Module 09: Additional Options for Organizing Data

Topics:
• Dictionaries
• Classes

Readings: ThinkP 11, 15, 16, 17
Collections of key-value pairs

• In CS115, you studied 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:
  – 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 – key retrieval is *essentially* $O(1)$
• The type used for the key must be immutable (e.g. Str, Int)
• Any type can be used for the value
• Keys are not sorted or ordered
• No reverse look-up by value (brute-force only)
Creating Dictionaries

• Create a dictionary by listing multiple key:value pairs

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

• Create an empty dictionary

```python
students = {}
```
Using a dictionary

• Retrieve a value by using its key as an index
  \[\text{wavelengths['blue']} \Rightarrow 400\]
  \[\text{students[2001]} \Rightarrow \text{KeyError:2001}\]

• Update a value by using its key as an index
  \[\text{wavelengths['red']} = 720\]

• Add a value by using its key as an index
  \[\text{wavelengths['orange']} = 630\]
Dictionary methods and functions

Module is called `dict`

- `len(d)` => number of pairs in `d`
- `d.keys()` => a view of keys in `d`
- `d.values()` => a view of values in `d`
  - Views can be used in for loops
- `k in d` => `True` if `k` is a key in `d`
- `d.pop(k)` => value for `k`, and removes `k:value` from `d`
- See `dir(dict)` for more
- Automatically imported in your program
Specifying a dictionary’s type

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

\[
\text{(dictof Key\_type Value\_type)}
\]

Example: \text{wavelengths} is of type \b (dictof Str Nat) \n
requires: keys are nonempty strings 
Each value > 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 times distinct characters occur in a string

def character_count(sentence):
    "character_count: Str->(dictof Str Nat)"

    characters = {}
    for char in sentence:
        if char in characters:
            characters[char] = \n            characters[char] + 1
        else:
            characters[char] = 1
    return characters
Next, find the most common character in a string

```python
def most_common_character(sentence):
    '''most_common_character: Str -> Str
    requires: len(sentence) > 0'''

    chars = character_count(sentence)
    most_common = ""
    max_times = 0

    for curr_char in chars:
        if chars[curr_char] > max_times:
            most_common = curr_char
            max_times = chars[curr_char]

    return most_common
```
Run-time basics for important dictionary operations

For a dictionary $d$ contains $n$ keys, *assume* the following runtimes:

- $d[k]$ is $O(1)$
- $d[k] = v$ is $O(1)$
- $k$ in $d$ is $O(1)$
- $\text{list}(d.\text{keys}())$ is $O(n)$
- $\text{list}(d.\text{values}())$ is $O(n)$

Note: the dictionary runtimes are more complicated than this, but we will work with these assumptions.
Exercise

Write a Python function `common_keys` that consumes two dictionaries with a common key type, and returns a list of all keys which occur in both dictionaries.
Dictionaries are mutable

• Dictionaries can be mutated:
  – Key:Value pairs added
  – Key:Value pairs deleted
  – Values updated for a particular Key

• Like lists, dictionaries can have aliases as well.
  Note that the following mutates `d1`.

```python
d1 = {3:'three', 2:'two'}
d2 = d1

d2[1] = 'one'
```
A function can mutate a dictionary too

```python
def purge(d):
    keys = list(d.keys())
    for k in keys:
        if d[k] == "":
            d.pop(k)
```

Suppose

```python
dt = {2: 'xx', 1: 'x', 0: '', 4: 'xxxx', -3: '', 3: 'xxx'}
```

what is the value of `dt` after calling `purge(dt)`?
Recall: Structures in Racket

To declare a new structure in Racket:

\[
(\text{define-struct Country} \\
  \text{(continent leader population)})
\]

;;; A Country is a

;;; (make-Country Str Str Str Nat)
Classes: like structures (but different)

To declare a similar thing in Python:

class Country:
    '''Fields: continent (Str),
    leader (Str),
    population (Nat)'''
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
  – Printing objects
  – Comparing objects
• These methods will use the local name `self` to refer to the object being used
Constructing objects with __init__

class Country:
    '''Fields: continent (Str), leader (Str),
    population (Nat)'''
    def __init__(self, cont, lead, pop):
        self.continent = cont
        self.leader = lead
        self.population = pop

To create a Country object:
 canada = Country("North America", "Trudeau", 35344962)
Memory model for classes

canada = Country("North America", "Trudeau", 35344962)
Accessing the fields of an object

```python
india = Country("Asia", "Modi", 1241491960)
print (india.continent)
print (india.leader == "Modi")
india.population += 1
```
__repr__ : Very helpful for debugging

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

However, including the following

class Country:

    # __init__ code ...

def __repr__(self):
    return "CNT: {0.continent}; L: {0.leader}; POP: {0.population}".format(self)

makes things much better!

```python
>>> print(canada)
CNT: North America; L: Trudeau; POP: 34500000
```
Comment on `__repr__`

- In practice, most Python programmers use `__str__` instead of `__repr__`
- The functions play very similar roles, but, for what we do in CS116, `__repr__` is a more convenient, so is used instead.
Aliases

```
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?

• Create a new object, and set all the fields:

```python
india_copy = Country
            (india.continent, india.leader, india.population)
```
\[ r = \text{Country}("A", "B", 10) \]
\[ s = r \]
\[ t = \text{Country}("A", "B", 10) \]
Comparing objects for equality

• Are two objects actually aliases? Use `is`
  
  `india_alias is india` → True
  
  `india_copy is india` → False

• Are the fields of two objects equal?
  
  – Would like
    
    • `india_copy == india` → True
  
  – But, that is not the default in Python

  – We need to provide another function first
__eq__ : specifying object equality

For objects \( x, y \), \( x == y \rightarrow True \)
only if \( x \) and \( y \) are aliases

If we want \( x == y \Rightarrow True \) if the corresponding fields are equal,
we can specify this by providing a function called __eq__

class Country:
    # __init__ and __str__ code ...
    def __eq__(self, other):
        return isinstance(other, Country) and
        self.continent == other.continent and\
        self.leader == other.leader and\
        self.population == other.population
Exercise: Write a function that returns the Country with higher population

```python
def higher_population(c1, c2):
    if c1.population >= c2.population:
        return c1
    else:
        return c2

canada = Country("North America", "Trudeau", 34108752)
us = Country("North America", 'Obama', 311591917)

## Test 1: second country has higher population
check.expect("T1", higher_population(canada, us), us)
```
Exercise

Write a function `leader_most_populous` that consumes a nonempty list of `Country` objects, and returns the leader of the most populous country in the list.
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 Racket structures)
• With classes, we can attach methods to types of objects (like for `str`, `list`, `dict`)
Class Methods

• Functions defined within the class (should be indented the same as `__init__`)
• First parameter is always `self`:
  – The function can mutate the fields of `self`
  – The function can use the fields of `self` in calculations and comparisons.
• Class methods are called using the same dot notation as the string and list methods.
• Class methods are like other functions. They may
  – Return values (or not)
  – Print information (or not)
Example **Country** class method:

```python
# Must be indented same amount as __init__
def election(self, winner):
    print("Election Results:")
    if self.leader == winner:
        print("{0} re-elected".format(self.leader))
    else:
        print("{0} replaces {1} as leader".format(winner, self.leader))
    self.leader = winner
```
Using `election`

```python
>>> us = Country("North America", "Obama", 307006550)
>>> us.election("Trump")
Election Results:
Trump replaces Obama as leader
>>> us.leader
Trump
```
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 CS430 to learn more
Goals of Module 09

- 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
- Be able to write class methods as well as other functions that use class objects