

Friday, February 21st

I. Administrivia

Pset 1 checkoff due 2/24 @ 9pm
Pset 2 checkoff ongoing, due 3/12 @9pm
Pset 3 checkoff due 2/26 @ 9pm
Microquiz 2 on Monday 2/24, during lecture

II. Data structures

- **Immutable data types:** cannot change element value after assignment
  - Examples of immutable data types we've seen:
    - int
    - float
    - bool
    - string
    - (NEW) tuple
- **Mutable data types:** can change element after assignment
  - We can think of mutable objects as being assigned to a certain place in memory. In this case, assigning a variable to a mutable object just means that it points to that object in memory.
  - Multiple variables can point to the same object in memory. This can be problematic because mutating a variable will affect the other variables that point to it. This is called aliasing, more on this later.
  - Examples of immutable data types we've seen:
    - (NEW) lists
    - (NEW) dictionaries

III. Tuples

- These are ordered sequences of objects. These objects can be of any type.
- immutable, i.e cannot be changed once created
- can be indexed
- you can slice a tuple giving you a subset of the original tuple

```
tuple1 = (1, 2, 3, 4)
len(tuple1)  # gives you the length of the tuple
tuple1[0:2] # gives (1,2)
```

IV. Lists

- ordered sequence of objects
- can be indexed & sliced similarly to tuples
- mutable, i.e. can be changed/modified after being created
  - For example given the two lists:
    ```
    list1 = [1,2,3, “MIT”]
    list2 = [4,5,6]
    ```
  - You can change the element at index 0 with
    ```
    list1[0] = 5
    ```
  - add an element to the end
    ```
    list1.append(5)
    ```
  - add all elements of `list2` to the end of `list1` with
    ```
    list1.extend(list2)
    ```
  - remove an element at specific index with
    ```
    del list1[index]
    ```
  - remove element at the end
    ```
    list1.pop()
    ```
  - remove a specific element with
    ```
    list1.remove(“MIT”)
    ```
    - note that if an element appears multiple times, this method will only remove the first occurance of that element
    - if the element is not present, throws error

V. Dictionaries

- Map keys to values
- Keys:
  - Must be immutable
  - Must be unique
  - Ordering is not guaranteed
  - `dict.keys()`
- Values:
  - Don’t need to be immutable or unique
  - `dict.values()`
- `.get(key, default)`
  - tries to get value associated with key, with a `default` “fallback”
  - The default of `default` is `None`
- Iterating over a dictionary iterates over the keys
- Using `in` tests for membership amongst keys
- always check `in dict`, not `in dict.keys()` for efficiency reasons

VI. Mutability

- Mutable objects can be changed after they are created
- What mutable types do we know?
- **Aliasing**: When two variable *names* refer to the same object
  - Example:

    ```
    a = [1,2,3,4]
    b = a
    ```

  - **Now b points to a. Since a list is mutable, if you make changes to b, you will change a.**
- **Cloning:** Making a copy (always a safer option!)
  - “But I can change strings after they’re assigned!”

    ```
    word = “the”
    aliased_word = word
    word += “ bird”
    print (word) # “the bird”
    print (aliased_word) # “the”
    ```

  - **Not actually changing the string**. += is the same as creating a new variable:
    - `word = word + “ bird”`
    - Why? Strings are immutable
- For immutable types: creates a new object instead of modifying the original one
- For mutable types: new name refers to same object
- Why does mutability matter?
  - Makes your code do unexpected things (more examples in code)

- How can I avoid mutability problems?
  - Make clones, or copies
  - Use temporary variables
- {code example} Don’t change lists while iterating over them!!
- {code example} `sort` vs `sorted`
  - `sort`: mutate the list, return nothing
  - `sorted`: doesn’t mutate the list, return a new sorted list

VII. Debugging Tips

- Print out the values of your variables
- Google is your friend if you encounter an error you don’t understand.
- The stack trace shows what line(s) caused the error -- use it!

- Using **assertions**:

  ```
  assert <boolean condition>
  ```

```
assert <boolean condition>, <argument>
```

- **Exceptions (and how to handle them)**
  - Exceptions occur when the syntax is correct but the code performs some operation that isn't allowed
    - `int(‘1.1’)`
    - trying to divide by zero.
  - Exceptions are better than letting the program silently fail
- **Terminology**
  - `raise` : you **raise** (or **throw**) an exception when you want an exception to occur
  - `try/except` : you **handle** (or **catch**) an exception when you want to do something (and not have the program crash) in the case you encounter an exception
  - an unhandled exception will cause a **Traceback** (or **stack trace**) to be printed to the interactive shell (in IDLE it's printed in red)

- **Exception Handling**
  - **`try/except:`** allows you to handle exceptions in your code
    - If you say `except` without specifying a specific exception, then it handles ALL exceptions that occur in the try block.

      ```
      try:
          ...
      except:
          ...
      ```

    - If a `ValueError` (or a subclass of `ValueError`) is raised (i.e. happens) within the `try` clause, then whatever is in the `try` clause after the erroring line is not executed, and the program jumps to the `except` clause:

      ```
      try:
          ...
      except ValueError:
          …
      ```

  - **`raise:`** `allows you to raise an error in your code`
    - You can raise a `ValueError` in your code by saying:

      ```
      raise ValueError
      ```