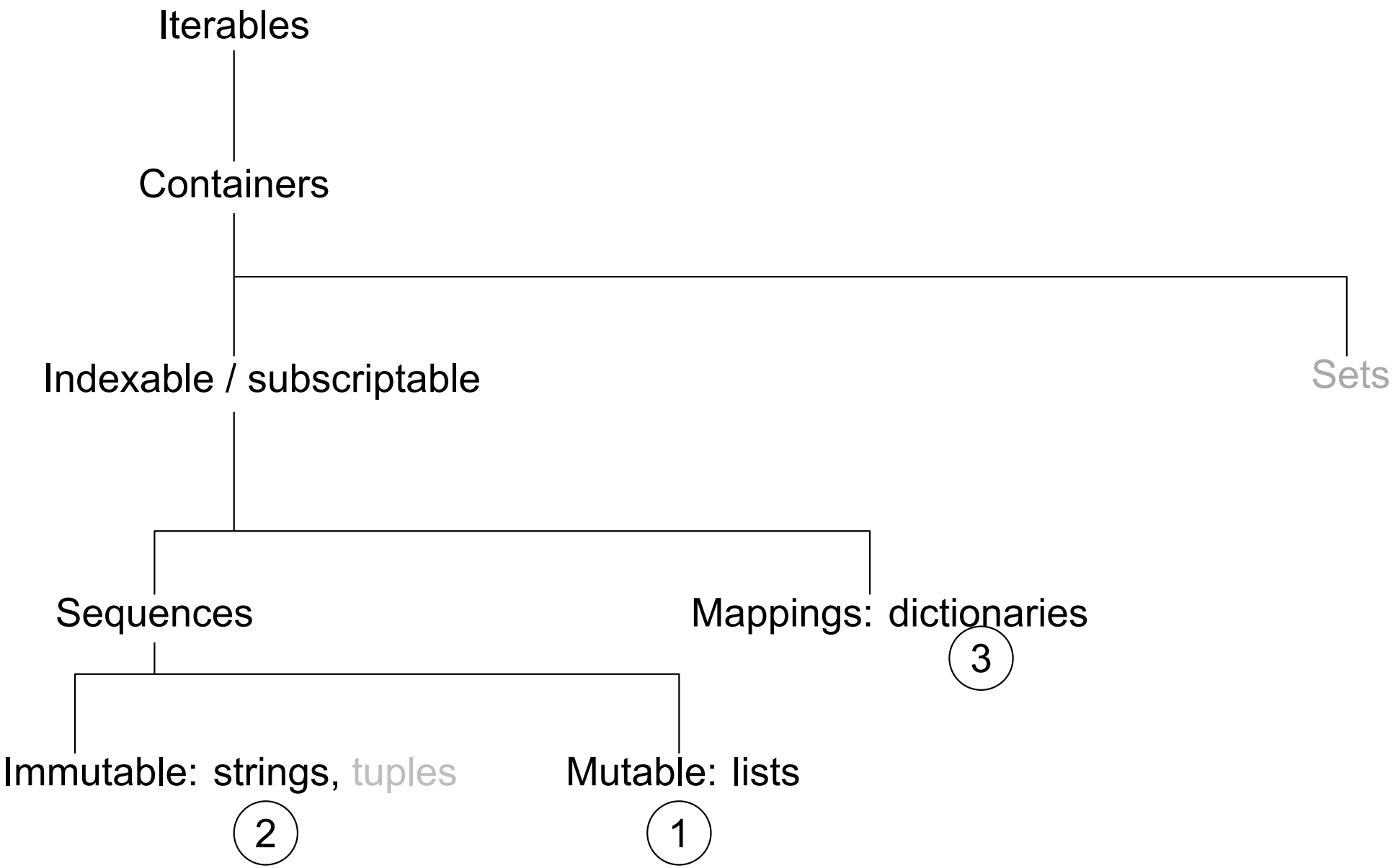
python: container types (lists, strings, dictionaries, sets)

intro. to computing

`https://mandarm.github.io/computing.html`

container types



container types

- lists: mutable sequence
  - elements accessed using *indices*, and **can be changed**
  - creating lists

```
l = [ 11, "y", 7.4 ]
l = []      # empty list
l = list(range(0,100,10)) # generally, list(iterable)
numbers = [10, 16, -2, 22, -98]
```

later: using list comprehension

- strings (`str`) : immutable sequence
  - elements accessed using *indices*, but **cannot be changed**
- dictionaries `{ 1:11, 2:"y", 3:7.4, 3.14:"pi" }`
  - elements accessed using *keys* (must be distinct)
- tuples `(11, "y", 7.4)`
- sets `{ 11, "y", 7.4 }` — *not "subscriptable" (indexable)*

lists: operations

(applicable to immutable sequences also)

operation	result
`x in s`	true if an item of `s` is equal to `x`, else false***
`x not in s`	false if an item of `s` is equal to `x`, else true***
`s + t`	concatenation of `s` and `t`, does not change `s`, `t`
`s * n` or `n * s`	equivalent to concatenating `s` `n` times††
`len(s)`	length of `s`

*** for strings, `x` can also be a substring, e.g.,

```
"gg" in "eggs" "ana" in "banana"
```

†† **later:** may lead to unexpected behaviour **exercise:** try using a negative value for `n`.

lists: indexing and slicing

indexing

negative indexes

`s` =	`-n`	`-(n+1)`	`...`	`-2`	`-1`
	`[ 10,`	`20,`	`...`	`40,`	`50, ]`
	`0`	`1`	`...`	`n-2`	`n-1`

positive indexes

note: $-0 \equiv 0$

slicing: `s[start:end:step]`

- `start`: starting index, **default** `0`, *included* in slice
- `end`: ending index, **default** `len(s)`, *excluded* from slice
- `step`: skip `step` elements (`step` must be non-zero)
  - if `step` is positive, and `start >= end`, slice is empty.
  - if `step` is negative,
    - `start` must be greater than `end`; (`start <= end` $\rightarrow$ slice is empty)
    - elements are listed in reverse order.
      example: `[22, 16, 10, -2, -98][10:0:-1]`
      $\rightarrow$ `[-98, -2, 10, 16]`

[correction] list slicing: more about `start` and `end`

how 'unusual' values of `start`, `end` are handled

1. if `start` (or `end`) is negative, it is replaced by `len(l) + start` (or `len(l) + end`).
2. if `start` or `end` are omitted, `none`, or *out of range*, they are replaced by appropriate ***end values***.

end values: `0`, or `-1`, or `len(s)`, or `len(s)-1` as appropriate, depending on the sign of `step` (see examples below)

examples:

- `l[n:0:-1]` prints elements in reverse order, starting from `n` (or `len(l)-1` if `n >= len(l)`), upto and **excluding** first element (index `0`)
- `l[n:-1:-1]` (`n >= len(l)`) prints empty list
  - `end` (`-1`) replaced by `len(l)-1` (rule 1)
  - either `n <= len(l)-1`, or is replaced by `len(l)-1`
- `l[n : -len(l)-1 : -1]` prints elements in reverse order, starting from `n` (or `len(l)-1` if `n >= len(l)`), upto and **including** first element
  - `end` (`-len(l)-1`) replaced (**once!**) by `-1` (rule 1)
- `l[::-1]` prints all elements in reverse order

-1 written by user is different from -1 obtained by replacement!

lists: more operations

(applicable to mutable sequences only)

operation	result
`s[i] = x`	item `i` of `s` is replaced by `x`
`del s[i]`	removes item `i` of `s`
`s[i:j] = t`	slice of `s` from `i` to `j` is replaced by the contents of `t`
	`i==j` $\rightarrow$ contents of `t` inserted in place of *empty slice* at index `i`
`del s[i:j:k]`	removes the elements of `s[i:j:k]` from the list, `:k` optional
	same as `s[i:j:k] = []`
`s[i:j:k] = t`	elements of `s[i:j:k]` are replaced by those of `t`
	`:k` optional, but if not 1, `s[i:j:k]` and `t` must have same length
`s += t`	extends `s` with contents of `t`
`s *= n`	updates `s` with its contents repeated `n` times

lists: some useful functions I

(applicable to immutable sequences also)

- `s.count(val)`: number of occurrences of `val` in sequence `s`
- `s.index(val)`
  `s.index(value, start)`
  `s.index(value, start, stop)`
  index of the **first** occurrence of `value` in sequence `s`, but error if it does not occur

later: exceptions (errors) and how to handle them

usage:

```
s = [ 0, 1, 2, 3, 2, 0, 1, 0, 3, 2]
s.count(0)
[ 0, 1, 2, 3, 2, 0, 1, 0, 3, 2].count(3)
```

lists: some useful functions II

(applicable to mutable sequences only)

- `s.append(value)`
- `s.insert(index, value)`
  - if `index` is out of range, it is replaced by appropriate end value (as for slicing)

    example:

```
>>> numbers = [16, 10]
>>> numbers.insert(-10,100) # numbers == [100, 16, 10]
>>> numbers.insert(100,200) # numbers == [100, 16, 10, 200]
```

- `s.pop(index)`: `index` optional, **default** is -1
  - get the item at `index` and remove it from the sequence
  - `index` *must be valid*, i.e., between `-n` and `n-1` (both inclusive)

lists: some useful functions III

- `s.remove(value)`: remove the first item that matches `value`, but error if it does not occur
- `s.extend(sequence)`
- `s.reverse()`, `reversed(s)`: **note the difference!**

lists: more useful functions

if the elements can be compared

- `min(l)` smallest item of `l`
- `max(s)` largest item of `s`
- `sort()`, `sorted()`
- `sum()` for numeric types

```
numbers = [10, 16, -2, 22, -98]
sorted(numbers)     # [-98, -2, 10, 16, 22]
reversed(numbers)   # <list_reverseiterator object at ...>
numbers.sort()      # changes in place
numbers.reverse()   # changes in place
```

note the difference between `sorted()` (returns a list) and `reversed()` (returns an iterable).

Strings (`str` type)

Creating strings

- Single, double, triple quotes
- `str()` function applied to any variable or expression
  (creates printable string representation of the variable / expression)
- Two adjacent string literals separated by only whitespace automatically joined into single string literal

```
s1 = 'allows embedded "double" quotes'
s2 = "allows embedded 'single' quotes"
s3 = '''Three single
(allows multiple embedded newlines and ") quotes'''
s4 = """Three double
(allows multiple embedded newlines and ') quotes"""
s5 = str(['P', 'y', 't', 'h', 'o', 'n'])   # NOT "Python"
s6 = str([1, 2, 3, 4])
s7 = str({"sky" : "blue", "grass" : "green", "blood" : "red"})
s8 = "hello" ' world'
```

Digression: escape sequences, 'raw' strings

```
>>> s4
"Three double \n allows multiple embedded newlines and ') quotes"
>>> print(s4)
>>> s = 'hi\n'
>>> print(s)
>>> len(s)
```

Escape sequences: character sequences starting with a backslash (\) that denote some other character that may be hard to type / represent directly

Examples:

- `\t` ≡ tab (somewhat useful for lining things up in columns)
- `\r` ≡ carriage return (takes cursor back to beginning of line)
  (try `print("hello\nworld")` and `print("hellooooooo\rworld")`)

Raw strings

- strings prefixed with `r`, e.g., `r"hi\n"`
- escape sequences treated literally, do **NOT** have special meaning
  (try `print(r"hello\nworld")` and `print(r"hellooooooo\rworld")`)

lists: operations

(applicable to immutable sequences also)

operation	result
`x in s`	true if an item of `s` is equal to `x`, else false***
`x not in s`	false if an item of `s` is equal to `x`, else true***
`s + t`	concatenation of `s` and `t`, does not change `s`, `t`
`s * n` or `n * s`	equivalent to concatenating `s` `n` times††
`len(s)`	length of `s`

*** for strings, `x` can also be a substring, e.g.,

```
"gg" in "eggs" "ana" in "banana"
```

†† **later:** may lead to unexpected behaviour **exercise:** try using a negative value for `n`.

String functions: character types

Digression: non-English languages and Unicode

- `s.isalpha()`: all characters in `s` are letters,

  Examples: `'Letters and spaces'.isalpha()`, `'LettersOnly'.isalpha()`
- `s.islower()`: all characters in `s` are lowercase
- `s.isupper()`: as above
- `s.isdigit()`, `s.isnumeric()`
- `s.isalnum()`
- `s.isspace()`: blanks, tabs, newlines, etc.

`s` must have at least one character of the desired type

String functions: changing case

Strings are **immutable**; all functions return a modified copy.

- `s.upper()`: all *cased* characters are converted to uppercase

  Question: Is `s.upper().isupper()` guaranteed to return `True`?
- `s.lower()`: as above
- `s.swapcase()`: uppercase characters are converted to lowercase and vice versa
- `s.capitalize()`: first character is capitalised, rest is lowercased

String functions: searching

`end`, or both `start` and `end` may be omitted

- `in` operator, e.g., `"ana" in "banana"`
- `s.find(substring, start, end)`: returns lowest index in the string where substring `sub` is found, -1 if `sub` is not found

  Question: What is the difference between `find` and `index`?
- `s.rfind(substring, start, end)`: as above, but returns highest index
- `s.startswith(prefix, start, end)`
  - possible to check whether `s` with any of a number of prefixes by using a *tuple* of strings in place of `prefix`
- `s.endswith(suffix, start, end)`: as above
- `s.count(sub, start, end)`: returns number of *non-overlapping* occurrences of `sub` in `s` (or appropriate slice of `s`)

  Question: What happens when `sub` is the empty string (`""`)?

String functions: replacing

- `s.removeprefix(somestring)`: if the string starts with `somestring`, remove it from the returned copy, otherwise return the original string `s`
- `s.removesuffix(somestring)`: as above
- `s.strip()` or `s.strip("some string")`: any leading / trailing characters that are in the specified string is removed

  (if no arguments are provided, leading and trailing *whitespace* is removed)

  Examples: `"  abcd  ".strip()`, `"  abcd  ".strip("dcba")`, `"ad  abcd  bc".strip("dcba")`
- `s.lstrip()`, `s.rstrip()`: as above, but only leading (`lstrip`) OR trailing (`rstrip`) characters are removed

  Question: What is the difference between ⟨`lstrip()`, `removeprefix()`⟩and ⟨`rstrip()`, `removesuffix()`⟩?
- `s.replace(oldstring, newstring, count)`: occurrences of `oldstring` are replaced by `newstring`
  - if `count` (optional) is not specified or negative, all occurrences are replaced; otherwise only the first `count` occurrences are replaced

  Question: What happens if `oldstring` is `""`?

String functions: splitting I

- `s.partition(seperator-string)`: split the string at the *first* occurrence of `separator-string`; return a tuple containing 3 strings: (a) part before the separator, (b) separator, (c) part after the separator.
  - separator not found $\Rightarrow$ tuple contains `s`, `""` and `""`
- `s.rpartition(seperator-string)`: as above, but use the *last* occurrence of `separator-string`

Digression: tuples

- Like lists, but *immutable*
- Use round brackets `( )` instead of square brackets `[ ]`

```
>>> t1 = ('a', 'b')
>>> t2 = tuple([1, 2, 3])
>>> t3 = (t1[1], t2[0])
```

String functions: splitting II

- `s.split(seperator-string)` can specify maximum number of pieces
  Return list of "words" obtained by splitting `s` wherever `seperator-string` (must be non-empty) appears

  Examples:

```
'1,,2,3'.split(',')
# ['1', '', '2', '3']   (empty string because of consecutive commas)

'1,2,,3,'.split(',')    # ['1', '2', '', '3', '']
'1<>2<>3<4'.split('<>') # ['1', '2', '3<4']

# maxsplit=1 means at most 2 pieces
'1,2,3'.split(',', maxsplit=1)  # ['1', '2,3']
```

  Question: What happens is `separator-string` does not occur in `s`?

  Question: What happens if `s` is the empty string?

String functions: splitting III

- `s.split()` or `s.split(None)`
  - return list of "words" obtained by splitting `s` wherever *one or more whitespace characters* appear(s)
  - *leading or trailing whitespace is eaten up*

Examples:

```
'1 2 3'.split()           # ['1', '2', '3']
'   1   2   3   '.split()  # ['1', '2', '3']
'1 2 3'.split(maxsplit=1)  # ['1', '2 3']
```

Homework: find out about `s.splitlines()`

String functions: joining

`s.join(iterable)`: return string formed by concatenating elements of iterable (must be strings) in iterable, with `s` inserted between successive elements

```
>>> ' followed by '.join(['abc', 'def', 'ghi'])
>>> '-'.join('Python')   # 'P-y-t-h-o-n'
```

Homework: String functions related to formatting

- `s.zfill(width)`
- `s.ljust(width, fillchar)`,
  `s.rjust(width, fillchar)`,
  `s.center(width, fillchar)`
- `s.format()`

Dictionaries (`dict` type)

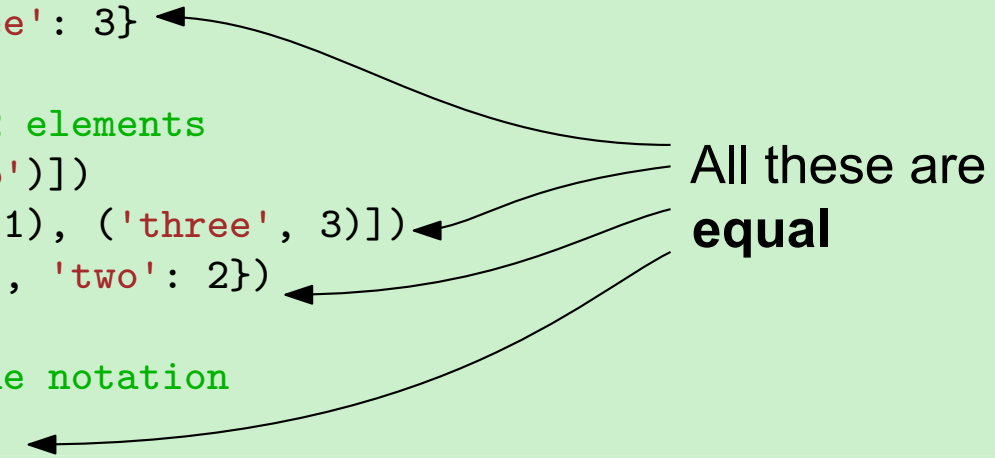
Dictionaries / mappings

- Set of *key-value* pairs
- Keys must be hashable
  - numbers, strings allowed as keys
  - lists, dictionaries not allowed as keys
- Keys need not be of same type
- Keys have to be distinct
- Creating dictionaries

```
d1 = { 2026: "1st year",  2025: "2nd year", 2024: "3rd year"}
d2 = { "1st year": 2026,  "2nd year": 2025, "3rd year": 2024}
d3 = {'one': 1, 'two': 2, 'three': 3}

# each item must have exactly 2 elements
d4 = dict([(1, 'jan'), (2, 'feb')])
d5 = dict([('two', 2), ('one', 1), ('three', 3)])
d6 = dict({'three': 3, 'one': 1, 'two': 2})

# also possible to use key=value notation
a = dict(one=1, two=2, three=3)
```

Dictionaries

```
d1 = { 2026: "1st year",  2025: "2nd year", 2024: "3rd year"}
d2 = { "1st year": 2026,  "2nd year": 2025, "3rd year": 2024}
d3 = {'one': 1, 'two': 2, 'three': 3}

# each item must have exactly 2 elements
d4 = dict([(1, 'jan'), (2, 'feb')])
d5 = dict([('two', 2), ('one', 1), ('three', 3)])
d6 = dict({'three': 3, 'one': 1, 'two': 2})

# also possible to use key=value notation
a = dict(one=1, two=2, three=3)
```



- Two dictionaries with same (key, value) pairs (regardless of ordering) are **equal** (==).
- Ordering comparisons between dictionaries (<, <=, >=, >) are invalid.

Accessing values

- Use *key* as index to access *value*

```
print(d1[2026], d1[2024], d2["2nd year"])
print(d4[1])
```

- Key must exist in the dictionary (as a key)
- Values that compare equal (such as 1, 1.0, and True) can be used interchangeably to index the same dictionary entry.

Exercise: Try the following.

```
>>> d3[1]
>>> d4[1.0]
>>> d4[True]
>>> d1 = { 2026: "1st year",  2026: "2nd year", 2024: "3rd year"}
>>> d3 = dict([(1, 'jan'), (2, 'feb'), (1, 'January'])
```

Basic operations/functions I

- `d.get(key, default-value)`: like `d[key]` but never returns error
  - `default-value` is optional
  - if `key` is not in `d`, return `default-value`, or `None` if default is not provided
- `d[key] = value` (`key` does not have to be an existing key in `d`)
- `d.setdefault(key, default-value)`
  - if `key` is in `d`, return its value;
  - otherwise, insert `key` with `default-value` and return `default-value`
  - if `default-value` (optional) is not provided, use `None`

Basic operations/functions II

- `d.pop(key)`, `d.pop(key, default)`
  - if `key` is in dictionary, *remove* it and return its value
  - if `key` is not in dictionary, return `default`
  - if `key` is not in dictionary, and `default` is not given → ERROR
- `d.popitem()`: remove and return a (key, value) pair from `d` (in Last In First Out (LIFO) order)
- `del d[key]`: remove entry corresponding to `key` (*must exist!*) from `d`
- `d.clear()`: remove *all* entries in `d`

- `key in d`, `key not in d`, `not key in d`
- `list(d)`: return list of all *keys* currently in `d`
- `len(d)`: number of items in the dictionary `d`

Dictionaries: ‘bulk’ updates

- `d.update(key1=value1, key2=value2, ...)`
  `d.update(dict2)`
  `d.update(iterable)` must contain tuples / other iterables of length two

  Add given key/value pairs to the dictionary `d` (existing keys are updated)
- `d1 | d2`: create new dictionary by merging dictionaries `d1` and `d2` (for common keys, values in `d2` are retained)
- `d1 |= d2`: similar
  - `d1` is updated
  - `d2` may be dictionary or iterable (as for `update`)

Dictionary views I

Dictionary view object: provide a *dynamic* view on the dictionary's entries, i.e., when the dictionary changes, the view is **auto-updated**.

Functions:

- `d.keys()`: return new view of the dictionary's keys
- `d.values()`: return **new** view of the dictionary's values
- `d.items()`: return **new** view of the dictionary's *items*, i.e., (key, value) pairs
- `len()` and membership (`in`) work for dictionary views

Dictionary views II

```
d1 = { 2026: "1st year",  2025: "2nd year", 2024: "3rd year"}
k1 = d1.keys()
v1 = d1.values()
e1 = d1.items()
# Print k1, v1, e1

d1[2027] = "Next incoming batch"
# Print k1, v1, e1 again: automatically get updated values
del d1[2027]
# Print k1, v1, e1 again: automatically get updated values

2026 in k1
(2024, "3rd year") in e1
```

Dictionary views III

Warning: counterintuitive property

```
>>> k2 = d1.keys()
>>> k1 == k2
True
>>> v2 = d1.values()
>>> v1 == v2
False
>>> e2 = d2.items()
>>> e1 == e2
False
```

Dictionaries: iteration and order of items I

```
for k in d1
for k in d1.values()
for k in reversed(d1)
for k in reversed(d1.items())
```

etc.

} work as expected

- Dictionaries preserve insertion order.
- Updating a key does not change its position.
- Keys added after deletion are inserted at the end.
- Dictionaries (also views) are reversible.

Dictionaries: iteration and order of items II

```
d = {"one": 1, "two": 2, "three": 3, "four": 4}
list(d)                 # ['one', 'two', 'three', 'four']
list(d.values())        # [1, 2, 3, 4]
d["one"] = 42
del d["two"]
d["two"] = None
# Check d.keys(), d.values(), etc. again
list(reversed(d))
list(reversed(d.values()))
```

Sets

Sets

- *Unordered* collection $\Rightarrow$ no indexing, slicing, sorting
- Elements have to be *hashable* and *distinct*
- Intended uses: removing duplicates from a sequence + set operations
- `set` type $\longrightarrow$ mutable, *not hashable*
  - cannot be used as dictionary key or element of another set
- `frozenset` type $\longrightarrow$ immutable, hashable
- Creating sets

```
S1 = {'BStat', 'BSDS', 'BMath'} # not permitted for frozenset
S2 = set()                      # empty set
S3 = frozenset({'BStat', 'BSDS', 'BMath'}) # OK, also frozenset(S1)
S4 = set('Indian Statistical Institute')
S5 = set(['MStat', 'MMath', 'MSQE'])
S6 = set([S1])      # Try removing [ ]
S7 = set([ [1,3,5], [2,4,6] ]) # Try removing inner [ ]
S8 = set([S3])      # Try removing [ ]
```

Basic operations

- Membership: `x in S`, `x not in S`,
- Cardinality: `len(S)`
- Iteration: `for x in S`

Operations on sets, frozensets I

- `S1.isdisjoint(S2)`
- `S1.issubset(S2)`, `S1 <= S2`
- `S1 < S2`: checks whether `S1` is a *proper* subset of `S2`
- `S1.issuperset(S2)`, `S1 >= S2`
- `S1 > S2`: as above
- `S1.union(S2, S3, ...)`, `S1 | S2 | S3 |...`: returns *new* set

Difference between `S1.union(S2)` and `S1 |S2`

- `union` function: accepts any iterable as argument
  Example: `S6.union([1,2,3], [2,4,1])`
- `|` operator: all operands must be sets
  Example: `S6 |[1,2,3]`

Operations on sets, frozensets II

- `S1.intersection(S2, S3, ...)`, `S1 & S2 & S3 & ...`
- `difference()`, or `-`: multiple arguments may be given
- `S1.symmetric_difference(S2)`, `^`
- Sets and frozensets may be mixed; result has type of first operand

Operations on sets I

(applicable to sets (mutable) only)

- `S1.add(element1)` (note: only *one* element at a time)
- `S1.remove(element1)` (ERROR if `element` is not a member)
- `S1.discard(element1)` (*no error* if `element` is not a member)
- `S1.pop()`: remove and return *arbitrary* element, ERROR if `S1` is empty
- `S1.clear()`: remove *all* elements

- `S1.add(element1)` (note: only *one* element at a time)
- `S1.update(S2, ...)`, `S1 |= S2 | S3 | ...`
- `S1.intersection_update(S2, ...)`, `S1 &= S2 & S3 & ...`
- `S1.difference_update(S2, ...)`, `S1 -= S2 | S3 |...` **NOTE!**
- `S1.symmetric_difference_update(S2)`, `S1 = S2`

Set operations on dict views

```
>>> # set operations on dict views
>>> keys & {'eggs', 'bacon', 'salad'}
{'bacon'}
>>> keys ^ {'sausage', 'juice'} == {'juice', 'sausage', 'bacon', 'spam'}
True
>>> keys | ['juice', 'juice', 'juice'] == {'bacon', 'spam', 'juice'}
True
```