6.189 – Notes
Session 8

Day 6: Immutable Objects

Earlier, we made a big deal about the fact that lists are mutable. The reason this is important is because certain objects are **immutable** – once created, their value *cannot* be changed.

Strings are a prime example of this. Although we treated strings the same as primitives like integers and booleans earlier, strings are actually objects.

Why did we do this? Think about this: if an object is immutable, it doesn't matter whether two variables are pointing to the same string or two different strings with the same value! Thus, while strings are actually immutable objects, we can treat them as we have before – as primitives. The only new meaning this revelation has is that like lists, strings have member functions.

For strings (and tuples, when we get to them), its easiest to think of them like primitives – directly stored in the variable table.

Day 6: Strings Revisited

Most of the member functions in lists modified the list and had no return value. Strings are immutable, though – how do string member functions work? It turns out that member functions of strings tend to *return* a new string.

Program Text:

```
message = "Hello"
print message
message.lower() #no effect
print message
message = message.lower()
print message
```

Output:

```
Hello
Hello
hello
```

Note: `lower()` is a function that converts a string into lowercase.

Here is a list of some useful string functions. *Don't try to memorize these!* Even I don't remember them – instead, when I need to look up a function I go to the Python Quick Reference website shown in class (and on the website.)

A quick reminder before starting: remember that `"A"` and `"a"` are completely different characters! When writing functions that manipulate strings, its generally a good idea to deal with a single case (usually lowercase).

Functions that return a new string

- `str.capitalize()` / `str.lower()`. Returns a copy of `str` with all letters converted to uppercase / lowercase.
- `str.strip()`. Returns a copy of `str` with all **whitespace** (spaces/tabs/newlines) from the beginning and end of the string removed.
  Example: `"  test  ".strip() == "test"`.
- `str.replace(old,new)`. Returns a copy of `str` with all instances of `old` within the string replaced with new.
  Example: `"hallo all!".replace("al", "el") == "hello ell!"`.

Functions which return information about a string

- `str.count(substring).` Returns the number of times `substring` appears within `str`.
- `str.find(substring)` / `str.rfind(substring).` Returns the position of the *first* instance of `substring` within `str`. rfind returns the position of the *last* instance of `substring`.
- s`.startswith(substring)` / `str.endswith(substring)`. Returns `True` if the string starts with / ends with substring.
  Example: `"Hello".startswith("he") == False`, but `"Hello".endswith("lo") == True`

Functions which transform the string into other types

- `str.split(separator)`. Returns a *list* of words in str, using separator as the delimiter string.
  Example: `"hello world, Mihir here".split(" ")` returns `["hello","world,","Mihir","here"]`.
  Example: `"mississippi".split("s")` returns `["mi", "", "i", "", "ippi"]`.
- `separator.join(seq)`. This one is tricky. It takes a *list* of strings `seq` and combines them into a string. Each element in `seq` is separated by `separator` in the returned string.
  Example: `" ".join(["hello","world"]) == "hello world"`

Day 4: Tuples

Tuples are the immutable counterpart of lists. Unlike a list, tuples cannot be changed.

Why/where are tuples useful? Think of a tuple as multi-dimensional data -- just like you can store an integer 5 in a variable, you can also store a two-dimensional coordinate `(6,-3)`. You'll develop an instinct for when to use tuples versus lists as you continue in course 6 – just remember that it tends to be much easier to use tuples whenever you can get away with it.

You can create tuples by using parentheses: `(1,3,8)` creates the tuple with elements 1, 3, 8. As you should expect, tuples are ordered: `(1,3) != (3,1)`

If you want to create a singleton tuple (a tuple with one element), you can use `tuple(5)` or use the notation `(5,)`. Note that `(5,) != 5` – they have completely different types (the former is a tuple and the latter is an integer.

Also note that `(4,6) != [4,6]` – one is a tuple and the other is a list. Similarly, `("a","b") != "ab"`. You *can* convert between these three formats using `str(x)`, `tuple(x)`, and `list(x)`, though.

You can nest tuples in tuples! Note that `((1,2),3) != (1,2,3)` – the former is a tuple that contains two elements (one tuple and one integer), whereas the latter is a tuple that contains three elements.

Notation for using tuples will be covered in the next section.

Day ?: Sequence notation

Lists, strings and tuples are all examples of sequences – a series of ordered items. In the case of strings, you can think of them as a sequence of characters.

Sequence isn't an official term or anything – just an observation that all three of these are very similar. In fact, they share much of the same syntax.

- **Indexing**. `seq[i]` will return the item (or character) at the `i`th position.
- **Length**. `len(seq)` returns the length of a sequence.
- **Slicing**. You can slice sequences the same way you sliced lists. `"hello"[0:3] == "hel"`
- `in`, `not in` **operators**. `x in seq` is `True` if and only if an item of `seq` is equal to `x`. For strings, you can check if a substring is in a string, e.g. `"ello" in "hello"` returns `True`.
- **Concatenation**. You can use the plus operator to combine two sequences of the same type. You can use * to duplicate it `n` times for some integer `n`, e.g. `"Yay! " * 5`
- **For loops**. We learned that the `for` operator works on lists. `for` actually works on any sequence – for strings, it iterates through each character in the string.

Day 7: Dictionaries

And so we come to the last major topic of this class: **dictionaries**. Like lists, dictionaries are an mutable object that we can use to store data. Where lists stored a sequence of items, dictionaries store a table.

key	value
`"a"`	`5`
`"test"`	`[1,2]`
`27`	`"test"`

Note that dictionaries are considered to be **unordered** – it doesn't matter what order we list entries in. We call the names on the left **keys** and the values on the right **values**.

You can store primitives and *immutable* objects (like strings and tuples) as keys. You can store anything (e.g. lists) as a value.

As we go through the dictionary notation, notice that a lot of it is consistent with the sequence notation above – the notation below should seem intuitive to you. Remember that dictionaries are fundamentally different from sequences, though – especially the fact that dictionaries are unordered.

- You can create a new dictionary using curly braces.
  Example: `example_dict = {}` creates an empty dictionary
  Example: `example_dict = {"a":5, "test":[1,2], 27:"Test"}` creates the above table.
  The amount of spacing around the colon `:` is irrelevant.
- `len(d)` works on dictionaries too - use it to find the number of entries in the dictionary (the above dictionary has length 3.)
- To access or change a value, use the same index notation.
  Example: `print example_dict["a"]` prints 5
  Example: `example_dict["a"] = 7`
  Note that this implies that a dictionary cannot contain two identical keys – writing `example_dict["a"] = 7` would just change the value that `a` is mapped to. This should make sense, though – remember that *dictionaries are unordered*. Also, you *can* have a dictionary with identical values.
- Use `del example_dict["a"]` to remove that entry from the dictionary.
- k in d will return true if the dictionary d contains an entry with key k.
  Example: `(27 in example_dict) == True`
  Example: `(5 in example_dict) == False`

Day 7: Dictionary Member functions

Like other objects, dictionaries have member functions. You don't really need to use these much, though – here are a few that might be useful

- `d.clear()`. Removes *all* items from `d`.
- `d.copy()`. Returns a *copy* of the dictionary `d`.
- `d.pop(k)`. Removes the entry with key `k` and returns its corresponding value. This is just like `del d[k]`, except that the function also *returns* the value of `d[k]`.

Day 7: For loops and dictionaries

Remember how `k in d` will return `True` if k is mapped to something? You can also use `for` loops with dictionaries. For loops will iterate over all the **keys** in the dictionary:

Program Text:

```
example_dict = {"a" : 5, "b" : True}
for k in example_dict:
  print k, ";", example_dict[k]
```

Output:

```
a ; 5
b ; True
```