Dictionaries

Outline

Dictionaries

- This week we'll work with a new type: `dict`.
- A Python `dict`, or "dictionary", is a kind of collection type
- Where lists map numerical indices to values
  - And indices are *dense*...
- Dictionaries map arbitrary keys to values
  - And keys are *sparse*

Creating Dictionaries

We have two ways to create a `dict`:

- `d = dict()` for an empty `dict`
- `d = {"a": 1, "b": 2}` for a `dict` literal.

Indexing Dictionaries

Indexing is implemented for dicts, like for lists:

```
d = dict()
d["a"] = 1
d["b"] = 2
d["c"] = d["a"] + d["b"]
```

Iterating Through Dictionaries

Also like lists, we can iterate through dictionaries; unlike lists, this gives us a series of indexes.

```
d = {1:"good morning", 3:"good afternoon", 5:"good nigh
for key in d:
    num = key + 2
    print(key, d[key][0:num])
```

What will this print?

Iterating Through Keys and Values

We could rewrite the preceding example:

```
d = {1:"good morning", 3:"good afternoon", 5:"good nigh
for key, value in d.items():
    num = key + 2
    print(key, value[0:num])
```

Exercise: Menu Prices

Imagine a dictionary `menu` which maps food names ("veggie burger", "gazpacho", etc) to their prices in dollars.
Write a function `compute_total(menu, items)` where `items` is a list of food names; the function should return the total cost of the given items according to the menu.

Exercise: Priciest Item

Write a function `most_expensive(menu)` that returns the most expensive item on the menu.

Keys and Values

Sometimes we only want to work with the keys or the values of a dictionary, and don't need to iterate through both at once. Dict's methods can help:

```
#list() converts any iterator to a list
alphabetized = list(menu.keys())
alphabetized.sort()
```

Dictionaries are Mutable

- Dictionaries, like lists, are *mutable*
- Be careful that your functions don't change their inputs by mistake
- This is a bug minefield

Revenge of the Variables

What will the values of `x` and `y` be after this code executes?

```
x = {"a":1, "b":2}
y = x
y["c"] = 3
```

Aliasing & Copying

- Aliasing is back with a vengeance
- Remember: Assignment is still the only thing that can change what a variable refers to
- BUT! Lists and dictionaries have internal mutability, so their *contents* can change.
- In the example above, `x` and `y` end up pointing to the *same* dict.

Copying

We can avoid aliasing with lists by slicing. But it turns out both lists and dicts have a copy method:

```
x = {"a":1, "b":2}
y = x.copy() # !!
y["c"] = 3
```

In this case, y points to a copy of x, while x still points to its original contents.

Hygiene

When a function receives a dict as a parameter, it should document whether:

1. It stores a reference (variable) pointing to this dict
2. It may modify this dict
3. The caller is allowed to use the dict anymore afterwards

When a function returns a dict, it should document whether:

1. The caller may modify the dict
2. The dict may be modified or aliased by some other code

When in doubt, arguments should be copied to be stored and return values that are stored elsewhere should be copied before returning them.

del

We can remove an element from a dictionary using the `del` statement:

```
x = {"apple": "pie", "banana":"pie", "chocolate":"ice c
del x["banana"]
```

in

You can use `in` to check if a key is present in a dictionary:

```
x = {"apple":"pie", "chocolate":"ice cream"}
if "banana" in x:
    print("found banana")
```

Dicts are Objects

Like strings and lists, dictionaries have a suite of methods to manipulate them; here are just a couple:

- `dict.get(key, default=None)`
  - Obtains `key` from `dict`, returning `default` if the key is absent (`default` is optional)
- `dict.keys()`, `dict.values()` return iterators

Exercise

Write a function `dict_subset(d1, d2)` which returns True if all of `d1`'s keys are defined in `d2`.

One more type: `set`

`dict` has the nice property that every key has at most one value; we could imagine using them to track e.g. which words have appeared in a document using a dict from words (strings) to bools. But then, if a value is False or if it's absent mean basically the same thing.

A mathematical *set* is a collection of values where each possible value appears at most once. Python has a `set` type as well which is similar in spirit to `dict`, but you use the `add` method to push items into the set and `remove` to get rid of an item. It can be nicer to work with than a dict if you really just care about *whether* a key is present.

Quiz

Assignment 6

We'll talk about the Naive Bayes classifier for this assignment next time, but it's actually ok to start on the assignment today—you have all the technical skill you need for it.
Just remember the "test as you go" idea from last week. You'll want to use that concept here as well. The writeup encourages you to build your code a piece at a time; and each piece can be tested independently of the others.