Data structures and collections

Programmazione Avanzata 2025-26

Data structures

- The Python language offers very powerful built-in data structures
  - `list` and `tuple`
  - `set`
  - `dict`
- They can be used to store and search information, and each is specialized to support some use cases
- Additional data structures are available in the standard library, to cover other use cases

Overview

- Alternatives
  - Dictionaries, maps, hash tables
  - Array data structures
  - Records, structs, data transfer objects
  - Sets, multisets
  - Stacks (LIFO)
  - Queues (FIFO)
  - Priority queues

Dictionaries, maps, hash tables

- `dict`
- `OrderedDict`
- `defaultdict`
- `ChainMap`
- `MappingProxyType`

Array data structures

- `list`
- `tuple`
- `array`
- `str`
- `bytes`
- `bytearray`

Records, structs, data transfer objects

- `dict`
- `tuple`
- `class`
- `dataclass`
- `Namedtuple, NamedTuple`
- `Struct`

Sets, multisets

- `set`
- `frozenset`
- `Counter`

Stacks (LIFO)

- `list`
- `deque`
- `LifoDeque`

Queues (FIFO)

- `list`
- `deque`
- `Queue`

Priority queues

- `list`
- `heapq`
- `PriorityQueue`

Considerations

- Some types are extremely versatile (**`list`**, **`dict`**)
- Some types are “improvements” of basic types
- Some types are extremely special-purpose (e.g. for parallel computation)

Cheatsheet

Schema sinottico delle principali operazioni sui contenitori					
Operation	str	list	tuple	set	dict
Create	`"abc"  'abc'`	`[a, b, c]`	`(a, b, c)`	`{a, b, c}`	`{a:x, b:y, c:z}`
Create empty	`""  ''`	`[]  list()`	`()  tuple()`	`set()`	`{}  dict()`
Access `i`-th item	`s[i]`	`l[i]`	`u[i]`		`d[key]` `d.get(key,default)`
Modify `i`-th item		`l[i]=x`			`d[key]=x`
Add one item (modify value)		`l.append(x)`		`t.add(x)`	`d[key]=x`
Add one item at position (modify value)		`l.insert(i,x)`			
Add one item (return new value)	`s+'x'`	`l+[x]`	`u+(x,)`		
Join two containers (modify value)		`l.extend(l1)`		`t.update(t1)`	
Join two containers (return new value)	`s+s1`	`l+l1`	`u+u1`	`t.union(t1)  t|t1`	
Does it contain a value?	`x in s`	`x in l`	`x in u`	`x in s`	`key in d` (search keys) `x in d.values()` (search values)
Where is a value? (returns index)	`s.find(x)` `s.index(x)`	`l.index(x)`	`u.index(x)`		
Delete an item, by index		`l.pop(i)  l.pop()`			`d.pop(key)`
Delete an item, by value		`l.remove(x)`		`t.remove(x)` `t.discard(x)`	
Sort (modify value)		`l.sort()`			
Sort (return new **list**)	`sorted(s)`	`sorted(l)`	`sorted(u)`	`sorted(t)`	`sorted(d)` (keys) `sorted(d.items())`

https://polito-informatica.github.io/Materiale/CheatSheet/Python_Cheat_Sheet-3.4.pdf

Comparison and ordering

- Objects can be compared if they define an **`__eq__()`** method
  - Used internally by **`==`** and **`!=`** operators
- Used internally by **`find()`**, **`index()`**, **`in`**, …
- Objects can be ordered if they define a **`__lt__()`** method (and optionally, other comparison dunder methods)
  - Must define **`__eq__()`**, in addition
  - Used internally by **`< <= > >=`** operators
  - Used internally by **`sort()`**, **`sorted()`**

Special case: predefined types

- Some built-in collections already define `__eq__()` and `__lt__()`, therefore they are comparable and sortable
  - Strings, lists, tuples compare their elements in left-to-right order
  - The contained values must be comparable/sortable, too
- Dictionaries support `__eq__()` but not `__lt__()`
  - Dictionaries cannot be ordered
- Sets support `__eq__()`, but define `__lt__()` to mean "subset"
  - Misleading, one shall not try to order a list of sets

Special case: dataclasses

- By default, a dataclass defines the **`__eq__()`** method
  - To prevent this behavior, **`@dataclass(eq=False)`** can be used
- By default, does not define the **`__lt__()`** method
  - Can be generated with **`@dataclass(order=True)`**
    - Generated **`__lt__()`**, **`__le__()`**, **`__gt__()`**, and **`__ge__()`**
  - Automaticaly generated methods compare all the fields of the object, in the order in which they are declared
  - One or more fields may be omitted from comparison and ordering methods, by inizializing them with **`field(compare=False)`**

Example

```
@dataclass(order=True)
Class DataClassCard
     rank: int
     suit: str = field(compare=False)
```

Sorting by other criteria

- To sort a collection using criteria different from **`__lt__()`**, or if **`__lt__()`** is not defined, **`key=`** argument is used
  - **`key=operator.itemgetter('keyname')`**
    - sort by **`dictionary['keyname']`**
  - **`key=operator.itemgetter(itemnumber)`**
    - Sort by **`list/tuple[itemnumber]`**
  - **`key=operator.attrgetter('attrname')`**
    - Sort by **`object.attrname`**
  - **`key = lambda obj: something(obj)`**
    - Sort by value of **`something()`** function
    - **`lambda v: v.voto`** equiv. to **`operator.attrgetter('name')`**

Dictionaries

- Map a “key” to a “value”
  - Key: unique value of a hashable type
  - Value: any object
- `dict`
  - Very efficient, constant time for insertion, search, deletion
  - Retains insertion order of elements
  - Has built-in syntax `{ key: val }` for creation

Dictionaries

- `d[key] = value # Sets a new value for a key`
- `d[key] # Retrieves value from the key. May raise KeyError`
- `d.clear() # Clears a dictionary`
- `d.get(key, default) # Returns the value for a key if it`
  `# exists in the dictionary, otherwise,`
  `# returns a default value`
- `d.items() # Returns a list of key-value pairs in a`
  `# dictionary`

Dictionaries

- `d.keys() # Returns a list of keys in a dictionary`
- `d.values() # Returns a list of values in a dictionary`
- `d.pop(key, default) # Removes a key from a dictionary,`
  `# if it is present, and returns its`
  `# value, otherwise, returns a`
  `# default value`
- `d.popitem() # Removes the last key-value pair from`
  `# a dictionary`
- `d.update(obj) # Merges a dictionary with another`
  `# dictionary`

“Hashable”?

- A hashable object
  - Has a hash value that never changes during its lifetime, defined by **`__hash__()`**
  - It can be compared to other objects, i.e, defines **`__eq__()`**
- Hashable objects that compare as equal must have the same hash value
  - $a == b \Rightarrow hash(a) == hash(b)$
- Instances of user-defined classes are hashable by default
  - They all compare unequal (except with themselves), and their hash value is derived from their **`id()`**
  - It is possible to redefine this behavior

Hash functions

- A hash function is a function that maps any object into an integer number (over 64 bit)
- It is needed to quickly discover if two objects are
  - Surely different
  - Very likely equal
- Used in the **`hash()`** function and internally by **`set`**, **`frozenset`** and **`dict`**

Other dictionaries

- **`collections.defaultdict`**
  - A class that automatically provides a default value for non-existent keys
  - Requires a “factory” function to build the default values: list, string, integer, … or custom ones
  - **`d = collections.defaultdict(int)`**
- **`types.MappingProxyType`**
  - Creates a “read-only” dictionary, without copying it
  - **`readonly_d = types.MappingProxyType(normal_d)`**
  - All modifications will generate an exception
    - **`TypeError: 'mappingproxy' object does not support item assignment`**

Main array types

- `list`
  - The most versatile one, mutable ordered sequence of objects of any value
  - Indexed by number, from `0` to `len()-1`
- `tuple`
  - An immutable version of a list: elements cannot be added, removed nor replaced
    - But elements can be mutated, if they are mutable
  - Hashable, if its elements are hashable
- `str`
  - An array of Unicode characters
  - Immutable

Specialized array types

- **`array.array`**
  - Implemented in C as an array of elements of the same basic type (**`byte`**, **`int`**, **`float`**)
  - The type is declared at the time of creation
    - `arr = array.array("f", (1.0, 1.5, 2.0, 2.5))`
  - Uses less memory than normal lists, but less versatile
- **`Bytes`**
  - Immutable array of single bytes
- **`Bytearray`**
  - Mutable array of single bytes

Records

- A record is a collection of data of different types, and different meanings, grouped together to represent a single high-level information

Records

- With dictionaries, implemented as

```
car = {
    "type": "Panda",
    "year": 2010
}
```

- With classes, implemented as

```
class Car:
    def __init__(self, type, year):
        self.type = type
        self.year = year
```

Records

- With tuples implemented as

```
car = ("Panda", 2010)
```

- With dataclasses implemented as

```
@dataclass
class Car:
    type: str
    year: int
```

Specialized record types

- **`collections.namedtuple`**
  - A tuple whose indices are not integers, but attributes (like objects)

```
Car = collections.namedtuple("Car", ("name", "year"))
c1 = Car("Panda", 2010)
c1.name # 'Panda'
```

  - Attribute values are immutable
- **`typing.NamedTuple`**
  - Uses a syntax similar to dataclasses

```
class Car(typing.NamedTuple):
    name: str
    year: int
```

Sets

- `set`
  - Mutable container of hashable objects
  - Duplicates are not allowed
  - Simple syntax: `{ 1, 2, 3 }`
  - Supports set-theory operations
- `frozenset`
  - An immutable version of a set: once created, its elements cannot be changed
  - Since it is hashable, it may be used as a key in a dictionary (or as an element in a set)

Multisets and collections.Counter

- The `Counter` class is useful for computing and storing frequencies of items (i.e. counts of elements that may appear more than once in a set)

```
cnt = collections.Counter([1, 2, 3, 3, 4, 5,
1, 8, 3, 5, 2, 2, 3, 8])
Counter({3: 4, 2: 3, 1: 2, 5: 2, 8: 2, 4: 1})
```

- Great for statistics, frequency counting, histogram, duplicate detection, ranking, etc.
  - Internally stored as a defaultdict, with keys at the set elements, and values as the occurrence counts, with default value = 0

Counter objects

- `c = Counter() # A new, empty counter`
- `c = Counter('gallahad') # A new counter from an iterable`
- `c = Counter(['eggs', 'ham']) # A new counter from an # iterable`
- `c = Counter({'red': 4, 'blue': 2}) # A new counter from a # mapping`
- `c = Counter(cats=4, dogs=8) # A new counter from keyword # args`

Counter objects

- Manually increasing counts:

```
for word in ['red', 'blue', 'red', 'green']:
cnt[word] += 1
```

- Equivalent to

```
cnt = Counter(['red', 'blue', 'red', 'green'])
```

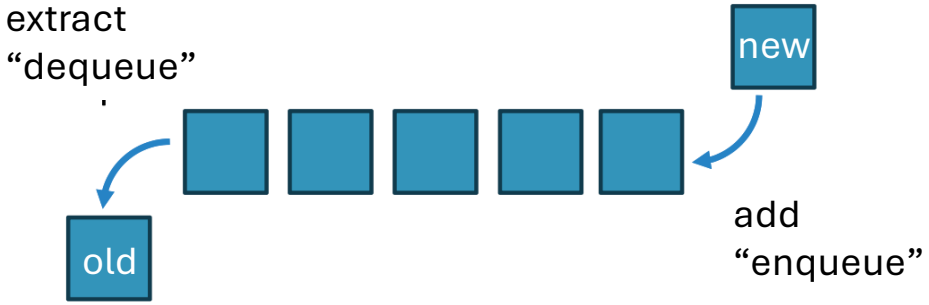
Counter objects

- `c.most_common(n)  # The 'n' (default: all) most common`
  `# items`
- `c.total() # Returns the total of all counts`
- `list(c) # Lists unique elements`
- `set(c) # Converts to a set`
- `dict(c) # Converts to a regular dictionary`
- `c.items() # Converts to a list of (elem, cnt) pairs`

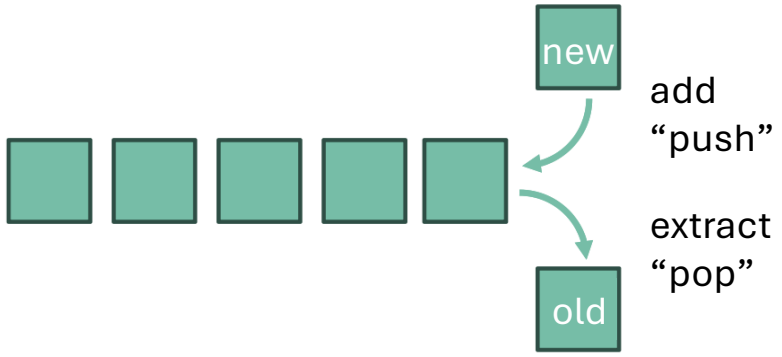
Counter objects

- `c.elements() # Returns a list [elem, …] with`
  `# repetitions`
- `Counter(dict(list_of_pairs)) # Converts from a list of`
  `# (elem, cnt) pairs`
- `c.most_common()[:-n-1:-1] # The ‘n’ least common`
  `# elements`
- `+c # Removes zero and negative counts`
- `c.clear() # Resets all counts`

Queues and stacks



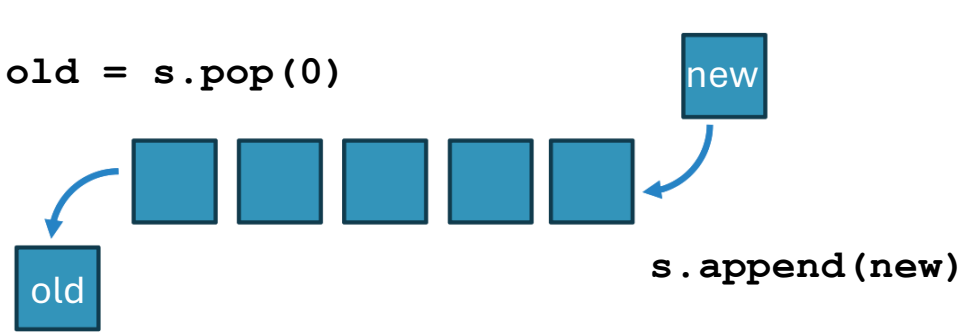
FIFO Queue – First-in First-out



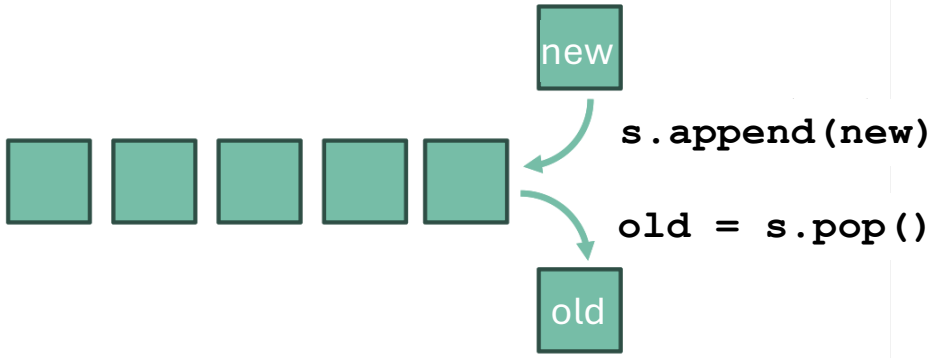
LIFO Stack – Last-in First-out

List implementations

```
s = list()
```



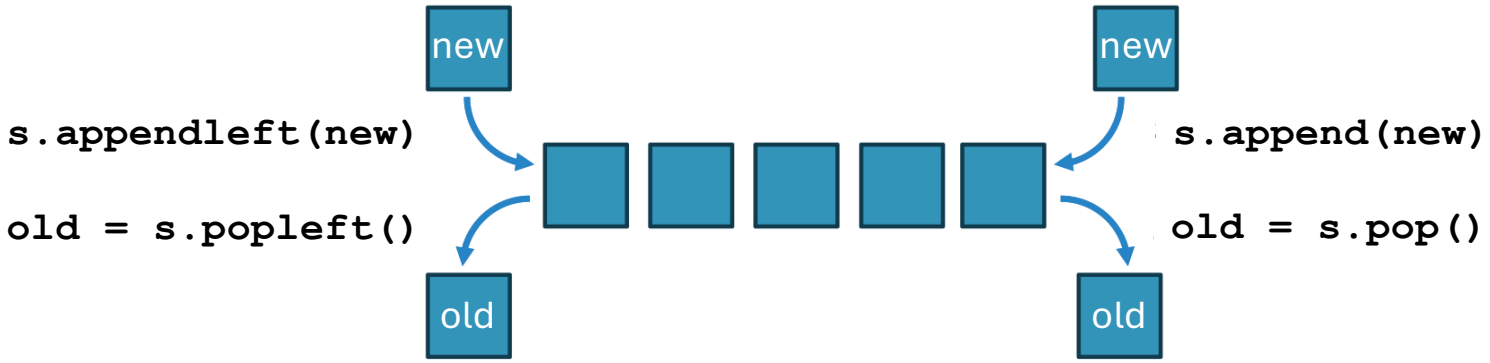
FIFO Queue – First-in First-out



LIFO Stack – Last-in First-out

Double-ended queue

```
s = collections.deque()
```



Double-ended queue

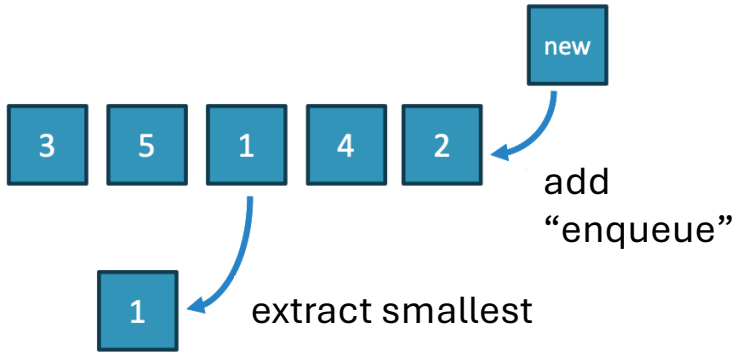
- As a FIFO Queue
  - **`append()`** and **`popleft()`**
    - Most popular choice
  - **`appendleft()`** and **`pop()`**
    - Also possible, same efficiency
- As a LIFO Stack
  - **`append()`** and **`pop()`**
    - Most popular choice
    - Might use a list, instead
  - **`appendleft()`** and **`popleft()`**
  - Also possible, same efficiency

Double-ended queue

- `d = deque() # Creates new empty deque`
- `d = deque(iterable) # Deques from list`
- `d = deque(maxlen=N) # Hosts max N elements, discards`
  `# older ones if more are added`
- `d.extend(iterable) # Adds list of elements at end`
- `d.extendleft(iterable) # Adds list of elements at`
- `# beginning`
- `d.rotate(n) # Rotates elements by n steps`
- `d[i] # Accesses element (slower than lists)`
- `d.index(x), d.insert(i, x), d.remove(x),`
  `d.reverse() # Same as lists`

Priority Queues

- Elements are added in any order
- Elements are removed according
- to their “priority”
- Priority is determined by the sorting order of the elements
  - Often, a `(priority, value)` tuple is created
  - Or, the the object’s `__lt__()` method is exploited



Priority Queues

- **heapq**
  - `h = [] # Uses plain lists`
  - `h = heapify(iterable)`
  - `len(h)`
  - `len(h)==0`
  - `heapq.heappush(h, x) # Items x must be comparable`
  - `x = heapq.heappop(h)`
- **queue.PriorityQueue**
  - `q = queue.PriorityQueue()`
  - `q.qsize()`
  - `q.empty()`
  - `q.full()`
  - `q.put(x) # Items x must be comparable`
  - `x = q.get_nowait()`