Advanced Data Structures: Sets and Dictionaries

CSE 231, Rich Enbody

More Data Structures

- List
- Tuple
- Dictionary
- Set
The dictionary is an important, very useful part of Python, as well as generally useful.
A dictionary is an *associative array* (list).

It as a list of *key:value* pairs: the *key* is used to retrieve the *value*.

\[ D[key] = value \]

```python
const = {'e': 2.7183, 'pi': 3.1416, 'h': 6.6261e-34}
print(const)
{'h': 6.6261399999999998e-034,
 'pi': 3.1415999999999999,
 'e': 2.7183000000000002}
```
keys and values

- Key must be **immutable**
  - strings, integers, tuples are fine
  - lists are NOT
  - (floats are immutable so they work, but approximations can be a problem)
- Value can be **anything**

Access dictionary elements

Access requires [ ], but the key is the index!

```python
name_D={}
# an empty dictionary
name_D['bill']=25
# added the pair 'bill':25
print(name_D['bill'])
# prints 25
```
again, common operators

- **len**(myDict)
  - number of key:value pairs in the dictionary
- element *in* myDict
  - Boolean, is element a key in the dictionary?
- **for** `k` *in* myDict:
  - iterates through the keys of a dictionary

Lots of methods

- `my_dict.items()` – all the key:value pairs
- `my_dict.keys()` – all the keys
- `my_dict.values()` – all the values
- key *in* my_dict
  - `or my_dict.has_key(key)`
  - does the key exist in the dictionary?
Dictionaries are iterable

```python
for key in my_dict:
    print(key)
    # prints all the keys
for key, value in my_dict.items():
    print(key, value)
    # prints all the key:value pairs
for value in my_dict.values():
    print(value)
    # prints all the values
```

Building dictionaries faster

`zip` creates pairs from two parallel lists

```python
zip(“abc”, [1, 2, 3]) yields a series of tuples
(‘a’, 1), (‘b’, 2), (‘c’, 3)
```

That’s good for building dictionaries.

We call the `dict` constructor which takes a series of pairs to make a dictionary

```python
dict(zip(“abc”, [1, 2, 3])) yields
{‘a’: 1, ‘c’: 3, ‘b’: 2}
```
Example:
simple Dicts

Common way to build dictionaries

```python
my_dict = {}  # initialize to empty
for word in word_list:
    if word in my_dict:
        my_dict[word] += 1  # modify value
    else:
        my_dict[word] = 1  # initial value
```
Alternative way to build dictionaries

my_dict = {}  # initialize to empty
for word in word_list:
    try:
        my_dict[word]+= 1  # modify value
    except KeyError:
        my_dict[word] = 1  # initial value

D.get(word, 0) returns D[word] if it exists, else returns 0
Example word Frequency

Example: phoneBook

CSE 231
Lookup of functions

command_set={"init":init_from_file,
"lookup":lookup, "add":add_pair,
"del":delete_pair, "print":print_book}

response = input("Command:").strip()

command_set[response]()

Once a function is defined, it is an object just like any other variable. It can be stored as a value in a dictionary. You can call the function by appending ()
In math, a *set* is a collection of objects. In a set, no two elements are identical; there is no order to the elements of a set.
Creating a set

```python
my_set = {'a', 'b', 'c', 'd'}
print(my_set)
{'a', 'c', 'b', 'd'}
```

Create an empty set:
```python
my_set = set()
```
- the "set" keyword creates a set
- cannot use {}. Why not?

Alternative
```
my_set = set('abcd')
```
- the "set" keyword creates a set
- the single argument must be iterable,
```python
print(my_set)
{'a', 'c', 'b', 'd'}
```
A set can consist of a mixture of different types of elements.

\[
\text{my\_set} = \{\text{'a'}, 1, \text{3.14159}, \text{True}\}
\]

duplicates are automatically removed

\[
\text{my\_set} = \text{set('aabbccdd')}
\]
\[
\text{print(my\_set)}
\]
\[
\{\text{'a'}, \text{'c'}, \text{'b'}, \text{'d'}\}
\]
common operations

- `len(my_set)`
  - the number of elements in a set
- `element in my_set`
  - Boolean, is element in the set?
- `for` `element` `in` `my_set`:
  - iterate through the elements in `my_set`

Intersection

```
my_set={ 'a', 'd', 'c', 'b' }  
new_set={ 'd', 'e', 'c', 'f' }  
```

Intersection:
```
my_set & new_set  
returns { 'c', 'd' }  
```
Union

my_set={'a','d','c','b'}
new_set={'d','e','c','f'}

my_set | new_set
returns {'a','b','c','d','e','f'}

Difference

my_set={'a','d','c','b'}
new_set={'d','e','c','f'}

my_set - new_set
returns {'a','b'}

not commutative
Symmetric Difference

my_set = {'a', 'd', 'c', 'b'}
new_set = {'d', 'e', 'c', 'f'}

my_set ^ new_set returns {'a', 'b', 'e', 'f'}

subset and superset

small_set = {'a', 'b', 'c'}
big_set = {'a', 'b', 'c', 'd', 'e', 'f'}

small_set < big_set returns True
big_set > small_set returns True
Other Set Ops

my_set.add('g')
  - adds to the set, no effect if item is in set already

Long versions

- my_set.intersection(new_set)
- my_set.union(new_set)
- my_set.difference(new_set)
- my_set.symmetric_difference(new_set)
- my_set.issubset(new_set)
- new_set.issuperset(my_set)
Other Set Ops (less common)

- `m_set.clear()`
  - emptys the set

- `my_set.remove('g')` versus `my_set.discard('g')`
  - remove throws an error if “g” isn’t there. discard doesn’t care. Both remove “g” from the set

- `my_set.copy()`
  - returns a shallow copy of `my_set`

Copy vs. assignment

```python
my_set=('a','b','c')
my_copy=my_set.copy()
my_ref_copy=my_set
my_set.remove('b')
```

```
my_set  {'a','c'}
my_copy  {'a','b','c'}
my_ref_copy  {'a','b','c'}
```
Example: simpleSets

CSE 231

Example: word count
Comprehensions

- Build a set
  \{ch for ch in 'abcd'}
  is \{'a','b','c','d'}

- Reverse a dictionary D
  \{v:k for k,v in D.items()\}