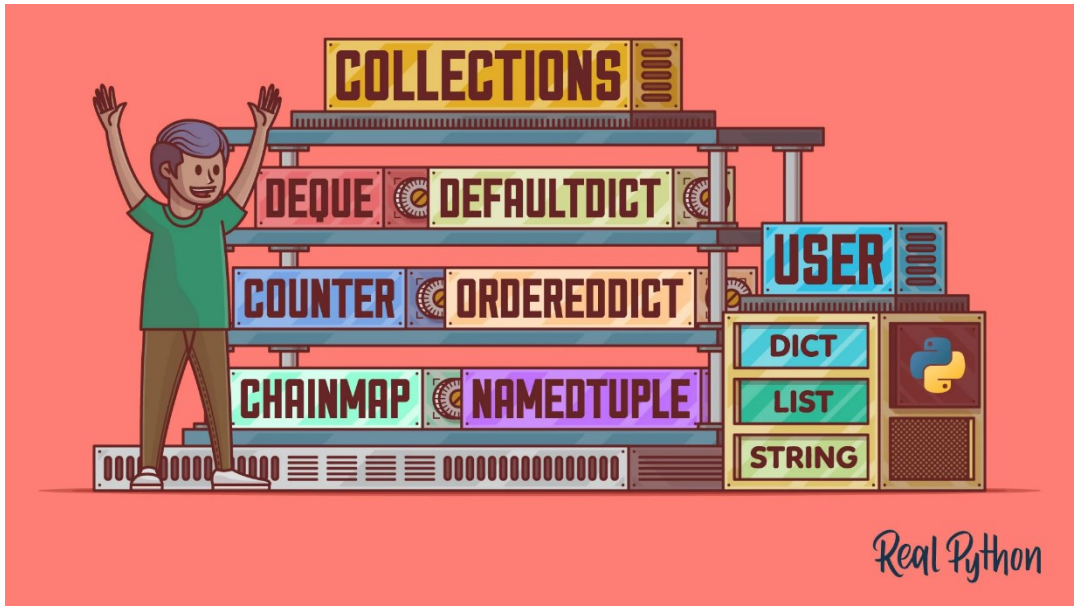
2024

TdP

Python data structures and collections

A wide choice of containers for your data

Fulvio Corno
Giuseppe Averta
Carlo Masone
Francesca Pistilli



https://realpython.com/python-collections-module/
https://realpython.com/python-data-structures/
https://docs.python.org/3/library/collections.html
https://docs.python.org/3/library/datatypes.html

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

- Possibili scopi x cui potremo dover raccogliere dei dati assieme

CODE

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Remember…

Riassunto della sintassi x compiere le operazioni fondamentali sulle strutture dati + importanti.

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[k]` `d.get(k,default)`
Modify `i`-th item		`l[i]=x`			`d[k]=x`
Add one item (modify value)		`l.append(x)`		`t.add(x)`	`d[k]=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`	`k 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(k)`
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-3.2.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`

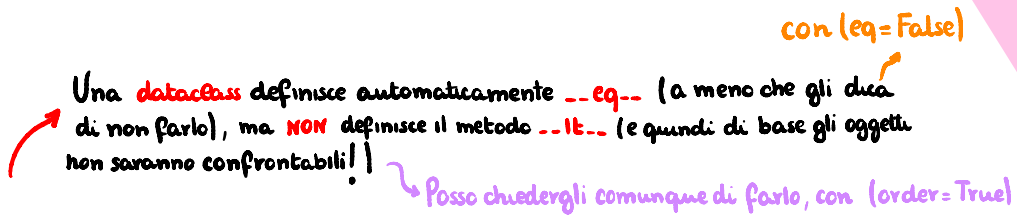
- I tipi primitivi (stringhe e numeri) sono confrontabili tra loro.
  - Anche liste e tuple! ↝ I dizionari non sono confrontabili tra loro!
- Gli oggetti di una classe generica creata da noi non definirà automaticamente i metodi `__eq__` e `__lt__`, quindi non potrò né confrontare gli oggetti della classe né ordinarli in una struttura data. Per poterlo fare, dovrò definire io i metodi `__eq__` e `__lt__` in cui specifico come confrontare gli oggetti.

Special case: predefined types

- Some built-in collections already define __eq__ and __lt__, therefore they are comparable and sortable
  - str, list, tuple compare their elements in left-to-right order
  - The contained values must be comparable/sortable, too
- Dictionaries support __eq__ but not __lt__
  - dict object cannot be ordered
- Sets support __eq__, but define __lt__ to mean “subset”
  - Misleading, do not try to order a list of sets

↳ Gli insiemi definiscono __lt__ non x intendere minore, ma sottoinsieme!

Special case: dataclasses



- By default, a dataclass defines the `__eq__` method
  - To prevent, define it as `@dataclass(eq=False)`
- By default, a dataclass does not define the `__lt__` method
  - To generate it, define with `@dataclass(order=True)`: will generate `__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 Voto:
    esame: str
    cfu: int
    punteggio: int
    lode: bool
    data: str = field(compare=False)
```

Se chiedo alla dataclass di generare anche il metodo __lt__ tramite order=True, allora Python genererà un confronto tra gli attributi in ordine di apparizione all'interno della classe. Nell'esempio a sx, i voti verranno confrontati prima per esame. A parità di esame per cui. A parità di CFU per punteggio e cosi via.
Se NON voglio che un attributo venga utilizzato per il confronto, dopo la sua definizione dovrò scrivere = field(compare=False).

Sorting by other criteria

Quello visto in precedenza è l'ordinamento NATURALE degli oggetti.

Se non voglio fare un ordinamento confrontando con __lt__, potrò usare il parametro key, che definisce qual è la chiave di ordinamento in quella specifica operazione.

Se ci metto la key, __lt__ viene ignorato!

Il valore del parametro che assume key è una funzione, che verrà utilizzata direttamente dal metodo .sort() x definire l'ordinamento!

- If you want to sort a collection using criteria different from the `__lt__` method (or if `__lt__` is not defined), use the `key=` argument
- `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
  - Example: `lambda v: v.voto` is equivalent to `operator.attrgetter('voto')`

Overview

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Dictionaries

Nei dizionario associo una key univoca con un value!

- Map a “key” to a “value”
  - Key: unique value of a hashable type
  - Value: any object

Come chiave di un dizionario potrò usare solamente oggetti di tipo hashable!

- `dict`
  - Very efficient, constant time for insertion, search, deletion
  - Retains insertion order of elements
  - Has built-in syntax `{ key: val }` for creation

Metodi principali!

`d[key] = value`	Set a new value for a key
`d[key]`	Retrieve 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.
`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”?

Come funzionano i dizionari?
Prende la chiave, e trasforma la chiave in un numero attraverso una funzione detta funzione di hash. Questo numero viene usato per indicizzare una tabella per indice. Se gli oggetti sono uguali, i numeri saranno uguali. Se gli oggetti sono diversi… dipende! Può darsi che i numeri vengano uguali o diversi.
Una buona funzione di hash è una funzione che prende i valori del dato e mi calcola un numero in un range possibilmente molto ampio evitando le collisioni più probabili.
Le chiavi di un dizionario dovranno essere oggetti hashable, il che significa che dovranno poter essere trasformati in un numero grazie a una funzione di hash.

- A hashable object
  - Has a hash value that never changes during its lifetime (defines __hash__)
  - It can be compared to other objects (defines __eq__)
- Hashable objects that compare as equal must have the same hash value
  - $a == b \Rightarrow hash(a) == hash(b)$
- Note: 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(). You can redefine this behavior

IMP: Quando creiamo una classe, se vogliamo che gli oggetti della classe possano essere usati come chiavi di un dizionario, allora dobbiamo definire la funzione __hash__ x quella classe! Inoltre, gli oggetti non dovranno MAI cambiare valore di hash nella loro lifetime!

Se due oggetti sono =, saranno = anche i risultati della funzione di hash applicata sugli oggetti!

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 in set, frozenset and dict.

https://docs.python.org/3/reference/datamodel.html#object.__hash__

Other dictionaries

Dizionari che sono classi figlie di dict().

- `collections.defaultdict`
  - A class that automatically provides a default value for non-existent keys
  - Requires a "factory" function to build the default values: list, str, int, … or custom
  - `d = collections.defaultdict(int)`

Nel defaultdict, se cerco un valore x una chiave NON esistente, mi viene restituito un valore di default!
↳ Il valore lo scelgo io quando creo il dizionario!

- `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`

Si trova nel modulo types!

Dizionario che riceve in input un dizionario e ne cancella tutti i metodi che vanno a modificare il dizionario!

Overview

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Main Array types

- `list`
  - The most versatile one, mutable ordered sequence of objects of any value
  - Indexed by number (0…len()-1)
- `tuple` → Lista immutabile!
  - 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

↳ NON ci serviranno quasi mai

- `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 Arrays of Single Bytes
- `bytearray`: Mutable Arrays of Single Bytes

Overview

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Records

record: struttura contenente dati di tipo ≠.

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

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

Implemented as…

dict

class

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

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

tuple

dataclass

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

Specialized record types

↳ NON useremo!

- `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
```

Overview

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

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's hashable, it may be used as a key in a dictionary (or as an element in a set)

Multisets and `collections.Counter`

La funzione .Counter() del modulo collections è un oggetto che riceve in input una struttura dati (ad esempio una lista) e conta le occorrenze di ciascun oggetto ∈ all'interno della struttura dati!
↳ Gli oggetti che metto dentro al counter dovranno sicuramente definire il metodo __eq__

- 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, …
- Internally stored as a defaultdict, with keys at the set elements, and values as the occurrence counts, with default value = 0

https://docs.python.org/3/library/collections.html#counter-objects

Creating **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

L'oggetto Counter può ricevere anche una stringa in input; in questo caso conterà le occorrenze di ciascun singolo carattere all'interno della stringa!

Posso passare un dizionario con dei conteggi parziali!

- Manually increasing counts:

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

Questa cosa equivale all'utilizzo dell'oggetto Counter, che è molto + easy!

equivalent to

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

What can I do with a `Counter`?

- Metodi che posso usare sull'oggetto Counter()!

```
c.most_common(n)                    # the ‘n’ (default: all) most common items
c.total()                           # total of all counts
list(c)                             # list unique elements

set(c)                              # convert to a set
dict(c)                             # convert to a regular dictionary
c.items()                           # convert to a list of (elem, cnt) pairs
c.elements()                        # return a list [elem, …] with repetitions
Counter(dict(list_of_pairs))        # convert from a list of (elem, cnt) pairs
c.most_common()[:-n-1:-1]           # n least common elements
+c                                  # remove zero and negative counts
c.clear()                           # reset all counts
```

Gli n elementi del Counter che hanno i conteggi (numero di occorrenze) maggiori.

totale dei conteggi di tutti gli elementi

lista degli elementi che compaiono senza conteggi

Lista di tuple

Lista contenente gli oggetti x il numero di volte che compaiono nel conteggio

funzione che ritorna l'oggetto col conteggio minore all'interno del Counter

Overview

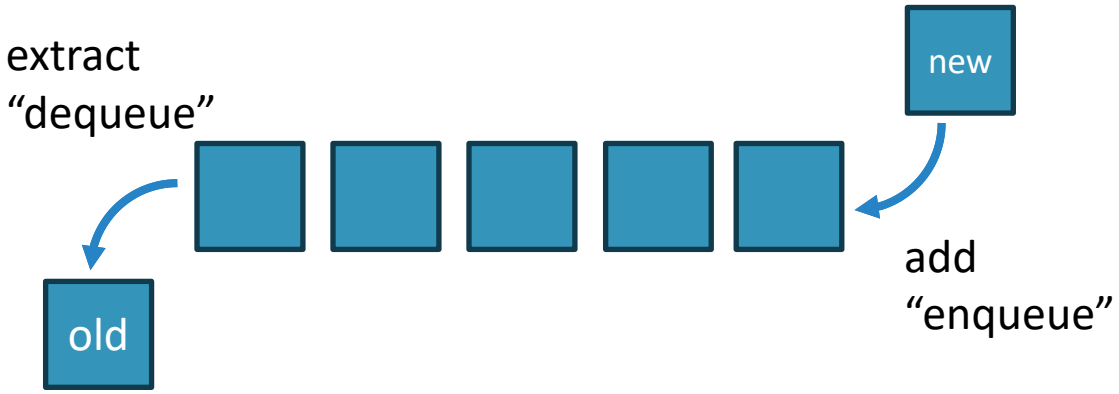
Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

Some types are extremely versatile (list, dict)

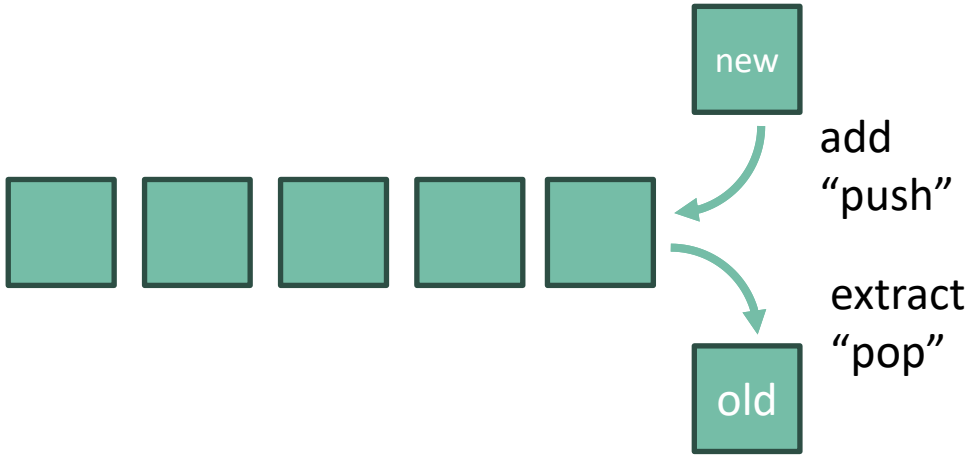
Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Queues and Stacks



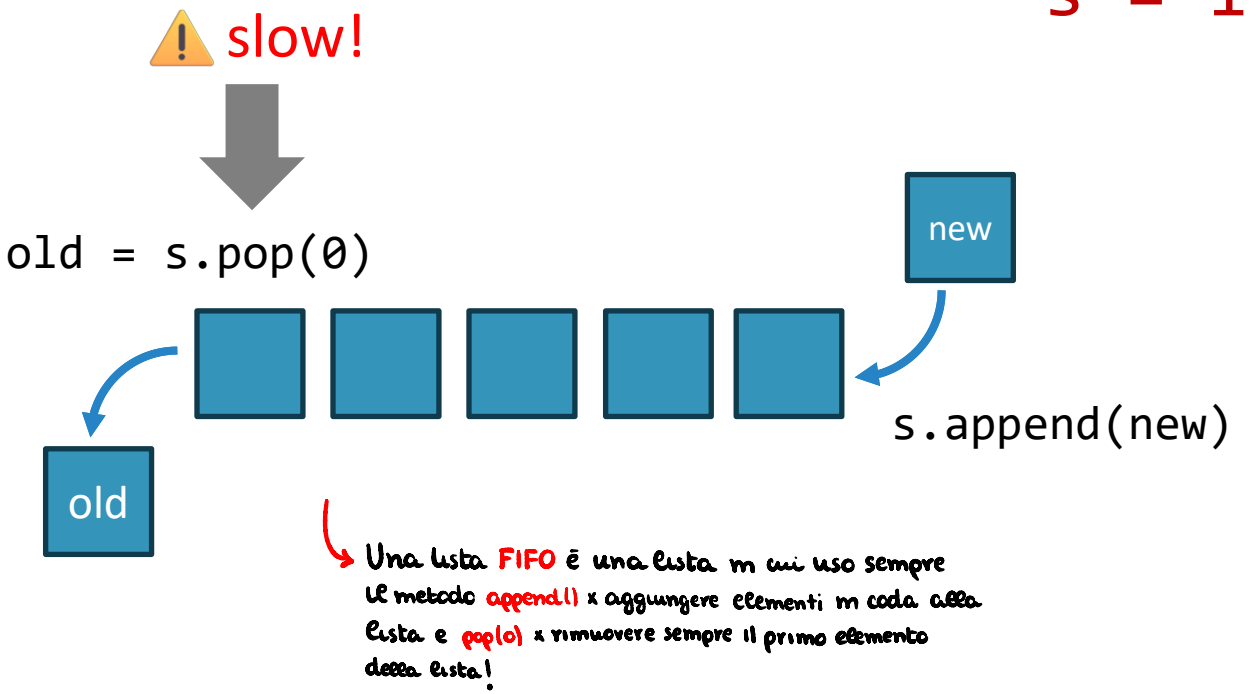
FIFO Queue – First-In First-Out



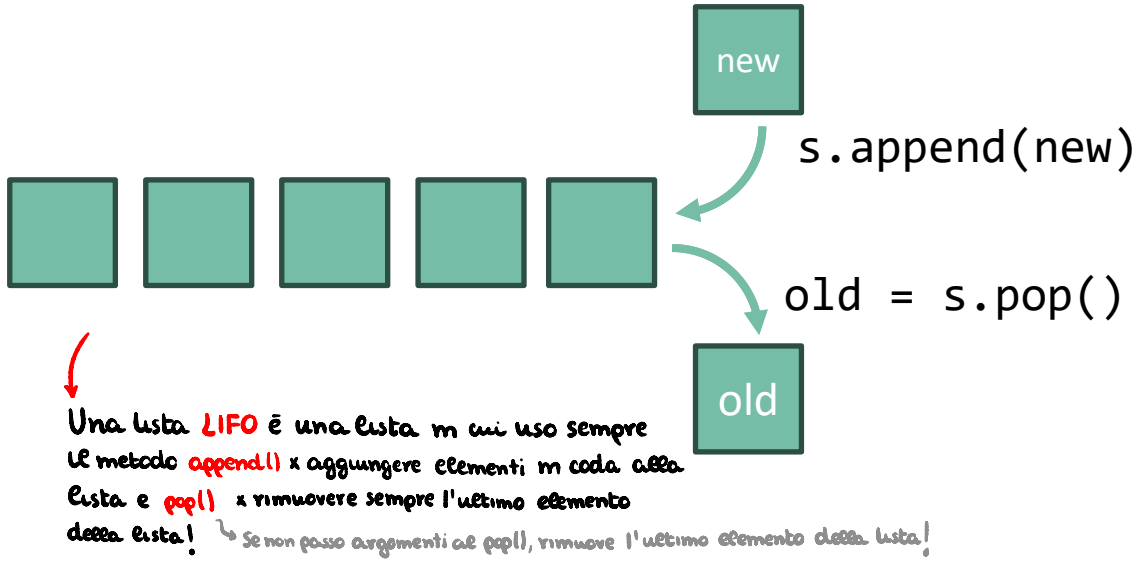
LIFO Stack – Larst-In First-Out

List implementations

```
s = list()
```



FIFO Queue – First-In First-Out

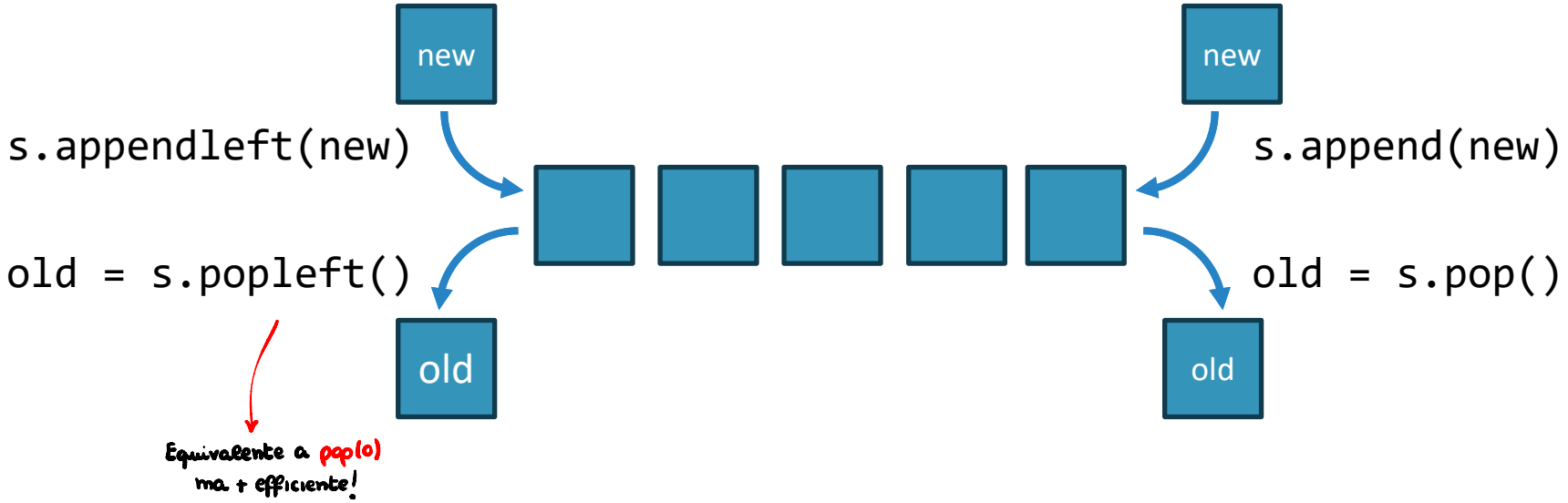


LIFO Stack – Larst-In First-Out

deque: double-ended queue

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

La dequeII è una struttura dati specializzata che può essere usata sia in modalità LIFO che FIFO in modo + efficiente rispetto alla classica lista!



All operations have the same efficiency

https://docs.python.org/3/library/collections.html#deque-objects

Using a deque

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

Other **deque** methods

`d = deque()`	New empty deque
`d = deque(iterable)`	Deque from list
`d = deque(maxlen=N)` ↝ N: numero massimo di elementi	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)`	Rotate elements by n steps
`d[i]` ↝ L'accesso agli elementi è + lento rispetto alle liste!	Access element (slower than lists)
`d.index(x), d.insert(i, x), d.remove(x), d.reverse()`	Same as lists

↓ Metodi utili che possono essere utilizzati su una deque!

https://docs.python.org/3/library/collections.html#deque-objects

Overview

Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `tuple` • `class` • `dataclass` • `namedtuple`, `NamedTuple` • `Struct`	• `set` • `frozenset` • `Counter`	• `list` • `deque` • `LifoDeque`	• `list` • `deque` • `Queue`	• `list` • `heapq` • `PriorityQueue`

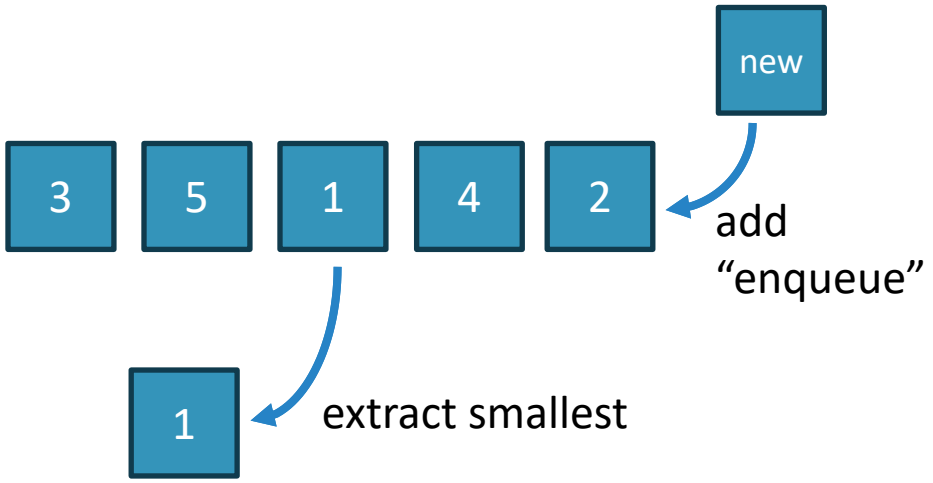
Some types are extremely versatile (list, dict)

Some types are “improvements” of basic types

Some types are very specialized (e.g. for parallel computation)

Priority Queues

• Nelle priority queue gli elementi vengono aggiunti in un qualsiasi ordine ma verranno poi estratti sulla base di una priorità!
↳ Per definire la priorità dovrò poter confrontare fra loro gli oggetti che quindi dovranno definire il metodo __lt__ nella classe
↳ + piccolo => priorità + alta!



- Elements are added in any order
- Elements are removed according to their "priority"
- Priority is determined by the sorting order of the elements
- Often, we create a tuple:
  - `(priority, value)`
- Or we rely on the object's `__lt__` method

Priority queues in Python

Items x must be comparable (implement __lt__)

IMPORTANTE!!!

heapq – uses plain lists

heapq è un modulo con una serie di funzioni che usano una lista normale in un modo particolare.

- h = [] → Lista semplice
- h = heapify(iterable)
- len(h)
- len(h)==0

Con .heappush(h,x) aggiungo l'elemento x alla lista h

- heapq.heappush(h, x)
- x = heapq.heappop(h)

→ Con .heappop(h) mi estrae e ritorna nella variabile x l'elemento con la priorità + alta della lista!

queue.PriorityQueue

Implementazione ad oggetti della PriorityQueue

- q = queue.PriorityQueue()

→ Creo un oggetto PriorityQueue dal modulo queue

- q.qsize()
- q.empty()
- q.full()
- q.put(x) → Aggiungo un elemento in un punto qualsiasi della queue
- x = q.get_nowait()

→ .get_nowait() mi restituisce l'oggetto della queue con la priorità + alta!

License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/