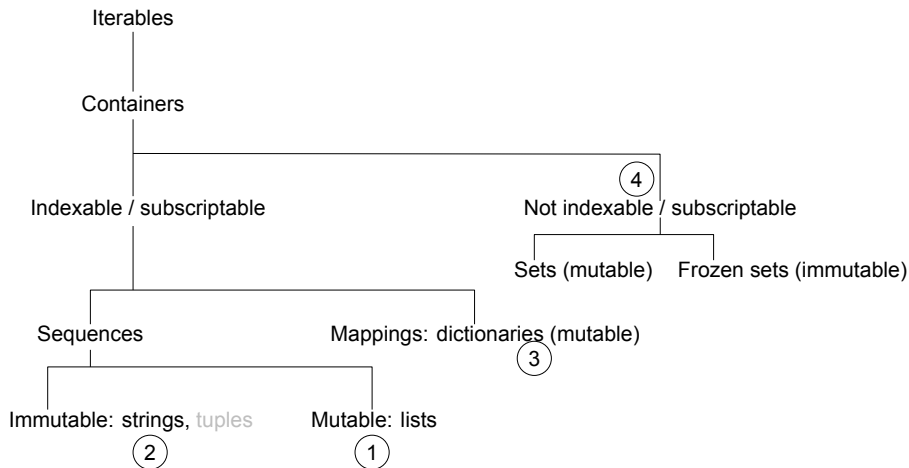


Python: container types (lists, strings, dictionaries, sets)

Intro. to computing

<https://mandarm.github.io/computing.html>

Container types



Container types I

■ 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: **mutable**

- example: { 1:11, 2:"y", 3:7.4, 3.14:"pi" }
- elements accessed using *keys* (must be distinct)

Container types II

- Sets: **mutable**, not “**subscriptable**” (indexable)
 - example: { 11, "y", 7.4 }
- Tuples (11, "y", 7.4): like lists, but **immutable**

Lists: operations

(applicable to immutable sequences also)

Operation	Result
<code>x in s</code>	true if an item of <code>s</code> is equal to <code>x</code> , else false***
<code>x not in s</code>	false if an item of <code>s</code> is equal to <code>x</code> , else true***
<code>s + t</code>	concatenation of <code>s</code> and <code>t</code> , does not change <code>s</code> , <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to concatenating <code>s</code> <code>n</code> times††
<code>len(s)</code>	length of <code>s</code>

*** 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` → slice is empty);
 - elements are listed in reverse order.
- example: `[22, 16, 10, -2, -98][10:0:-1]`
→ `[-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 $\text{len}(l) + \text{start}$ (or $\text{len}(l) + \text{end}$).
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 $\text{len}(s)$, or $\text{len}(s)-1$ 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 $\text{len}(l)-1$ if $n \geq \text{len}(l)$), upto and **excluding** first element (index 0)
- `l[n:-1:-1]` ($n \geq \text{len}(l)$) prints empty list
 - end (-1) replaced by $\text{len}(l)-1$ (rule 1)
 - either $n \leq \text{len}(l)-1$, or is replaced by $\text{len}(l)-1$
- `l[n : -len(l)-1 : -1]` prints elements in reverse order, starting from n (or $\text{len}(l)-1$ if $n \geq \text{len}(l)$), upto and **including** first element
 - end ($-\text{len}(l)-1$) 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
<code>s[i] = x</code>	item <code>i</code> of <code>s</code> is replaced by <code>x</code>
<code>del s[i]</code>	removes item <code>i</code> of <code>s</code>
<code>s[i:j] = t</code>	slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by the contents of <code>t</code> <code>i==j</code> → contents of <code>t</code> inserted in place of <i>empty slice</i> at index <code>i</code>
<code>del s[i:j:k]</code>	removes the elements of <code>s[i:j:k]</code> from the list, <code>:k</code> optional same as <code>s[i:j:k] = []</code>
<code>s[i:j:k] = t</code>	elements of <code>s[i:j:k]</code> are replaced by those of <code>t</code> <code>:k</code> optional, but if not 1, <code>s[i:j:k]</code> and <code>t</code> must have same length
<code>s += t</code>	extends <code>s</code> with contents of <code>t</code>
<code>s *= n</code>	updates <code>s</code> with its contents repeated <code>n</code> 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` $\equiv$ tab (somewhat useful for lining things up in columns)
- `\r` $\equiv$ carriage return (takes cursor back to beginning of line)
(try `print("hello\nworld")` and `print("hellooooo\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"hellooooo\rworld")`)

Lists: operations

(applicable to immutable sequences also)

Operation	Result
<code>x in s</code>	true if an item of <code>s</code> is equal to <code>x</code> , else false***
<code>x not in s</code>	false if an item of <code>s</code> is equal to <code>x</code> , else true***
<code>s + t</code>	concatenation of <code>s</code> and <code>t</code> , does not change <code>s</code> , <code>t</code>
<code>s * n</code> or <code>n * s</code>	equivalent to concatenating <code>s</code> <code>n</code> times††
<code>len(s)</code>	length of <code>s</code>

*** 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(separator-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(separator-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)`
`s.split(sep="seperator-string")`

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

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 if `seperator-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')])
```

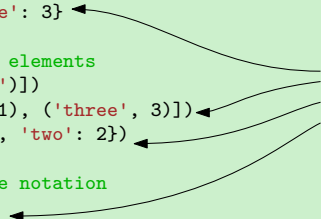
```
d5 = dict([('two', 2), ('one', 1), ('three', 3)])
```

```
d6 = dict({'three': 3, 'one': 1, 'two': 2})
```

```
# also possible to use key=value notation
```

```
a = dict(one=1, two=2, three=3)
```

All these are
equal



- 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

- *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 $\rightarrow$ mutable, *not hashable*
 - cannot be used as dictionary key or element of another set
- *frozenset* type $\rightarrow$ 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])` → OK
- **`|`** operator: all operands must be sets (or set-like)
Example: `S6 | [1,2,3]` → ERROR

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 I

- Keys views: keys are unique and hashable
 - support set-like operations, and operations with sets
- Items views:
 - ~~support set-like operations (keys are unique and hashable)~~
support set-like operations *if all values in an items view are hashable*
 - support operations with sets *if all values in an items view are hashable*
- Values views: **NOT** set-like
- For set operators, set-like views accept *any iterable* as the other operand

But recall: for set operators, sets can only accept other sets as operands.

Official documentation (*library/stdtypes.html#dictionary-view-objects*)
Items views also have set-like operations since the (key, value) pairs are unique and the keys are hashable. If all values in an items view are hashable as well, then the items view can interoperate with other sets.

Set operations on dict views II

Examples:

```
# Set
computer_types = { "laptop", "desktop", "server", "mobile", "tablet" }

# Example in which values are hashable
brands1 = {'lenovo': 5, 'hp': 5, 'asus': 3, 'acer': 2}
keys1 = brands1.keys()
values1 = brands1.values()

# Example in which values are hashable
brands2 = {'ibm': 1, 'sony': 3, 'dell': 5, 'apple': 4, 'msi': 1}
keys2 = brands2.keys()
values2 = brands2.values()

# Example in which values are NOT hashable
models = {"lenovo": [ "yoga", "ideapad", "thinkpad", "loq" ], \
          "apple" : [ "macbook air 13", "macbook air 15", "macbook pro"
                    ] }
```

Set operations on dict views III

```
# keys and values appear in the same order (insertion / creation order)
list(keys1)           # ['lenovo', 'hp', 'asus', 'acer']
list(values1)         # [5, 5, 3, 2]

# dict views: set operations and operations with sets
computer_types | keys1 | models.keys()      # mixing sets and key views OK
computer_types | keys1 | brands2.items()    # OK (values are hashable)
computer_types | keys1 | models.items()     # NOT OK

(keys1 | keys2) & models.keys()
brands1.items() ^ brands2.items() == brands1.items() | brands2.items()
brands2.items() | [5, 5, 3, 2]               # any iterable OK

computer_types | [5, 5, 3, 2]                # NOT OK
computer_types | set([5, 5, 3, 2])           # OK
```

Comprehensions and generator expressions

List comprehensions I

- Concise way to create lists (also sets, dicts) from other iterables
- Elements of new list typically obtained by
 - applying some operation to each member of the other iterable, OR
 - selecting only elements of the other iterable that satisfy some condition

Examples

```
squares = [x*x for x in range(10)]
roots = [x**0.5 for x in range(10)]
odd_nums = [ x for x in range(10) if x%2 == 1 ]
capitalized = [ s.capitalize() for s in ["bengaluru", "chennai",
                                         "delhi", "kolkata"] ]
```

- General form:
[<expression> <for clause> <more for or if clauses (optional)>]

LATER: `map()` and `filter()`

List comprehensions II

Using list comprehension

```
newlist = [ <expression> for <clause1> if <condition1>
           for <clause2> if <condition2> ]
```

Without using list comprehension

```
newlist = []
for <clause1>:
    if <condition1>:
        for <clause2>:
            if <condition2>:
                newlist.append(<expression>)
```

List comprehensions III

Examples:

```
# Using list comprehension
```

```
l = [ (x, y) for x in [1,2,3] for y in [3,1,4] if x != y ]
```

```
# Without using list comprehension
```

```
l = []
```

```
for x in [1,2,3]:
```

```
    for y in [3,1,4]:
```

```
        if x != y:
```

```
            l.append((x, y))
```

```
# Flattening a list
```

```
L = [[1,2,3], [4,5,6], [7,8,9]]
```

```
l = [num for list_element in L for num in list_element]
```

List comprehensions IV

```
A = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
```

```
# Without ANY listcomp
```

```
transposed = []
for i in range(len(A[0])):
    transposed_row = []
    for row in A:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
```

```
# Without NESTED listcomp
```

```
transposed = []
for i in range(len(A[0])):
    transposed.append([row[i] for row in A])
```

```
# Using NESTED listcomp
```

```
A_transpose = [ [row[i] for row in A] for i in range(len(A[0])) ]
```

Dictionary and set comprehensions

```
brands = {'ibm': 1, 'sony': 3, 'dell': 5, 'apple': 4, 'msi': 1}
inverse_map = { brands[k]:k for k in brands }
brands |= inverse_map

consonants = {x for x in 'some long string' if c not in 'aeiou'}
```

Generator expressions I

- General form: (same as for list comprehensions, except for enclosing brackets)
(<expression> <for clause> <more for or if clauses (optional)>)
- Interpretation: as for list comprehensions, **BUT**
 - list comprehension: entire list is generated and stored in memory
 - generator expression: elements are
 - *generated on demand* **ONCE** (more soon)
 - deleted from memory after they are iterated over⇒ generator expressions take much less memory
- Generator expression generating from mutable object (e.g., list) evaluated at *time of use*, NOT *time of creation*

```
mylist = ["a", "b", "c"]  
gen = (elem + "1" for elem in mylist)  
mylist.clear()  
for x in gen: print (x)      # prints nothing
```

Generator expressions II

Examples:

```
list_comp = [x ** 2 for x in range(10) if x % 2 == 0]
gen_exp = (x ** 2 for x in range(10) if x % 2 == 1)
print(list_comp)           # [0, 4, 16, 36, 64]
print(gen_exp)             # <generator object <genexpr> at ...>

print(list_comp[2])        # 16
list_comp += [100]         # OK

print(gen_exp[2])          # ERROR!
gen_exp += list_comp       # ERROR!
print(list(gen_exp))       # [1, 9, 25, 49, 81]
```

Comprehensions vs. generator expressions I

Broad thumbrules

- Use list comprehension if you need
 - indexing, slicing, or other list operations. e.g., reversing
 - multiple passes over the data, e.g., sorting
- Use generator expression if you need
 - only one pass over the data, e.g., `sum()`, `max()`, `min()`
 - `any()` or `all()`
 - to process very large quantities of data

Comprehensions vs. generator expressions II

Examples

```
# In the following examples, the full list has to be generated anyway
# => better to use listcomp

gen_exp = (x ** 2 for x in range(10) if x % 2 == 1)
list_comp += gen_exp

' '.join( str(i) for i in range(10) )

# Genexps generate sequence ONCE
l = (x ** 2 for x in range(10) if x % 2 == 0)
print("First pass")
for i in l: print(i)    # As expected
print("Second pass")
for i in l: print(i)    # Nothing is printed!!
print("More passes")
for i in (x ** 2 for x in range(10) if x % 2 == 0): print(i)
for i in (x ** 2 for x in range(10) if x % 2 == 0): print(i)
```

Comprehensions vs. generator expressions III

Difference between genexps and iterators

```
l1 = range(5)
l2 = (x ** 2 for x in range(5) )
for i in l1:           # Repeat nested for loops
    for j in l1:       #   with l2 instead of l1
        print(i, j)
```

Pathological example with 'infinite' sequence

```
from itertools import count # "infinite" generator
from math import log

listcomp = [ log(i) for i in count(10) ]
genexp = ( log(i) for i in count(10) )
for x in genexp:
    print(x)
    input('Continue?')
```

Digression: `any()` and `all()`

- `any(iterable)`: return True if any element of `iterable` evaluates to True; otherwise (or if `iterable` is empty), return False
- `all(iterable)`: return True if all elements of `iterable` evaluates to True, or if `iterable` is empty

Condition is vacuously true

Lazy evaluation / short-circuiting behaviour: *stop evaluating a Boolean condition as soon as its truth value is determined*

⇒ stop checking elements of `iterable` for

- `any()` → as soon as True element is found
- `all()` → as soon as False element is found

Question: which is preferable as argument to `any()`, `all()`: genexp or listcomp?

Odds and ends

Lists of sublists: * vs. comprehension

```
ls1 = [[] for i in range(3)]
ls1[0].append(10)
ls1[1].append(20)
ls1.append(30)
print(ls1)           # [[10], [20], [], 30]

ls2 = [[]]*3         # [ [], [], [] ] 3 references to the SAME empty list
ls2[0].append(10)    # [ [10], [10], [10] ]
ls2[1].append(20)    # ls2[0] and ls2[1] refer to the SAME list
ls2.append(30)
print(ls2)           # [[10, 20], [10, 20], [10, 20], 30]
```

- List comprehension: elements are newly created
- Concatenation using *: **items are referenced multiple times, NOT copied**
⇒ results with mutable elements may be unexpected (as above)

Question: what if elements are immutable?

Lists of sublists: * vs. comprehension (CONTD.)

Elements are references to **immutable** objects:

```
x = 10
l = [ x ] * 4
x = 20
# Q1. does l change?
l[0] += 1
# Q2. do all elements of l change or only l[0]?
```

Elements are references to **mutable** objects:

```
l = [ [] ] * 3          # l = [[], [], []] (3 references to SAME empty list)
l[0] = l[0] + [1, 2]     # l = [[1,2], [], []] (l[0] reassigned to new list)
l[1].append(3)           # l = [[1,2], [3], [3]] (l[1], l[2] refer to same list)
l[2] += [4, 5]           # l = [[1,2], [3,4,5], [3,4,5]] (+= modifies in place)
l[2] = l[0] + l[2]       # l = [[1,2], [3,4,5], [1,2,3,4,5]] (= reassigns)
```

Simple (=) vs. augmented assignments (+=, etc.)

For mutable objects, simple assignments reassign, but augmented assignments modify in place;

⇒ `l[2] += [4, 5]` is like `l[1].append(3)` rather than `l[2] = l[0] + l[2]`.

Lists of sublists: * vs. comprehension (CONTD.)

Elements are references to mutable objects (dictionaries):

```
brands = {'ibm': 1, 'sony': 3, 'dell': 5, 'apple': 4, 'msi': 1}
l = [ brands ] * 2 # brands, l[0] and l[1] all refer to the same object
brands["hp"] = 5
l[1]["lenovo"] = 10
```

Elements are references to mutable objects (sets):

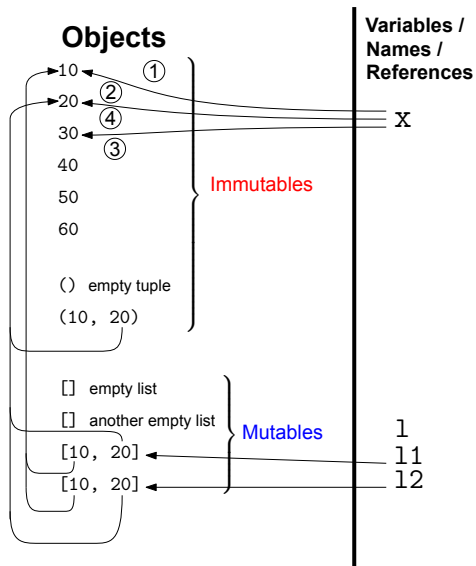
```
computer_types = { "laptop", "desktop", "server", "mobile", "tablet" }
l = [ computer_types ] * 2 # as above
computer_types.add("wearable")
l[1].add("embedded")
```

Making sense of the above I

- **Objects:** values, e.g., numbers, sequences, dictionaries, sets, etc.
 - each object occupies memory
 - each object has unique *identity*
 - `id(obj)`: return unique identity of `obj` (often its memory address)
- **References:** names used in a Python program for the above
- **Assigning a value to a variable** (say `X`):
 - `X` now *references* (refers to) / *is bound to* a different object / value
 - object that `X` referred to earlier **not** overwritten, remains unchanged

Making sense of the above II

```
x = 10
x = 20
x = 30
print(id(10))
print(id(20))
print(id(30))
print(id(x))
x -= 10
print(id(x))
l1 = [10, 20]
l2 = [10, 20]
l3 = []
l4 = []
l5 = l6 = []
print(id(l1), id(l1[0]), id(l1[1]))
print(id(l2), id(l2[0]), id(l2[1]))
print(id(l3) == id(l4))
print(id(l5) == id(l6))
```



Making sense of the above III

- When a name is used in an assignment
 - as an *l-value* → it may be reassigned to another object
 - as an *r-value* → the object it references is used in its place

Iterating over a changing iterable I

Thumbrule: avoid unless you clearly understand what is happening

Problems

```
numbers = [1, 2, 4, 3]
for n in numbers:
    if n % 2 == 0:
        numbers.remove(n)
print(numbers)  # Expected: [1, 3]; actual: [1, 4, 3] (4 is skipped)

# Infinite loop
l = [1, 2]
for i in l:
    l.append(i*i)
    print(l)
    input("Continue?")
```

Iterating over a changing iterable II

```
# Error! (dictionary changed size during iteration)
brands = {'ibm': 1, 'sony': 3, 'dell': 5, 'apple': 4, 'msi': 1}
for k in brands: brands[brands[k]] = k
for k in brands.keys(): brands[brands[k]] = k

# Error! (set changed size during iteration)
S = {1, 2, 4, 3}
for element in S:
    if element % 2: S.remove(element)
```

Iterating over a changing iterable III

Solutions:

- Lists: use list comprehension or slicing

```
for n in numbers[:]:    # slicing creates a (shallow) copy
    if n % 2 == 0:
        numbers.remove(n)
```

- Dictionaries: use dictionary comprehension or (static) list of current keys `list(dictionary.keys())`
- Sets: use set comprehension or `list(s)`