Module 9

Topics:

•Dictionaries

•Classes

Readings: ThinkP 11, 15, 16, 17

Collections of key-value pairs

- Recall association lists from CS115, collections of key-value pairs, where
  - Key: describes something basic and unique about an object (e.g. student ID, SIN, cell's DNA signature)
  - Value: a property of that object (e.g. student's major, person name, type of organism)
- Key-value pairs are basic to computer applications:
  - Looking up someone in an online phonebook
  - Logging onto a server with your userid and password
  - Opening up a document by specifying its name

Dictionaries, or key-value collections

- Built into Python
- Use **{}** for dictionaries
- Very fast
  - essentially O(1) to look up a value in a dictionary
- The type used for the key must be immutable (e.g. strings, int)
- Any type can be used for the value
- The dictionary is not ordered

Creating Dictionaries

- Create a dictionary by listing multiple **key:value** pairs

```
wavelengths = {'blue': 400,
   'green': 500, 'yellow':600,
   'red':700}
```

- Create an empty dictionary

```
students = {}
```


Using a dictionary

- Retrieve a value by using its key as an index

```
wavelengths['blue'] => 400
students[2001] => KeyError:2001
```

- Update a value by using its key as an index

```
wavelengths['red'] = 720
```

- Add a value by using its key as an index

```
wavelengths['orange'] = 630
```


Dictionary methods and functions

Module is called **dict**

- **len(d)=>** number of pairs in **d**
- **d.has_key(k) => True** if **k** is in **d**
- **d.keys() =>** list of keys in **d**
- **d.values() =>** list of values in **d**
- **d.pop(k) => value** for **k**, and removes **k:value** from **d**
- See **dir(dict)** for more

Specifying a dictionary's type

Since we have both keys and values, both must be specified:

```
(dictof key_type value_type)
```

Example: **wavelengths** is of type

```
(dictof str int[>0])
```


When to use dictionaries

- Generally faster to look up keys in a dictionary than in a list
- Only use dictionaries if the order is not important
  - If order is important , use a list instead
- Very useful when counting number of times an item occurs in a collection (e.g. characters or words in a document)

Example: Counting number of distinct characters in a string

```
## distinct_characters:
##    str -> int[>=0]
def distinct_characters (s):
    characters = {}
    for char in s:
        characters[char] = True
    return len(characters)
```

Instead, count number of times each character occurs

```
## character_count: str
          -> (dictof str[len=1] int[>0])
def character_count (sentence):
    characters = {}
    for char in sentence:
        if characters.has_key (char):
            characters[char] =
                 characters[char] + 1
        else:
            characters[char] = 1
    return characters
```


Next, find the most common character in a string

```
## most_common_character: str[non-empty]
##    -> str[len=1]
def most_common_character (sentence):
    chars = character_count(sentence)
    diff_chars = chars.keys()
    most_common = diff_chars[0]
    max_times = chars[most_common]

    for curr_char in diff_chars[1:]:
      if chars[curr_char] > max_times:
         most_common = curr_char
         max_times = chars[curr_char]
    return most_common
```


"Usual" run-time for important dictionary operations

Assume dictionary **d** contains n keys, including **k**

- **d[k]** is usually O(1)
- **d.keys()** is O(n)
- **d.values()** is O(n)
- **d.has_key(k)** is usually O(1)
- **k in d** is O(n)

Exercise

Write a Python function **`common_keys`** that consumes two dictionaries, and produces a list of all keys which occur in both dictionaries.

Recall: Structures in Scheme

To declare a new structure in Scheme:

```
(define-struct Country
  (continent leader population))
;; A Country is a structure
;; (make-Country c l p), where
;; c is a string (for country's
;; continent), l is a string (for
;; the name of the country's leader),
;; and p is a nat (for the population)
```


Classes: like structures (but with more)

To declare a similar thing in Python:

```
class Country:
  'Fields: continent, leader,
  population'
```

Using classes

- Python includes a very basic set-up for classes
- We will include several very important methods in our classes to help with
  - Creating objects
  - Comparing objects
  - Printing objects
- These methods will use the local name **`self`** to refer to the object being used

Constructing objects with `__init__`

```
class Country:
 'Fields: continent, leader, population'
 def __init__(self, continent,
  leader, population):
    self.continent = continent
    self.leader = leader
    self.population = population
```

To create a `Country` object:

```
canada = Country("North America",
  "Harper", 34500000)
```


Accessing the fields of an object

```
india = Country("Asia", "Singh",
        1170938000)
print india.continent
print india.leader == 'Singh'
india.population += 1
```

Aliasing is again an issue

```
india_alias = india
india_alias.population += 1
```

The population of both **india** and **india_alias** is increased (since there is only one **Country** object here)

What if you want another copy of an object, rather than an alias?

Two approaches:

- Create a new object, and set all the fields

```
india_copy = Country (india.continent,
  india.leader, india.population)
```

- Use the module **copy**, with the function **copy** or **deepcopy**

```
import copy
india_copy2 = copy.copy(india)
india_copy2.leader = 'Nehru'
## value of india.leader is still 'Singh'
```


__ repr __ : Very helpful for debugging

```
>>> print canada
<__main__.Country instance at 0x0286EC10>
```

However, including the following

```
class Country:
  # __init__ code not included ...
  def __repr__(self):
      s1 = "CNT: " + self.continent
      s2 = "; L: " + self.leader
      s3 = "; POP: " + str(self.population)
      return s1 + s2 + s3
```

makes things much better!

```
>>> print canada
CNT: North America; L: Harper; POP: 34500000
```

`__ eq __` : specifying object equality

For objects `x,y,` `x==y` → `True` only if `x` and `y` are aliases

If we want `x==y` => `True` if the corresponding fields are equal, we can specify this by providing a function called `__ eq __`

```
class Country:
  def __eq__(self, other):
     return type(self) == type(other) \
       and self.continent==other.continent\
       and self.leader==other.leader \
       and self.population==other.population
```


`__ ne __` : specifying object inequality

- `check.expect` actually checks for inequalities, so `__ ne __` is needed as well

```
class Country:
  def __ne__(self, other):
     return not(self==other)
```


Exercise: Write a function that produces `Country` with higher population

```
def higher_population(c1, c2):
      if c1.population >= c2.population:
            return c1
      else:
            return c2
canada = Country("North America", "Harper",
                 34108752)
us = Country("North America", 'Obama', 307006550)
## Test 1: second country has higher population
check.expect("T1", higher_population(canada, us),
             us)
```

Exercise: Determine total population for a list of countries

```
## total_population:
##   (listof Country) ->
##       int[>=0]
## Produces total population in
## countries
def total_population(countries):
```


There's a lot more to Python classes

- Use **`dir(c)`** to see available methods and fields, where **`c`** is object or the type name
- Classes join a related set of values into a single compound object (like Scheme structures)
- With classes, we can attach methods to types of objects (like for **`str`**, **`list`**, **`dict`**)
  - *not officially part of CS116 – but very interesting!*

Object-oriented design

- Classes are used to associate methods with the objects they work on
- Classes and modules allow programmers to divide a large project into smaller parts
- Different people can work on different parts
- Managing this division (and putting the pieces back together) is a key part of software engineering
- See CS246 or CS432 to learn more

Goals of Module 9

- Use dictionaries to associate keys and values for extremely fast lookup
- Be able to define a class to group related information into a single compound object