Data Structures in Python

3. Sets

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Sets

- Python provides a data structure that represents a mathematical ***set***.
- As with mathematical sets, we use curly braces ({}) in Python code to enclose the elements of a literal set.
- Python distinguishes between set literals and dictionary literals by the fact that all the items in a dictionary are colon-connected (:) key-value pairs, while the elements in a set are simply values.
- Unlike Python lists, sets are unordered and may contain no duplicate elements.
- The following interactive sequence demonstrates these set properties:

```
>>> S = {10, 3, 7, 2, 11}
>>> S
{2, 3, 7, 10, 11}
>>> T = {5, 4, 5, 2, 4, 9}
>>> T
{9, 2, 4, 5}
```

Note that the element ordering of the input is different from the ordering in the output. Also observe that sets do not admit duplicate elements.

Sets

- We can make a *set* out of a *list* using the ***set conversion function***:

```
>>> L = [10, 13, 10, 5, 6, 13, 2, 10, 5]
>>> S = set(L)
>>> L
[10, 13, 10, 5, 6, 13, 2, 10, 5]
>>> S
{2, 5, 6, 10, 13}
```

- As you can see, the element ordering is not preserved, and duplicate elements appear only once in the set.
- Python ***set*** notation exhibits one important difference with mathematics: the expression {} does not represent the empty set.
- In order to use the curly braces for a set, the set must contain at least one element.
- The expression ***set()*** produces a set with no elements, and thus represents the empty set.
- Python reserves the {} notation for empty dictionaries.

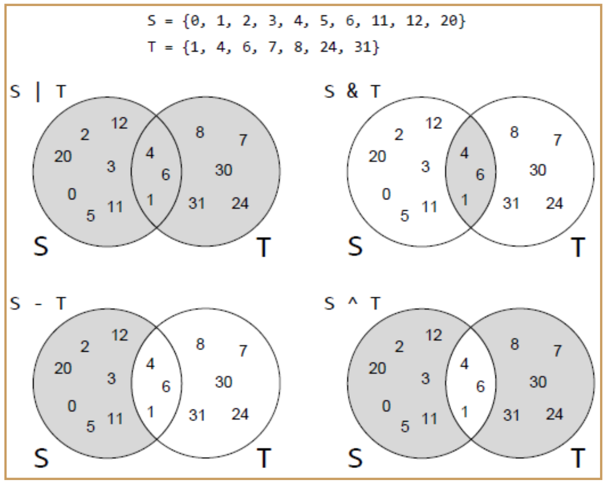
Sets

- Note that the elements contained in a set must be of an ***immutable*** type.
- Python supports the standard mathematical set operations of ***intersection***, ***union***, ***set difference***, and ***symmetric difference***.

Operation	Mathematical Notation	Python Syntax	Result Type	Meaning
Union	$A \cup B$	`A | B`	set	Elements in A or B or both
Intersection	$A \cap B$	`A & B`	set	Elements common to both A and B
Set Difference	$A - B$	`A - B`	set	Elements in A but not in B
Symmetric Difference	$A \oplus B$	`A ^ B`	set	Elements in A or B, but not both
Set Membership	$x \in A$	`x in A`	bool	x is a member of A
Set Membership	$x \notin A$	`x not in A`	bool	x is not a member of A
Set Equality	$A = B$	`A == B`	bool	Sets A and B contain exactly the same elements
Subset	$A \subseteq B$	`A <= B`	bool	Every element in set A also is a member of set B
Proper Subset	$A \subset B$	`A < B`	bool	A is a subset B, but B contains at least one element not in A

Sets

- The figure illustrates how the set operations work.
- The following interactive sequence computes the ***union*** and ***intersection*** of two sets and tests for ***set membership***:




```
>>> S = {2, 5, 7, 8, 9, 12}
>>> T = {1, 5, 6, 7, 11, 12}
>>> S|T
{1, 2, 5, 6, 7, 8, 9, 11, 12}
>>> S&T
{12, 5, 7}
```

```
>>> 7 in S
True
>>> 11 in S
False
```


Sets

- To determine whether or not two sets have any elements in common, use the method: **x1.isdisjoint(x2)**
  It returns ***True*** if x1 and x2 have no elements in common.

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'baz', 'qux', 'quux'}
>>> x1.isdisjoint(x2)
False
>>> x2 - {'baz'}
{'quux', 'qux'}
>>> x1.isdisjoint(x2 - {'baz'})
True
```

- **Note:** There is no operator that corresponds to the ***isdisjoint()*** method.

- If x1.isdisjoint(x2) is ***True***, then x1 & x2 is the ***empty set***:

```
>>> x1 = {1, 3, 5}
>>> x2 = {2, 4, 6}
>>> x1.isdisjoint(x2)
True
>>> x1 & x2
set()
```

Sets

➢ Modifying a Set

- Although the elements contained in a set must be of immutable type, sets themselves can be modified. There are some operators and methods that can be used to change the contents of a set.

❖ *Augmented Assignment Operators*

- Each of the union, intersection, difference, and symmetric difference operators listed above has an ***augmented assignment*** form that can be used to modify a set.
- Modify a set by union: x1 |= x2 [| x3 ...]
  x1 |= x2 adds to x1 any elements in x2 that x1 does not already have:

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}
>>> x1 |= x2
>>> x1
{'qux', 'foo', 'bar', 'baz'}
```

7

Sets

- Modify a set by intersection: x1 &= x2 [& x3 ...]
  x1 &= x2 updates x1, retaining only elements found in both x1 and x2:

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}
>>> x1 &= x2
>>> x1
{'foo', 'baz'}
```

- Modify a set by difference: x1 -= x2 [| x3 ...]
  x1 -= x2 updates x1, removing elements found in x2:

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}
>>> x1 -= x2
>>> x1
{'bar'}
```

- Modify a set by symmetric difference: x1 ^= x2
  x1 ^= x2 updates x1, retaining elements found in either x1 or x2, but not both:

8

Sets

```
>>> x1 = {'foo', 'bar', 'baz'}
>>> x2 = {'foo', 'baz', 'qux'}
>>> x1 ^= x2
>>> x1
{'bar', 'qux'}
```

❖ ***Other Methods For Modifying Sets***

- Python supports several additional methods that modify sets.
- Add an element to a set: x.add(<elem>)

x.add(<elem>) adds ***<elem>***, which must be a single immutable object, to x:

```
>>> x = {'foo', 'bar', 'baz'}
>>> x.add('qux')
>>> x
{'bar', 'baz', 'foo', 'qux'}
```

- Remove an element from a set: x.remove(<elem>)

x.remove(<elem>) removes ***<elem>*** from x. Python raises an exception if ***<elem>*** is not in x:

9

Sets

```
>>> x = {'foo', 'bar', 'baz'}
>>> x.remove('baz')
>>> x
{'bar', 'foo'}
>>> x.remove('qux')
Traceback (most recent call last):
  File "<pyshell#58>", line 1, in <module>  x.remove('qux')
KeyError: 'qux'
```

- Remove an element from a set: x.discard(<elem>)

x.discard(<elem>) also removes ***<elem>*** from x. However, if ***<elem>*** is not in x, this method quietly does nothing instead of raising an exception:

```
>>> x = {'foo', 'bar', 'baz'}
>>> x.discard('baz')
>>> x
{'bar', 'foo'}
>>> x.discard('qux')
>>> x
{'bar', 'foo'}
```

10

Sets

- Remove a random element from a set: x.pop()
  x.pop() removes and returns an arbitrarily chosen element from x. If x is empty, x.pop() raises an exception:

```
>>> x = {'foo', 'bar', 'baz'}
>>> x.pop()
'bar'
>>> x
{'baz', 'foo'}
>>> x.pop()
'baz'
>>> x
{'foo'}
```

```
>>> x.pop()
'foo'
>>> x
set()
>>> x.pop()
Traceback (most recent call last):
  File "<pyshell#82>", line 1, in <module>
x.pop()
KeyError: 'pop from an empty set'
```

- Clear a set: x.clear()
  x.clear() removes all elements from x:

```
>>> x = {'foo', 'bar', 'baz'}
>>> x
{'foo', 'bar', 'baz'}
>>> x.clear()
>>> x
set()
```

11

Sets

- As with list comprehensions and generator expressions, we can use ***set comprehension*** to build sets.
- The syntax is the same as for list comprehension, except we use curly braces rather than square brackets.
- The following interactive sequence constructs the set of perfect squares less than 100:

```
>>> S = {x**2 for x in range(10)}
>>> S
{0, 1, 64, 4, 36, 9, 16, 49, 81, 25}
```

- The displayed order of elements is not as nice as the list version, but, again, element ordering is meaningless with sets.
- When treated as a Boolean expression, the ***empty set*** (***set()***) is interpreted as ***False***, and any other set is considered ***True***.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

12

Set Quantification with *all* and *any*

- Python provides functions named ***all*** and ***any*** that respectively correspond to mathematical ***universal*** and ***existential*** quantification.
- ***Universal quantification*** means that a particular property is true for all the elements of a set.
- ***Existential quantification*** means that at least one element in the set exhibits a particular property.
- In mathematics the ∀ (***for all***) symbol represents universal quantification, and the ∃ (***there exists***) symbol represents existential quantification.
- To see how we can use these quantifiers in a Python program, consider the set S = {1, 2, 3, 4, 5, 6, 7, 8}.
- To express in mathematics the fact that all the elements in set S are greater than zero, we can write **$(\forall x \in S)(x > 0)$**
- This is a statement that is either ***true*** or ***false***, and we can see that it is a ***true*** statement.

13

Set Quantification with *all* and *any*

- In Python, we first will use a list comprehension to see which elements in S are greater than zero. We can do this by building a list of Boolean values by using a Boolean expression in the list comprehension:

```
S = {1, 2, 3, 4, 5, 6, 7, 8}
[x > 0 for x in S]
[True, True, True, True, True, True, True, True]
```

- We can see that all the entries in this list are True, but the best way to determine this in code is to use Python's ***all function***:

```
all([x > 0 for x in S])
True
```

- The ***all function*** returns ***True*** if all the elements in a list, set, or other iterable possesses a particular quality.
- We do not need to create a list; a generator expression is better (note that parentheses replace the square brackets):

Science Minia University

14

Set Quantification with *all* and *any*

all((x > 0 for x in S))
True

and in this case the inner parentheses are superfluous. We can rewrite the expression as:

all(x > 0 for x in S)
True

- This expression is Python's way of checking the mathematical predicate **($\forall$x $\in$ S)(x > 0)**
- The ***any function*** returns ***True*** if any element in a list, set, or other iterable possesses a particular quality.
- This means the ***any function*** represents the mathematical existential quantifier, $\exists$:

any(x > 0 for x in S)
True

- This expression is Python's way of checking the mathematical predicate **($\exists$x $\in$ S)(x > 0)**

Set Quantification with *all* and *any*

- Certainly if the property holds for all the elements in set S, there is at least one element for which it holds.
- Are all the elements of S greater than 5?

all(x > 5 for x in S)
False

- The answer is false, of course, because the set contains 1, 2, 3, 4, and 5, none of which are greater than 5.
- But, there are some elements in S that are greater than 5:

any(x > 5 for x in S)
True

- The answer is True as the elements 6, 7, and 8 are all greater than 5.
- Does the set contain an element greater than 10?

any(x > 10 for x in S)
False

- We can see that none of the elements in S are greater than 10.

Set Quantification with *all* and *any*

- If none of the set's elements possess the particular property, it certainly cannot be true for all the elements in the set:

```
all(x > 10 for x in S)
False
```

- The ***all*** and ***any*** functions work with any iterable object: sets, lists, dictionaries, and generated sequences.

Sets vs. Lists

- If order does not matter and all elements are unique, the ***set*** type does offer a big advantage over the ***list*** type: testing for membership using ***in*** is much faster on sets than lists.
- The following program creates both a set and a list, each containing the first 1,000 perfect squares. It then searches both data structures for all the integers from 0 to 999,999, and reports the time required for both searches.

17

Sets vs. Lists

```
# Data structure size
size = 1000
# Make a big set
S = {x**2 for x in range(size)}
# Make a big list
L = [x**2 for x in range(size)]
# Verify the type of S and L
print('Set:', type(S), ' List:', type(L))
from time import perf_counter
# Search size
search_size = 1000000
# Time list access
start_time = perf_counter()
for i in range(search_size):
    if i in L:
        pass
stop_time = perf_counter()
print('List elapsed:', stop_time - start_time)
```

```
# Time set access
start_time = perf_counter()
for i in range(search_size):
    if i in S:
        pass
stop_time = perf_counter()
print('Set elapsed:',
        stop_time - start_time)
```

Run

```
Set: <class 'set'>  List: <class 'list'>
List elapsed: 21.215297687
Set elapsed: 0.212674243000000215
```

Note that the set membership test was almost 100 times faster than the exact same test performed on the list.

18

Sets vs. Lists

- Recall that the word count program grouped words from a text file according to their length. The program contained a check to avoid duplicate entries:

```
if size in groups:
  if word not in groups[size]:      # Avoid duplicates
            groups[size] += [word] # Add the word to its group
else:
  groups[size] = [word] # Add the word to a new group
```

- We know now that if we used sets of words rather than lists of words we could have eliminated the check for duplicate entries.

```
if size in groups:
  groups[size] | {word}   # Add the word to its group
else:
  groups[size] = {word}  # Add the word to a new group
```

- By removing this extra check we also remove the application of the ***in*** operator on a list. This removes the potentially costly search for an element within a large list, since testing for membership within a list is more costly than testing for membership within a set.

19

Frozen Sets

- Python provides another built-in type called a ***frozenset***, which is in all respects exactly like a set, except that a *frozenset* is *immutable*. You can perform non-modifying operations on a *frozenset*:

```
>>> x = frozenset(['foo', 'bar', 'baz'])
>>> x
frozenset({'foo', 'baz', 'bar'})
>>> len(x)
3
>>> x & {'baz', 'qux', 'quux'}
frozenset({'baz'})
```

- But methods that attempt to modify a *frozenset* fail:

```
>>> x = frozenset(['foo', 'bar', 'baz'])
>>> x.add('qux')
Traceback (most recent call last):
  File "<pyshell#127>", line 1, in <module>  x.add('qux')
AttributeError: 'frozenset' object has no attribute 'add'
```

20

Frozen Sets

```
>>> x.pop()
Traceback (most recent call last):
  File "<pyshell#129>", line 1, in <module> x.pop()
AttributeError: 'frozenset' object has no attribute 'pop'
>>> x
frozenset({'foo', 'bar', 'baz'})
```

Frozensets and Augmented Assignment

- Since a ***frozenset*** is immutable, you might think it can't be the target of an augmented assignment operator. But observe:

```
>>> f = frozenset(['foo', 'bar', 'baz'])
>>> s = {'baz', 'qux', 'quux'}
>>> f &= s
>>> f
frozenset({'baz'})
```

- Python does not perform augmented assignments on frozensets in place. The statement **x &= s** is equivalent to **x = x & s**. It isn't modifying the original x. It is reassigning x to a new object, and the object x originally referenced is gone.

21

Frozen Sets

- *Frozensets are useful in situations where you want to use a set, but you need an immutable object.* For example, you can't define a set whose elements are also sets, because set elements must be immutable:

```
>>> x1 = set(['foo'])
>>> x2 = set(['bar'])
>>> x3 = set(['baz'])
>>> x = {x1, x2, x3}
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    x = {x1, x2, x3}
TypeError: unhashable type: 'set'
```

- If you really need to define a set of sets, you can do it if the elements are frozensets, because they are immutable:

```
>>> x1 = frozenset(['foo'])
>>> x2 = frozenset(['bar'])
>>> x3 = frozenset(['baz'])
>>> x = {x1, x2, x3}
>>> x
{frozenset({'bar'}), frozenset({'baz'}), frozenset({'foo'})}
```

22

Frozen Sets

- Likewise, recall from the previous lecture on *dictionaries* that a dictionary key must be immutable. You can't use the built-in set type as a dictionary key:

```
>>> x = {1, 2, 3}
>>> y = {'a', 'b', 'c'}
>>> d = {x: 'foo', y: 'bar'}
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    d = {x: 'foo', y: 'bar'}
TypeError: unhashable type: 'set'
```

- If you need to use sets as dictionary keys, you can use frozensets:

```
>>> x = frozenset({1, 2, 3})
>>> y = frozenset({'a', 'b', 'c'})
>>>
>>> d = {x: 'foo', y: 'bar'}
>>> d
{frozenset({1, 2, 3}): 'foo', frozenset({'c', 'a', 'b'}): 'bar'}
```

23

Enumerating the Elements of a Data Structure

- The following code prints out the contents of a list named *lst*, along with the indices of the individual elements:

```
for i in range(len(lst)):
    print(i, lst[i])
```

- This code requires two function calls in order to manage the indices: one call to ***len*** to determine the highest index and another call to the ***range constructor*** to produce each index.
- The **__builtins__** module provides a function named ***enumerate*** that returns an iterable object that produces tuples. Each tuple pairs an index with its associated element.
- The following code uses the ***enumerate*** function to produce the same results as the above code:

```
for i, elem in enumerate(lst):
    print(i, elem)
```

- One call to ***enumerate*** replaces the two calls from before.

Enumerating the Elements of a Data Structure

- In some circumstances code that uses enumerate can be slightly more efficient than the code that manually manages the integer index.
- The ***enumerate*** function accepts any type of object that supports iteration.
- The following program demonstrates the use of ***enumerate*** with ***lists***, ***tuples***, ***dictionaries***, ***sets***, and ***generators***:

```
lst = [10, 20, 30, 40, 50]
t = 100, 200, 300, 400, 500
d = {"A": 4, "B": 18, "C": 0, "D": 3}
s = {1000, 2000, 3000, 4000, 5000}
print(lst)
print(t)
print(d)
print(s)
for x in enumerate(lst):
    print(x, end=" ")
print()
```

25

Enumerating the Elements of a Data Structure

```
for x in enumerate(t):
    print(x, end=" ")
print()
for x in enumerate(d):
    print(x, end=" ")
print()
for x in enumerate(s):
    print(x, end=" ")
print()
def gen(n):
    """ Generate n, n - 2, n - 3, ..., 0. """
    for i in range(n, -1, -2):
        yield i
for x in enumerate(gen(20)):
    print(x, end=" ")
print()
# Optionally specify beginning index
for x in enumerate(t, 1):
    print(x, end=" ")
print()
```

26

Enumerating the Elements of a Data Structure

Output

```
[10, 20, 30, 40, 50]
(100, 200, 300, 400, 500)
{'A': 4, 'B': 18, 'C': 0, 'D': 3}
{4000, 1000, 5000, 2000, 3000}
(0, 10) (1, 20) (2, 30) (3, 40) (4, 50)
(0, 100) (1, 200) (2, 300) (3, 400) (4, 500)
(0, 'A') (1, 'B') (2, 'C') (3, 'D')
(0, 4000) (1, 1000) (2, 5000) (3, 2000) (4, 3000)
(0, 20) (1, 18) (2, 16) (3, 14) (4, 12) (5, 10) (6, 8) (7, 6) (8, 4) (9, 2) (10, 0)
(1, 100) (2, 200) (3, 300) (4, 400) (5, 500)
```

- The last call to ***enumerate*** in the above program uses an optional parameter specifying the beginning index to use in the enumeration. The default starting index is 0.