CIS 1100

Sets & Dicts

Python

Fall 2024

University of Pennsylvania

Learning Objectives

- Explain the purpose of sets as *unordered* collections of *unique* elements.
- Use basic operations of sets: checking for membership, adding & removing elements, set intersection & difference
- Apply knowledge of comprehensions to sets
- Explain the purpose of dicts as *mappings* from keys to vbalues
- Use basic operations of dicts: checking for membership, adding/updating/removing key-value pairs
- Apply knowledge of comprehensions to dicts

CIS 1100

Sets

Python

Fall 2024

University of Pennsylvania

Sets as Unordered Collections

Recall that lists are sequences: *ordered* collections of values.

- *ordered* ➡️ indexed
- no restrictions on the values we store (types, repeats)

Sets are collections, too, but they are **unordered** and they do not allow **duplicate elements.**

- **unordered** ➡️ no indexing!
- can store values of (nearly) any type
- each value can be present at most one time

Set Syntax

Set literals are defined with curly braces (`{}`).

- `{3, 10, 15}` is a set with three elements
- `{"Harry", "Travis"}` is a set with two elements
- `{}` is not a set at all
  - it's a `dict` (more on this shortly)
  - unlike how `[]` gives us an empty list, we need to write `set()` to give us an empty set (a set with no elements)

Uniqueness & No Ordering

- Any two sets that have exactly the same elements are considered equal to each other.

```
>>> set_one = {3, 10, 15}
>>> set_two = {15, 10, 3}
>>> set_one == set_two
True
```

- Adding a "duplicate" value to a set has no effect.

```
>>> set_with_duplicates = {"Harry", "Travis", "Harry"}
>>> len(set_with_duplicates)
2
```

Restrictions

Sets cannot store "unhashable" elements.

- What is or isn't hashable is of no concern to us now...
- ...but keep in mind that you can't store **lists, sets, or dicts** within sets.
- `tuple` and `str` values are still OK!

CIS 1100

Set Operations

Python

Fall 2024

University of Pennsylvania

Standard Collection Operations

Lots of stuff familiar from lists:

- `len()` tells you how many elements are present
- `x in s` allows you to check if some element `x` is in a set `s`
- `for x in s` allows you to iterate over the elements of `s` one-by-one
  - sets are unordered, so the iteration order is (sort of) unpredictable

Creating Sets

You can use `set()` to turn another collection into a set. This adds all elements of the other collection to the set (and therefore removes all duplicates.)

```
>>> fibs = [0, 1, 1, 2, 3]
>>> fib_set = set(fibs)
>>> fib_set
{0, 1, 2, 3}
```

Iteration over Sets

Using a `for` loop still visits each element in the set, but you don't know the order!

```python
my_set = {3, 10, 15}
for number in my_set:
    print(number)
```

```
10
3
15
```

Adding Elements

To add an element to a set, use the `.add()` method:

```
names = {"Crosby", "Young", "Stills"}
names.add("Nash")
print(names)
```

🖨️ 👇

```
{'Nash', 'Stills', 'Crosby', 'Young'}
```

Removing Elements

To remove an element from a set, use the `.remove()` method:

```
names = {"Crosby", "Young", "Stills", "Nash"}
names.remove("Young")
print(names)
```

🖨️ 👇

```
{'Nash', 'Stills', 'Crosby'}
```

Removing Elements

If you try to `.remove()` an element that's not present, you get a `KeyError` (a program crash!)

```
names = {"Crosby", "Young", "Stills", "Nash"}
names.remove("Harry")
print(names)
```

🖨 👇

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'harry'
```

Removing Elements Safely

If you try to `.discard()` an element that's not present, nothing happens!

```python
names = {"Crosby", "Young", "Stills", "Nash"}
names.discard("Harry")
print(names)
```

🖨️ 👇

```
{'Nash', 'Stills', 'Crosby', 'Young'}
```

Set Operations

Name	Meaning	Method	Operator
Union	Create a new set with all elements from both	`s.union(t)`	`s | t`
Intersection	Create a new set with only elements that appear in both sets	`s.intersection(t)`	`s & t`
Difference	Create a new set with only elements in `s` that don't appear in `t`	`s.difference(t)`	`s - t`
Symmetric Difference	Create a new set with elements that appear in only one set *but not both*	`s.symmetric_difference(t)`	`s ^ t`

Set Operations

I have two sets `session_one` and `session_two` that contain the names of people who attended recitation one and recitation two, respectively. How can I...

- find all of the people who attended both?
- find all of the people who attended at least one?
- find all the people who attended exactly one?
- find all the people who attended the first but not the second?

Set Operations

I have two sets `session_one` and `session_two` that contain the names of people who attended recitation one and recitation two, respectively. How can I...

- find all of the people who attended both?
    - `both = session_one & session_two`
- find all of the people who attended at least one?
    - `at_least_one = session_one | session_two`
- find all the people who attended exactly one?
    - `exactly_one = session_one ^ session_two`
- find all the people who attended the first but not the second?
    - `just_first = session_one - session_two`

Set Relations: Supersets

Set `s` is a superset of set `t` if all elements of `t` are present in `s`.

- `s >= t` is `True` when all elements of `t` are present in `s`
  Set `s` is a strict superset of set `t` if all elements of `t` are present in `s` and `len(s) > len(t)`
- `s > t` is `True` when `s >= t` and `len(s) > len(t)`

Set Relations: Subsets

Set `s` is a subset of set `t` if all elements of `s` are present in `t`.

- `s <= t` is `True` when all elements of `s` are present in `t`

  Set `s` is a strict subset of set `t` if all elements of `s` are present in `t` and `len(s) < len(t)`
- `s < t` is `True` when `s <= t` and `len(s) < len(t)`

CIS 1100

Set Comprehensions

Python

Fall 2024

University of Pennsylvania

Set Comprehensions

Create sets with comprehensions exactly the same way it's done with lists, but using `{}` instead of `[]`:

```
new_set = {expression(elem) for elem in sequence if condition(elem)}
```

Set Comprehension

Get a set of all of the vowels present in a string:

```
>>> word = "Avarice"
>>> vowels = {letter.upper() for letter in word if letter in "AaEeIiOoUu"}
>>> vowels
{'A', 'I', 'E'}
```

CIS 1100

Dicts

Python

Fall 2024

University of Pennsylvania

Dicts as Mappings from Keys to Values

Dicts are **unordered** collections of key-value pairs.

- Short for "dictionary"
- Represent associations from **keys** to **values**
- Allow for looking up some information associated with a search key
- Keys must be unique, values do not need to be unique

What is a Mapping?

Any association from **keys** (things you can search by) to **values** (information you might want to know.)

The Penn Directory, for example:

```
Name : Email
Harry Smith : sharry@seas
Travis McGaha : tqmcgaha@seas
...
```

Here, the names are keys and the emails are values.

Dict Syntax

Dict literals are defined with curly braces (`{}`) and separate keys and values with a colon.

- `{3, 10, 15}`
  - is a **set** with three elements
- `{"Harry" : "sharry", "Travis" : "tqmcgaha"}`
  - is a **dict** with two elements (key-value pairs)
- `{}` is an empty dict
  - writing just `dict()` gets the same result

Uniqueness & No Ordering

- Any two dicts that have exactly the same elements are considered equal to each other.

```
>>> one = {"Harry" : "sharry", "Travis" : "tqmcgaha"}
>>> two = {"Travis" : "tqmcgaha", "Harry" : "sharry"}
>>> one == two
True
```

Restrictions

Dicts cannot store "unhashable" keys.

- What is or isn't hashable is of no concern to us now...
- ...but keep in mind that you can't use **lists, sets, or dicts** as keys.
- `tuple` and `str` keys are still OK!

CIS 1100

Dict Operations

Python

Fall 2024

University of Pennsylvania

Standard Operations

Lots of stuff familiar from lists and sets:

- `len()` tells you how many elements are present
- `k in d` allows you to check if some **key** `x` is in a dict `d`
- `for k in d` allows you to iterate over the **keys** of `d` one-by-one
  - dicts are unordered, so the iteration order is (sort of) unpredictable

Views of the Dictionary Contents

- `.keys()` is a view of the keys in a dict.
- `.values()` is a view of the values in a dict.
- `.items()` is a view of the key-value pairs in a dict (as tuples).

Views of the Dictionary Contents

```python
recitations = {210 : "Sukya", 211 : "Jared", 212: "Molly", 213: "Adi", 214: "Cedric"}
for key in recitations.keys():
    print(key)
```

```
210
211
212
213
214
```

(Also works the same with `for key in recitations:`)

Views of the Dictionary Contents

```python
recitations = {210 : "Sukya", 211 : "Jared", 212: "Molly", 213: "Adi", 214: "Cedric"}
for value in recitations.values():
    print(value)
```

🖨

```
Sukya
Jared
Molly
Adi
Cedric
```

Views of the Dictionary Contents

```python
recitations = {210 : "Sukya", 211 : "Jared", 212: "Molly", 213: "Adi", 214: "Cedric"}
for item in recitations.items():
    print(item)
```

🖨

```
(210, 'Sukya')
(211, 'Jared')
(212, 'Molly')
(213, 'Adi')
(214, 'Cedric')
```

Adding Elements

To add an element to a dict, use the "indexing" (`[]`) syntax with assignment (`=`):

```python
faves = {2022: "Things to Come and Go", 2023: "Checkout 19", 2024: "Last Summer in the City"}
faves[2021] = "Gilead"
print(faves)
```

🖨️ 👇

```
{2022: 'Things to Come and Go', 2023: 'Checkout 19', 2024: 'Last Summer in the City', 2021: 'Gilead'}
```

Looking Up Elements

To check the value associated with a key, use the "indexing" syntax:

```python
faves = {2022: "Things to Come and Go", 2023: "Checkout 19", 2024: "Last Summer in the City"}
print(faves[2022])
```

🖨️ 👇

```
'Things to Come and Go'
```

Looking Up Elements

If a key is not present, you end up with a `KeyError` (crash!) when looking for it:

```
faves = {2022: "Things to Come and Go", 2023: "Checkout 19", 2024: "Last Summer in the City"}
print(faves[1854])
```

🖨️ 👇

```
KeyError
```

Updating Elements

To update the value associated with a key, reassign it!

```python
faves = {2022: "Things to Come and Go", 2023: "Checkout 19", 2024: "Last Summer in the City"}
faves[2024] = "The Details"
print(faves)
```

🖨️ 👇

```
{2022: 'Things to Come and Go', 2023: 'Checkout 19', 2024: 'The Details'}
```

Removing Elements

To remove a key-value pair from a dict, use `del`:

```python
faves = {2022: "Things to Come and Go", 2023: "Checkout 19", 2024: "Last Summer in the City"}
del faves[2024]
print(faves)
```

🖨 👇

```
{2022: 'Things to Come and Go', 2023: 'Checkout 19'}
```

(Leads to `KeyError` again if you delete a key not present)

CIS 1100

Using Dictionaries

Python

Fall 2024

University of Pennsylvania

Dicts as Counters

If I poll the class and get a list of everyone's favorite restaurant, how can I count how many times each restaurant was named?

Dicts as Counters

```
def get_counts_from_list(faves_list):
    counter = {}
    for restaurant in faves_list:
        if restaurant in counter:
            counter[restaurant] = counter[restaurant] + 1
        else:
            counter[restaurant] = 1
    return counter
```

Dicts as Counters

What were the final counts?

```
>>> tally = get_counts_from_list(["Han Dynasty", "Tampopo", "Halal Guys", "Tampopo", "Tampopo"])
>>> tally
{'Han Dynasty': 1, 'Tampopo': 3, 'Halal Guys': 1}
```

Did Goldie get any votes? Tampopo?

```
>>> "Goldie" in tally
False
>>> "Tampopo" in tally
True
>>> tally["Tampopo"]
3
```

CIS 1100

Dict Comprehensions

Python

Fall 2024

University of Pennsylvania

Dict Comprehensions

Create dicts with comprehensions exactly the same way it's done with sets, but specifying `key:value` pairs:

```
new_set = {key : value for elem in sequence if condition(elem)}
```

Dict Comprehension

Get a mapping of the length of each string in a list:

```
>>> names = ["Harry", "Travis", "Jared", "Adi"]
>>> name_lengths = {name : len(name) for name in names}
>>> name_lengths
{'Harry': 5, 'Travis': 6, 'Jared': 5, 'Adi': 3}
```