15-110 Principles of Computing – S19

Lecture 14: Dictionaries 2, Sets

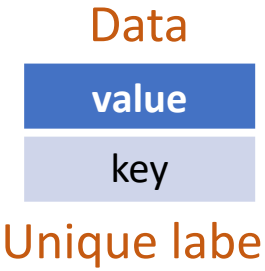
Teacher:
Gianni A. Di Caro

Recap on dictionaries

- **Dictionary**: *associative data structure* in the form of a *collection* (unordered) of *pairs* (key, value)

```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```

- To form a dictionary element, a **unique *label*** (an identification *key*) is associated to a piece of data (the *value* identified by the label)



key ↦ value

- In the dictionary, data (values) are *inserted*, *accessed*, and *modified* using their associated labels
- In sequences, data (values) are *inserted*, *accessed*, and *modified* using a positional index
  - Update value of existing keys

```
d['John'] = 30
```

  - Add a new key-value pair:

```
d['Kim'] = 18
```

  - Delete an existing item:

```
del d['Jim']
```

- A dictionary is a **mutable** data type (e.g., if `x` is a dictionary, `y = x` creates an **alias**)
- A **key** can only contain **immutable** data types, a **value** can be of **any type**
- In a dictionary, neither the keys nor the values need to be of the same type

E.g. `'Jim', 2, (1.5,2)` can be keys in the same dictionary

Why do we need an associative data structure?

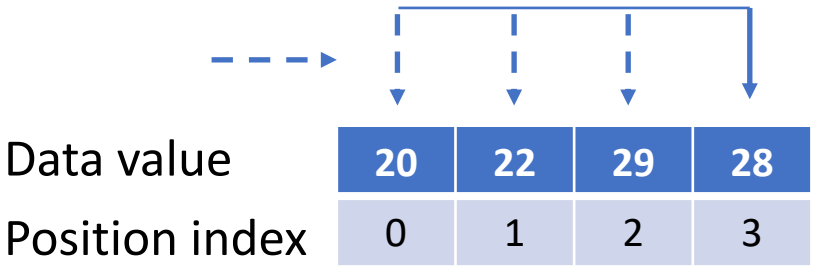
- ✓ Because when data can be naturally and uniquely **labeled** that provides a way to:
  - Organize data through the labels
  - Describe the data through the labels
  - Search the data using the labels

    - Collection of phone numbers: name as labels
    - Collection of animal data: name as label
    - People data: ID number or name as labelss
    - Car databases: Name+Model+Year as labels
    - Values of a function $f(x, y)$: coordinate values (x, y) as labels
    - ….

Why do we need an associative data structure?

- ✓ Because by using labels we can access to values much more **efficiently** than with lists, for instance
  - Dictionaries are in fact **hashed** data types, while lists (sequences) are *indexed* data types

`value in data_list ?`

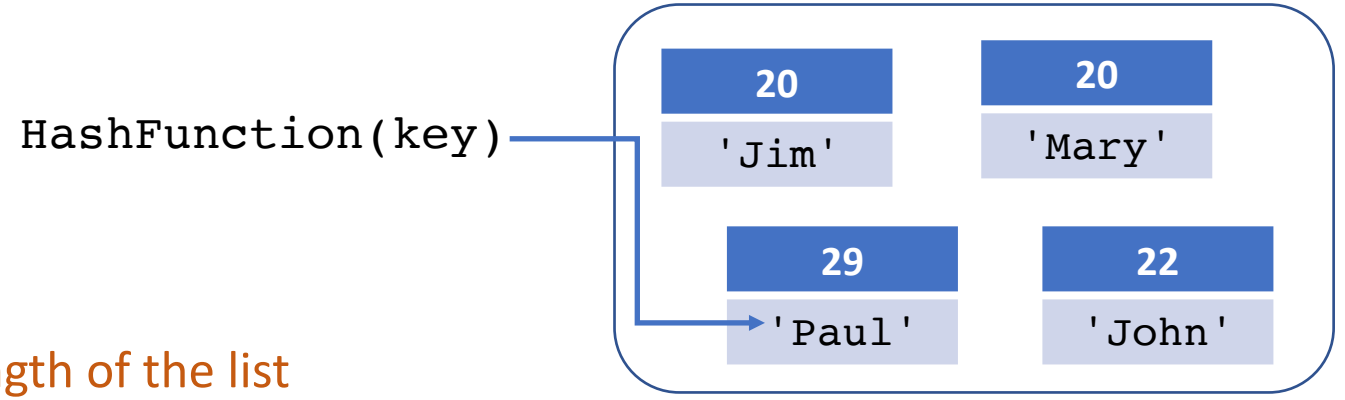



```
x = [20, 22, 29, 28]
```

Worst-case search time **linear** with the length of the list

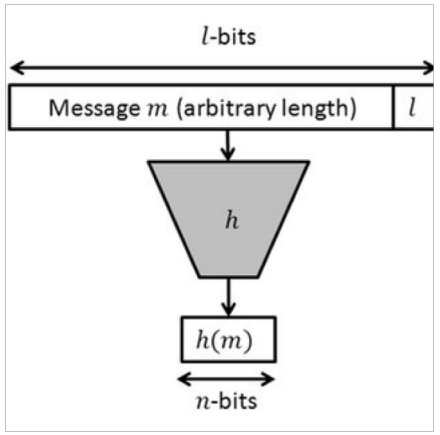
`key in dictionary ?`

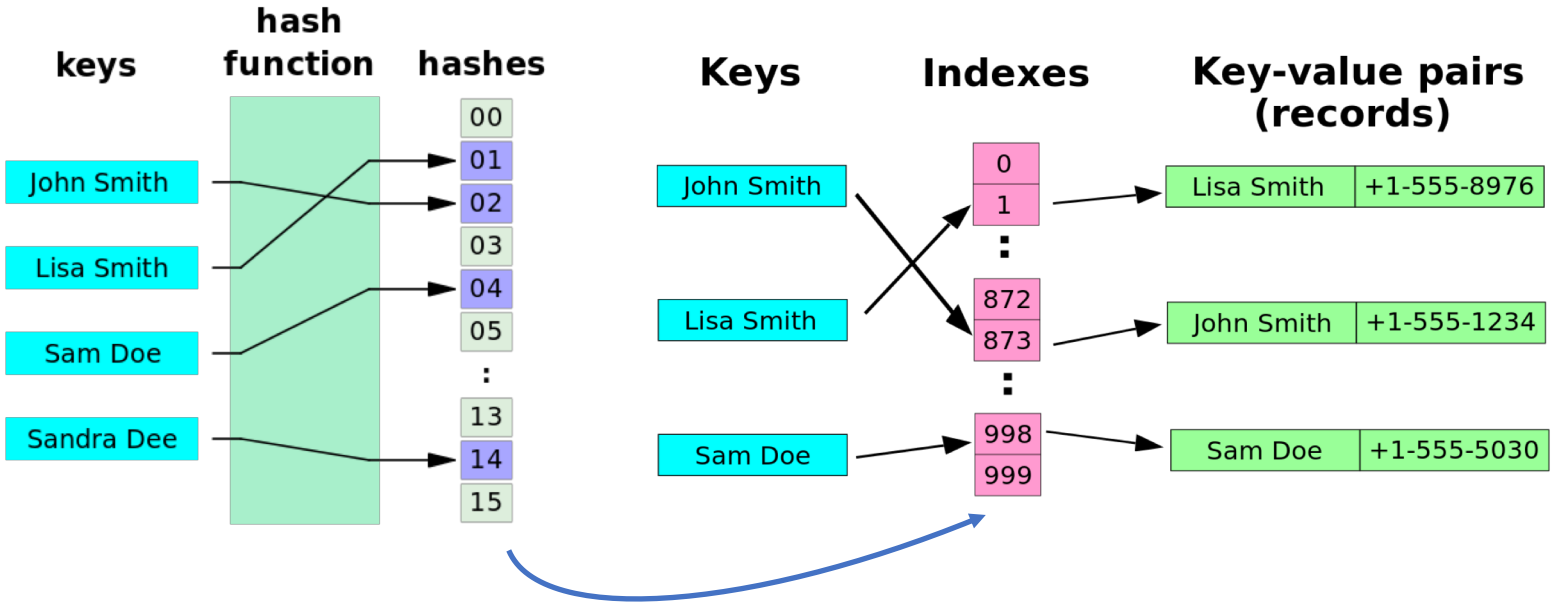
```
d = {'John': 22, 'Jim': 20, 'Mary': 20, 'Paul': 29}
```

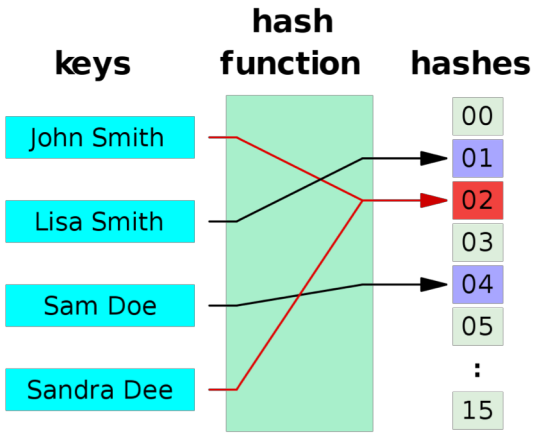


Constant search time
(independent of dictionary size)

Dictionary as hashed data type







- **Computation time:**
  Time to compute the hash
  + one direct memory access

- **Insert a new pair** `(key, value)` to a dictionary:
  - ✓ `hash(key)` computes an *index*
  - ✓ The pair `(key, value)` is stored directly at the *index* location

- **Access the value associated to key:**
  - ✓ `hash(key)` gives the *index*
  - ✓ The pair `(key, value)` at the *index* location is returned directly

Efficiency: Time comparison between lists and dictionaries

```
import time
n = 100000000
large_dict = {}
large_list = []
large_set = {0}
type_str = {type({}): "Dict:", type([]): "List", type({1}): "Set:"}

def initialize(data_struct, num):
    if len(data_struct) > 1:
            data_struct.clear()
    start = time.process_time()
    if type(data_struct) == dict:
        for i in range(num):
            data_struct[i] = i
    elif type(data_struct) == list:
        for i in range(num):
            data_struct.append(i)
    else:
        print("Error in data type!")
        return

    end = time.process_time()
    print(type_str[type(data_struct)], end - start, flush=True)
```

Efficiency: Time comparison between lists and dictionaries

```
def find_value(val, where):
    start = time.process_time()
    check = val in where
    end = time.process_time()
    print(type_str[type(where)], check, end - start, flush=True)
```

```
initialize(large_dict, n)
   Dict: 20.498152000000005

initialize(large_list, n)
   List 16.622092000000066
```

Initialization: List is the winner, 20% faster!

```
find_value(n-1, large_dict)
   Dict: True 4.0000001035878086e-06

find_value(n-1, large_list)
   List True 2.794321999999656
```

Initialization: Dict is the winner, one million times faster!

Creation of dictionary variables: use literals

- Empty dictionary:

```
v = {}
```

- Creation of a dictionary with literals:

```
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}

numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}

by_name = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
           'F. Dupont': [17623, 'F', 'France'] }

by_country = {'USA': ['J. Smith', 35672, 'M'], 'Jordan': ['M. Saleh', 27623, 'M'],
              'France': ['F. Dupont', 17623, 'F'] }
```

Creation of dictionary variables: use a list of tuples

- Use a list of tuples and the built-in function `dict(key_val_list)`
  that builds a dictionary directly from *sequences* of input key-value pairs:

```
word_list = [ ('Hello', 5), ('this', 4), ('is', 2), ('a', 1), ('list', 4) ]
word_dict = dict(word_list)

parabola = dict( [(0,0), (0.5, 0.25), (1,1), (1.5, 2.25)] )

v = dict( [ ('USA', [35672, 'M', 'J. Smith']),
            ('Jordan', [27623, 'M', 'M. Saleh']),
            ('France', [17623, 'F', 'F. Dupont']) ] )
```

Creation of dictionary variables: use list of keys with default values

- Use a list of keys and assign a common optional value to the keys by using the method `fromkeys(key_list, <value>)`

```
list_of_words = ["This", "is", "a", "list", "of", "key", "strings"]
dict_of_words = dict.fromkeys(list_of_words, 0)

primes = [2, 3, 5, 7, 11, 13]
dict_of_primes = dict.fromkeys(primes, 'p')
```

Creation of dictionary variables: use of two lists

- Use two lists of the same length, one containing the keys and the other the values, and pair them using the function `zip(key_list, value_list)`

```
list_of_keys = [1, 2, 3, 4, 5, 6]

list_of_values = ['r', 'p', 'p', 'r', 'p', 'r' ]

numbers = dict(zip(list_of_keys, list_of_values))
```

Creation of dictionary variables: cloning and aliasing

- **Cloning**: make a *shallow copy* of the content of a dictionary using method `copy()`

```
new_dict_same_content = numbers.copy()
new_dict_same_content[36] = 'r'
if 36 not in numbers:
    print("Change in the new dictionary didn't affected previous dictionary")
```

- **Cloning**: make a shallow copy of the content of a dictionary using method `copy()`

```
alias_dict = numbers
alias_dict[36] = 'r'
if 36 in numbers:
    print("Change in the new dictionary affected previous dictionary!")
```

Fundamental operations: insert, access, modify

```
accounts = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
            'F. Dupont': [17623, 'F', 'France'] }
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}
```

- Insert a value with a new key:

```
accounts['Y. Honda'] = [75643, 'F', 'Japan']
phone_numbers['Luc'] = 6653
```

- Access the value associated to an existing key:

```
a = accounts['Y. Honda']
```
a is of type list and has value `[75643, 'F', 'Japan']`

```
p = phone_numbers['Paul']
```
`p` is of type `int` and has value `5472`

- Modify the value associated to an existing key:

```
accounts['J. Smith'] = [35672, 'M', 'Canada']
accounts['J. Smith'][2] = 'Qatar'
```

Functions and operators for inspecting a dictionary

```
accounts = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
            'F. Dupont': [17623, 'F', 'France'] }
phone_numbers = {'Ann': 5461, 'Paul': 5472, 'Mark': 3541, 'Liz': 2451}
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- Count all the key-value items present in the dictionary: `len(dictionary)` function

  `len(accounts)` → 3 (int type)

  `len(numbers)` → 6 (int type)

- Get the list with the keys present in the dictionary: `list(dictionary)` function

  `list(accounts)` → `['J. Smith', 'M. Saleh', 'F. Dupont']`

  `list(phone_numbers)` → `['Ann', 'Paul', 'Mark', 'Liz']`

  `list(numbers)` → `[1, 2, 3, 4, 5, 6]` (list type)

- Check whether a key exists or not in the dictionary: membership operators `in`, `not in`

  `3 in numbers` → `True`

  `'Jim' not in phone_numbers` → `True` (bool type)

Methods for inspecting a dictionary: `keys()`

- Get a *dynamic view* on the dictionary keys: `dict.keys()` method, returns a **view object**

```
numbers.keys()    →  dict_keys([1, 2, 3, 4, 5, 6])    (view object)
```

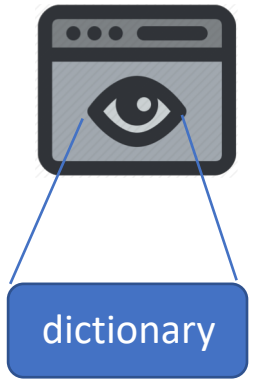
vs.

```
list(numbers)  → [1, 2, 3, 4, 5, 6]                   (list object)
```

→ The keys() method doesn't return a *physical list* with the current keys, as list() does, instead it provides a **view object,** a window view on the dictionary which is dynamically kept up-to-date

- ✓ Save memory
- ✓ If things changes in the dictionary, these can be seen through the view object

view object




```
numbers = {1: 'p', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
keys_now_in_dict = list(numbers)
keys_view = numbers.keys()
numbers[13] = 'p'
print("Is 13 in dict? From static list copy:", (13 in keys_now_in_dict) )
print("Is 13 in dict? From dynamic view:", (13 in keys_view) )
```

Methods for inspecting a dictionary: `values()`, `items()`

- Get a *dynamic view* on the dictionary values : `dict.values()` method, returns a **view object**

```
numbers.values()  →  dict_values(['p', 'p', 'p', 'r', 'p', 'r'])
```

- Get a *dynamic view* on the entire dictionary: `dict.items()` method, returns a **view object**

```
numbers.items()  → dict_items([(1, 'p'), (2, 'p'), (3, 'p'),
                               (4, 'r'), (5, 'p'), (6, 'r')])
```

Methods for inspecting a dictionary: iterations

- Iterate over all dictionary values :

```
for k in numbers:
  print('Key:', k)

for i in numbers.items():
  print('Pair (key, value):', i[0], i[1])
```

Observations:

- A dictionary is "*identified*" by its collection of keys, this is why directly iterating over the dictionary in the first example is in practice equivalent to iterate over the keys, that are the returned sequence values
- Iterations over `dict.items()` return the pairs (key, value) as tuples, where the key has index 0 and the value has index 1

Methods for accessing and modifying a dictionary: `get()`

- Get the value for a specified key if key is in dictionary: `dict.get(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. The `value` parameter is optional. If `value` is not passed, `None` is returned.

```
key = 3
x = numbers.get(key)
if x != None:
    print('Value associated to key', key, 'is:', x)
```

- Advantage over the use of the `[]` operator:

```
x = numbers[key]
```

→ Throws an Error if key is not in the dictionary!

Methods for accessing and modifying a dictionary: pop( )

- Remove and return dictionary element associated to passed key: `dict.pop(key, <value>)`

  `key` is the key to be searched, and `value` is the value to return if the specified key is not found in the dictionary. If `value` is not passed, an error is thrown in the case `key` is not in the dictionary

```
key = 3
x = numbers.pop(key, None)
if x != None:
   print('Removed pair (', key, ':', x, ')')
```

- Advantage over the use of `del` and `[ ]` operators:

```
key = 11
del numbers[key]
```

→ Throws an Error since the selected key is not in the dictionary!

Methods for accessing and modifying a dictionary: `popitem()`

- Remove and return the last inserted dictionary element: `dict.popitem()`

  A pair (key, value) is removed from the dictionary following a LIFO order (last-in, first-out). The removed pair is returned as a tuple

```
x = numbers.popitem()
if len(numbers) > 0:
    print('Removed the last inserted key-value pair (', x, ')')
    print('New size of the dictionary:', len(numbers))
```

Methods for accessing and modifying a dictionary: `clear()`

- Remove *all* elements from a dictionary element: `dict.clear()`

  All elements are removed, no values are returned, after the call `dict` is equivalent to `{}`

```
numbers.clear()
print('Removed all elements')
```

Methods for accessing and modifying a dictionary: `update( )`

- Update the dictionary with the elements from the another dictionary object (or from an iterable of key/value pairs (e.g., tuple)): `dict.update([other])`

  Takes as input a dictionary (or an iterable of tuples) and use the input to update `dict`

```
some_primes = {13:'p', 17:'p', 23:'p'}
numbers.update(some_primes)
print('Updated dictionary:', numbers)

new_entry= {12:'r'}
numbers.update(new_entry)
print('Dictionary updated with a new single entry')

l = [(10, 'r'), (12, 'r')]
numbers.update(l)
print('Dictionary updated with a list of entries')
```

Methods for accessing and modifying a dictionary: `setdefault()`

- Insert a new (key, value) pair only if key doesn't already exist, otherwise return the current value: `dict.setdefault(key, <value>)`

  The function aims to *update* the dictionary with a new (`key, value`) pair only if the given `key` is not already in the dictionary, otherwise the dictionary (i.e., the existing value of `key`) is *not updated* and the current value associated to the specified `key` is returned instead. If `value` is not passed as input, the default `None` value is used.

```
val = numbers.setdefault(30, 'r')      # key 30 doesn't exist, pair (30,'r') is inserted in numbers
val = numbers.setdefault(30, 'rr')     # key 30 it exists now, its value isn't updated, val gets 'r'


new_dict = {}
for i in range(10):
    new_dict.setdefault(i)     # new_dict gets initialized with 10 keys and None values
```

Relational and arithmetic operators for dictionaries

```
accounts = {'J. Smith': [35672, 'M', 'USA'], 'M. Saleh': [27623, 'M', 'Jordan'],
            'F. Dupont': [17623, 'F', 'France'] }
numbers = {1: 'r', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
```

- == operator: check whether two dictionary are the same, same (key , value) pairs

```
x = accounts == numbers   → False

accounts2 = accounts.copy()
x = accounts == accounts2   → True
```

- Other relational operators `>, >=, <, <=` do not apply to dictionary operands
- Arithmetic operators do not apply to dictionary operands

Useful operations on key and value sets: `sorted()`, `sort()`

- Get the **sorted list of keys** from the dictionary items:

```
sorted_keys = sorted(numbers) → [1, 2, 3, 4, 5, 6]
sorted_keys = sorted(numbers.keys())
```

- Get the **sorted list of values** from the dictionary items:

```
sorted_values = sorted(numbers.values()) →  ['p', 'p', 'p', 'p', 'r', 'r']
```

- Equivalent way, using the `list()` function:

```
keys_to_be_sorted = list(numbers.keys())
keys_to_be_sorted.sort()
```

- Get the **sorted list of keys, paired with their associated values** :

```
sorted_dict_list = sorted(numbers.items())
                  → [(1, 'p'), (2, 'p'), (3, 'p'), (4, 'r'), (5, 'p'), (6, 'r')]
```

Useful operations on key and value sets

- Watch out! The `sorted()` function and the `sort()` method could have been used without a comparison function given that in these example all keys / values are homogeneous (`int` or `str`) and python knows how to perform comparisons among these homogeneous data types

- In the general case, the use of sort functions/methods might require the additional definition of a comparison function, based on the characteristics of the keys / values to sort

- This applies also to `min(), max(), sum()`

Useful operations on key and value sets: `min()`, `max()`, `sum()`

- Find **min / max** of key/values from the dictionary items:

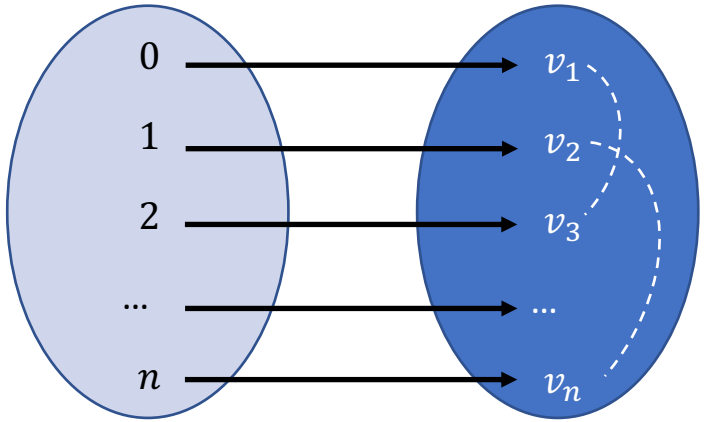
```
max_key_val = max(numbers)      → 6
min_key_val = min(numbers)      → 1
max_key_val = max(numbers.keys())      → 6
min_key_val = min(numbers.keys())      → 1

max_values = max(numbers.values())   → r
min_values = min(numbers.values())   → p
```

- Find **sum** of key/values from the dictionary items:

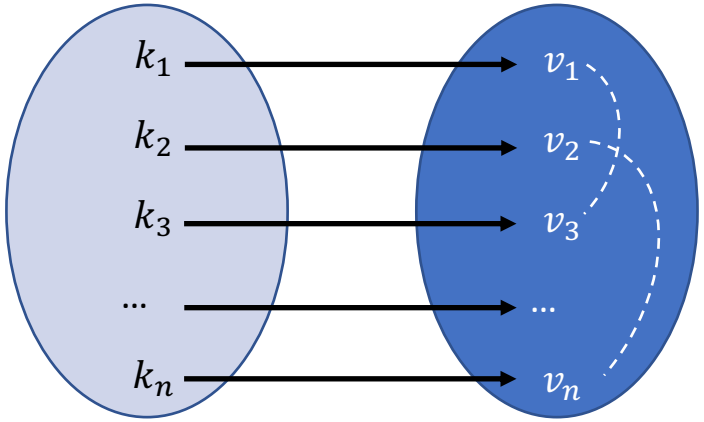
```
key_sum = sum(numbers) → 34
key_sum = sum(numbers.keys()) → 34
values_sum = sum(numbers.values()) → Error, sum not defined over strings!
```

Sets: unordered, unhashed, non-scalar data structures

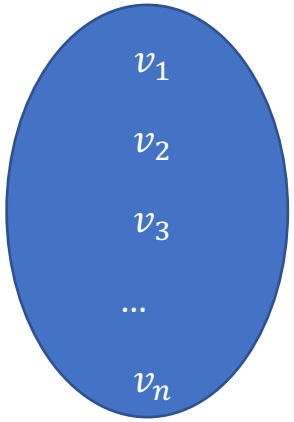
- **Set data structure**: unordered collection of items where every element is unique (no duplicates) and must be immutable
- The set itself is mutable: elements can be inserted and removed, aliases between sets can be created



Sequences
(Lists, tuples)



Dictionaries



Sets

Creation of a set

- Use of { } brackets with literals:

```
new_set = {1, 2, 3, 4.5, 5.3, True, (1,2), 'Hello'}
print(new_set, type(new_set))
```

✓ Any mixed collection of immutable types is admitted

```
new_set = {1, 2, 3, [1,2]}
```
→ Error, the list [1,2] is a mutable object

✓ The set variable itself is mutable

Creation of a set: aliasing (=) and cloning (copy() method)

- ✓ **Aliasing** (same content, same identity):

```
A = {1, 2, 3}
B = A
B.add(11)
print(A,B)  #Output: {1,2,3,11} {1,2,3,11}
```

- ✓ **Cloning** (shallow copy):

```
A = {1, 2, 3}
B = A.copy()
B.add(11)
print(A,B)  #Output: {1,2,3} {1,2,3,11}
```

Creation of a set

- Use of built-in function `set()` to create a new set from an *iterable* (string, list/tuple, dictionary, set): each element of the iterable object defines an element of the set, **duplicates are discarded**

```
l = [1, 2, 3, 4.5, 5.3, True, (1,2)]     # set from a generic list
new_set = set(l)
print(new_set, type(new_set))


numbers = {1: 'p', 2: 'p', 3:'p', 4:'r', 5:'p', 6:'r'}
new_set = set(numbers)                   # set from a dictionary using the keys, set is {1, 2, 3, 4, 5, 6}
new_set = set(numbers.values())          # set from a dictionary using the values, set is {'p','r'}

new_set = set("apple")                   # set from a string, new_set is {'e', 'a', 'l', 'p'}
new_set = set(["apple"])                 # set from a list of strings, new_set is {'apple'}
empty_set = set()                        # empty set
```

Changing a set: add(), update()

- ✓ No indexing, no hashing make sense in sets

- Add single elements with method `add()`:

```
my_set = {1,3,5}
my_set.add(2)
my_set.add(1)
```

→ no error is thrown but no duplicate element is inserted since 1 in already in set

- Add multiple elements with method `update(iterable)` that takes as input iterables (strings, tuples/lists, dictionaries, sets) and add each element into the set, no duplicates are inserted

```
my_set = {1,3,5}
my_set.update([2,3,4])          # my_set now contains {1,2,3,4,5}
my_set.update({2,2,2})          # my_set is left unchanged, it still contains {1,2,3,4,5}
my_set.update({'c':1, 'b':2}) # my_set now contains {1, 2, 3, 4, 5, 'b', 'c'}
my_set.update("apple")          # my_set now contains {1, 2, 3, 4, 5, 'b', 'c', 'e', 'a', 'l', 'p'}
```

Changing a set: discard(), remove ()

- Remove single element, no error is thrown if the element isn't there: `discard()`

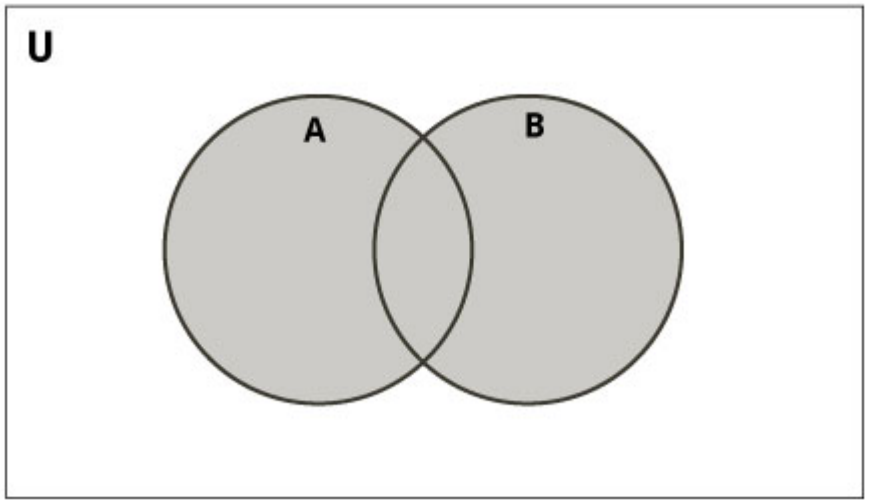
```
my_set = {1,3,5}
my_set.discard(3)    # my_set is now {1,5}
my_set.discard(4)    → no error is thrown, set stays unchanged
```

- Remove single element, an error is thrown if the element isn't there: `remove()`

```
my_set = {1,3,5}
my_set.remove(3)    # my_set is now {1,5}
my_set.remove(4)    → an error is thrown!
```

Operations with sets: Union

- Sets are mathematical objects, and we can perform the usual mathematical operations on them: **union, intersection, difference, symmetric difference**

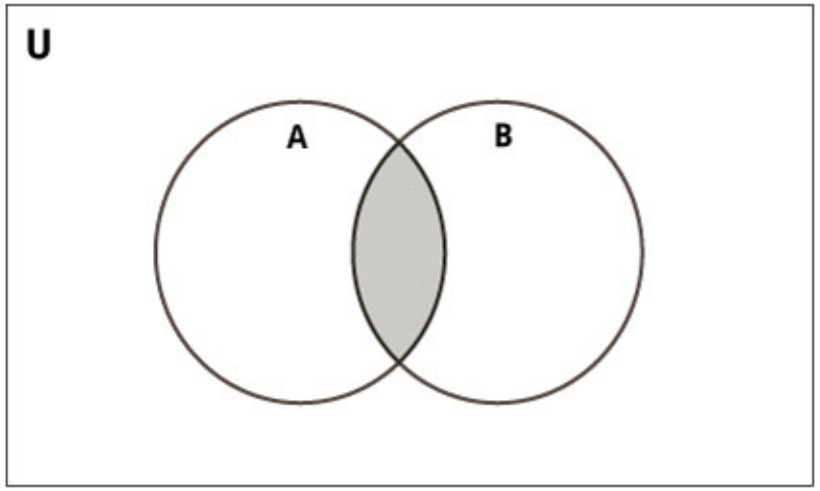


- **Union**, operator |
- Union method: `C = A.union(B)`

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
C = A.union(B)

print(A | B, C == (A|B))
#Output: {1, 2, 3, 4, 5, 6, 7, 8}  True
```

- In-place union (update / modify A):
  - `A |= B`
  - `A.update(B)` (returns None)

Operations with sets: Intersection

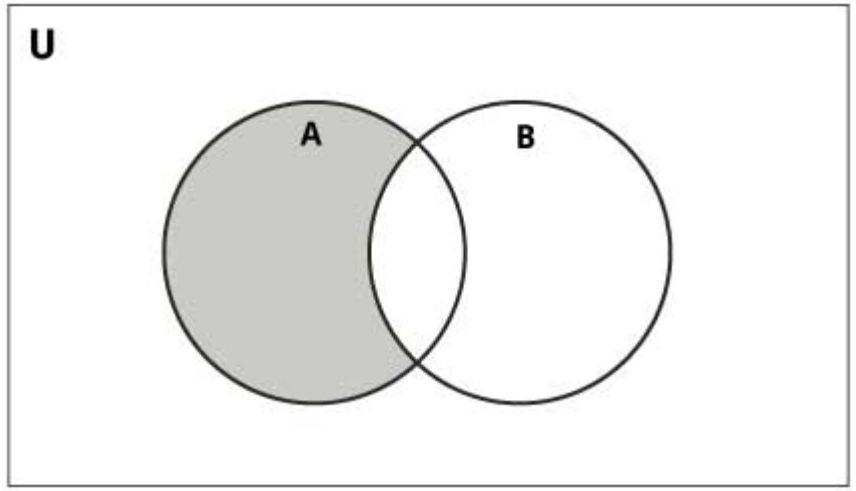


- **Intersection**, operator `&`
- Intersection, method: A.intersection(B)

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
C = A.intersection(B)

print(A & B, C == (A&B))
#Output: {4, 5} True
```

- In-place intersection (update / modify `A`):
  - `A.intersection_update(B)` (returns `None`)
  - `A &= B`

Operations with sets: Difference

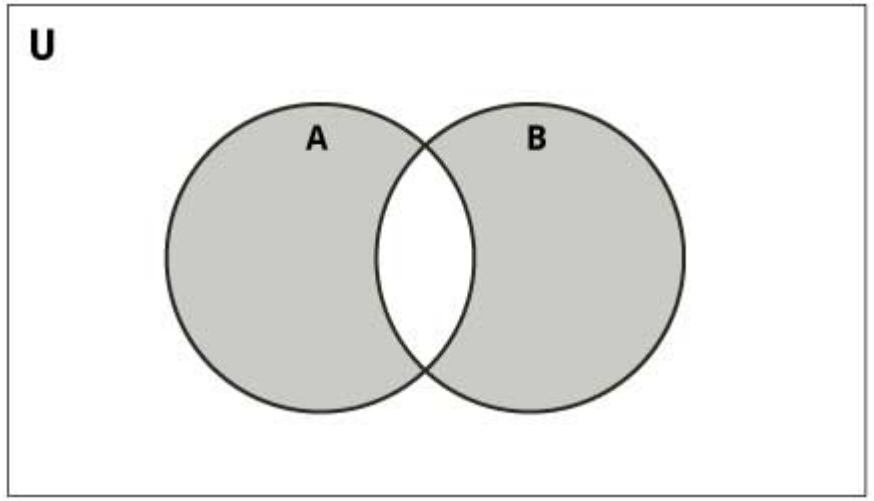


- **Difference**, operator –
- Difference method: A.difference (B)

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
C = A.difference (B)

print(A - B, C == (A-B))
#Output: {1, 2, 3} True

print(B – A, C == (B-A))
#Output: {6, 7, 8} False
```

- In-place difference (update / modify A):
  - `A.difference_update(B)` (returns None)
  - `A -= B`

Operations with sets: Symmetric difference



- **Symmetric difference**, operator ^
- Difference method: A.symmetric_difference (B)

```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
C = A.symmetric_difference(B)

print(A ^ B, C == (A^B))
#Output: {1, 2, 3, 6, 7, 8} True
```

- In-place difference (update / modify A):
  - `A.symmetric_difference_update(B)` (returns None)
  - `A ^= B`

(Regular) Operations with sets: membership, iteration

- **Membership:** `in` operator

```
my_set = set("apple")      # my_set is {'e', 'a', 'l', 'p'}
in_set = 'a' in my_set     # in_set is True
in_set = 2 in my_set       # in_set is False
```

- **Iteration:**

```
my_set = set("apple")
for letter in my_set:          # for loop with letter taking the values in {'e', 'a', 'l', 'p'}
    print(letter)
```

Relational operators

- **Equality: `==`, `!=`**

```
A == B     # returns True if the two sets are equal in content, False otherwise
A != B     # returns True if the two sets have some differences, False otherwise
```

- **Subset of: `<=`, `A.issubset(B)`**

```
A <= B             # returns True if A is a subset of B, False otherwise
A.issubset(B)      # returns True if A is a subset of B, False otherwise
```

- **Superset of: `>=`, `A.issuperset (B)`**

```
A >= B             # returns True if A is a superset of B, False otherwise
A.issuperset(B)    # returns True if A is a superset of B, False otherwise
```

- **Strictly subset or superset: `>`, `<`**

```
A > B              # returns True if A >= B and A != B, False otherwise
A < B              # returns True if A <= B and A != B, False otherwise
```

Useful built-in functions that can be used with sets

Function	Description
all()	Return `True` if all elements of the set are true (or if the set is empty).
any()	Return `True` if any element of the set is true. If the set is empty, return `False.`
enumerate()	Return an enumerate object. It contains the index and value of all the items of set as a pair.
len()	Return the length (the number of items) in the set.
max()	Return the largest item in the set.
min()	Return the smallest item in the set.
sorted()	Return a new sorted list from elements in the set(does not sort the set itself).
sum()	Retrun the sum of all elements in the set.

List of all methods for sets

Method	Description
add()	Adds an element to the set
clear()	Removes all elements from the set
copy()	Returns a copy of the set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from the set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns `True` if two sets have a null intersection

Use the help() function to get complete descriptions inline

issubset()	Returns `True` if another set contains this set
issuperset()	Returns `True` if this set contains another set
pop()	Removes and returns an arbitary set element. Raise `KeyError` if the set is empty
remove()	Removes an element from the set. If the element is not a member, raise a `KeyError`
symmetric_difference()	Returns the symmetric difference of two sets as a new set
symmetric_difference_update()	Updates a set with the symmetric difference of itself and another
union()	Returns the union of sets in a new set
update()	Updates the set with the union of itself and others

https://docs.python.org/3/library/stdtypes.html - set-types-set-frozenset