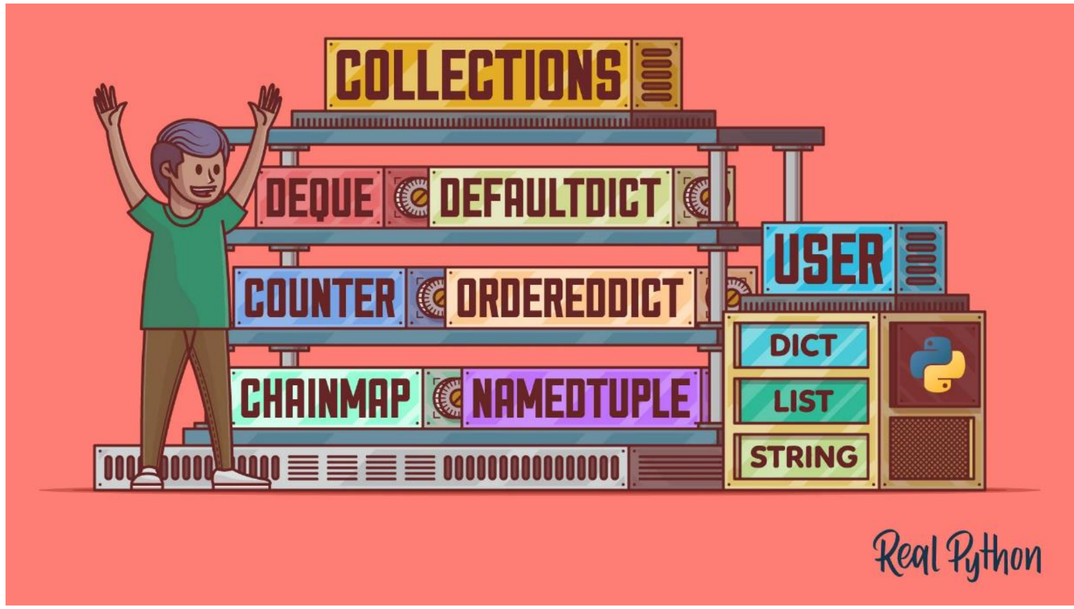
2025

TdP

Python data structures and collections

A wide choice of containers for your data

Giuseppe Averta
Carlo Masone
Francesca Pistilli
Davide Buoso
Gaetano Falco



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

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

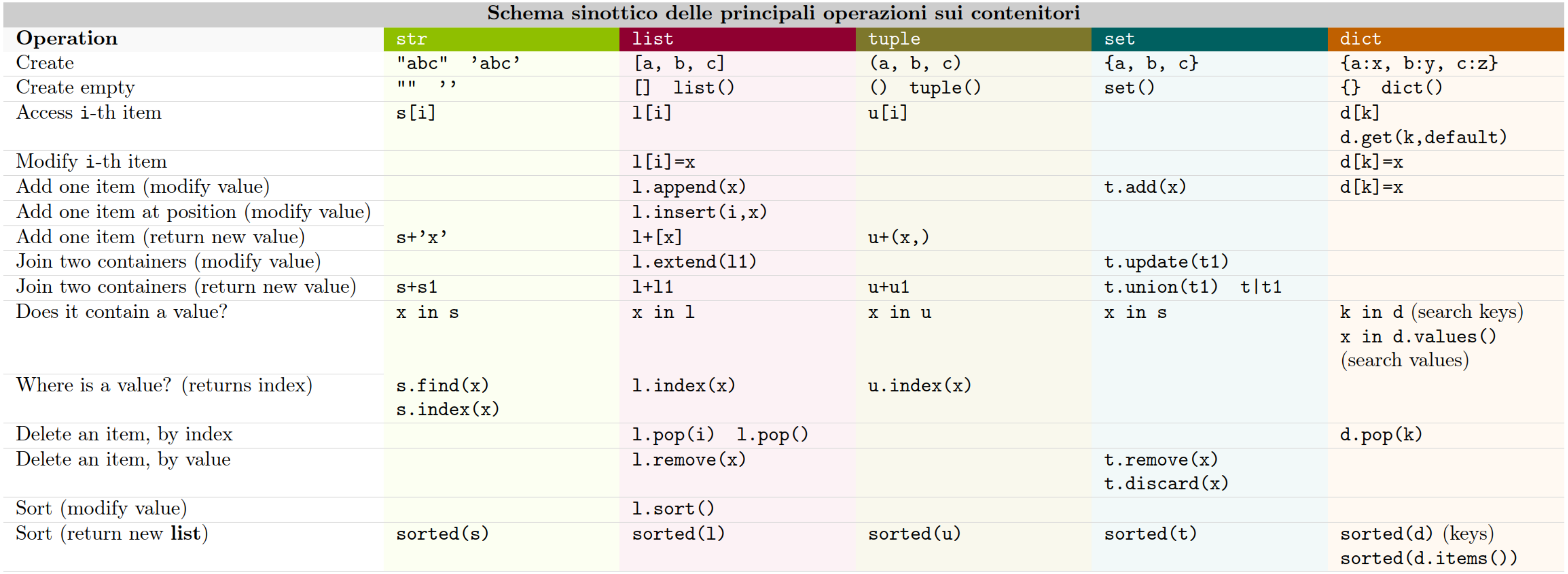
- `list`
- `heapq`
- `PriorityQueue`

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…



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`

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

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)
```

Sorting by other criteria

- 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
- `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`

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

- 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

`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”?

- 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

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

- `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)`
- `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`

Overview

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`

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`
  - 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 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`	• `list`	• `dict`	• `set`	• `list`	• `list`	• `list`
• `OrderedDict`	• `tuple`	• `tuple`	• `frozenset`	• `deque`	• `deque`	• `heapq`
• `defaultdict`	• `array`	• `class`	• `Counter`	• `LifoDeque`	• `Queue`	• `PriorityQueue`
• `ChainMap`	• `str`	• `dataclass`				
• `MappingProxyType`	• `bytes`	• `namedtuple`, `NamedTuple`				
	• `bytearray`	• `Struct`				

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

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

Implemented as…

dict

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

class

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

tuple

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

dataclass

```
@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
```

Overview

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`

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`

- 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`

- Manually increasing counts:

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

equivalent to

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

What can I do with a **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

Overview

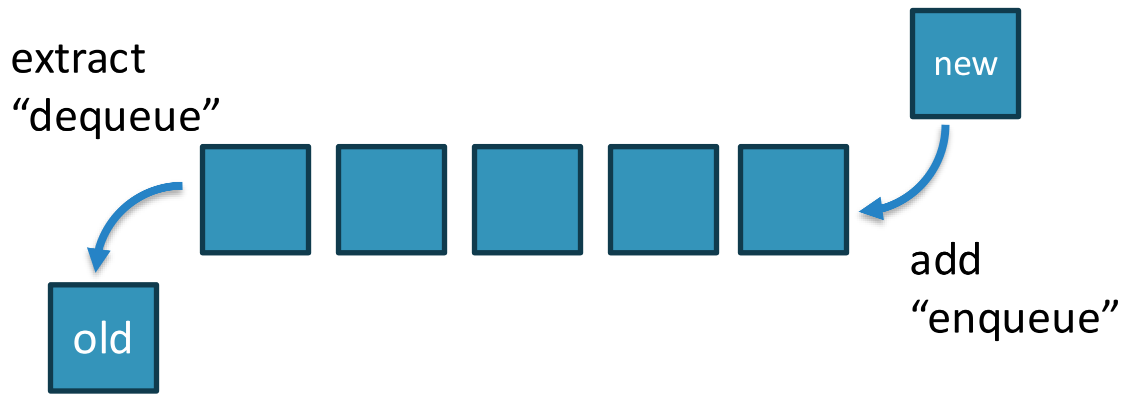
Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
• `dict` • `OrderedDict` • `defaultdict` • `ChainMap` • `MappingProxyType`	• `list` • `tuple` • `array` • `str` • `bytes` • `bytearray`	• `dict` • `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)

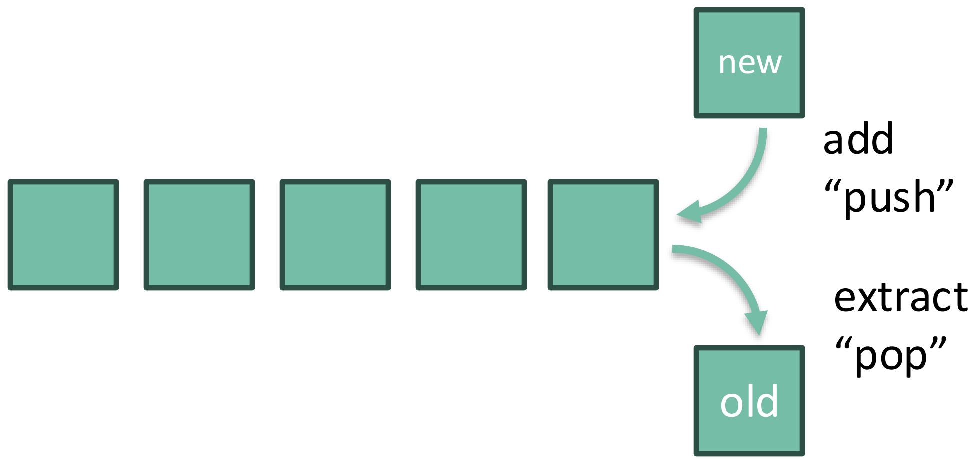
Some types are “improvements” of basic types

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

Queues and Stacks



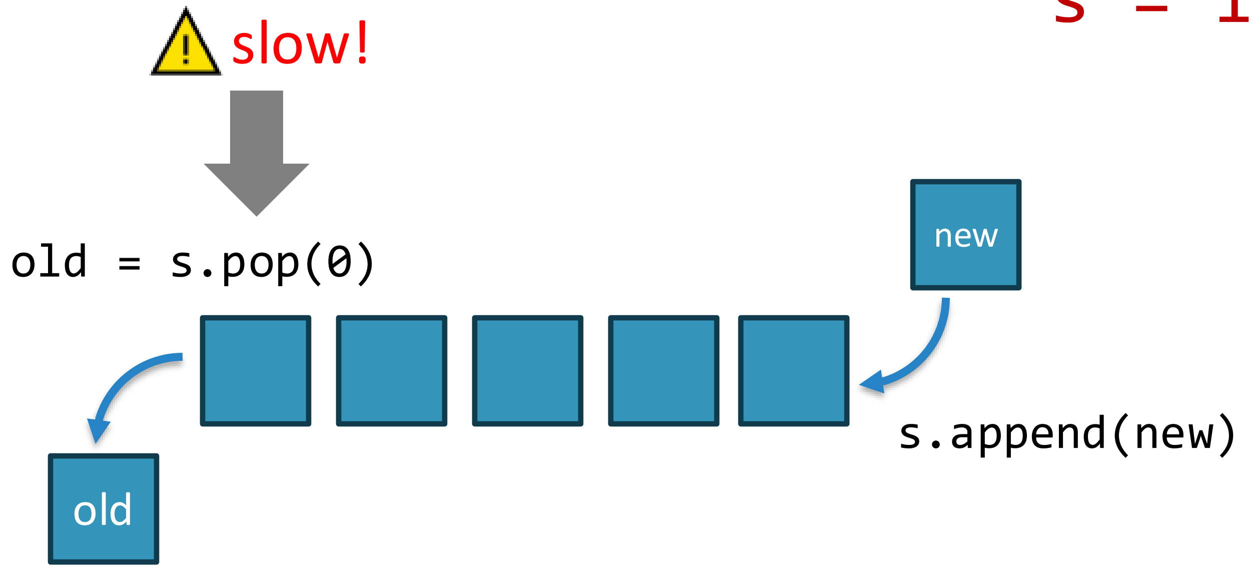
FIFO Queue – First-In First-Out



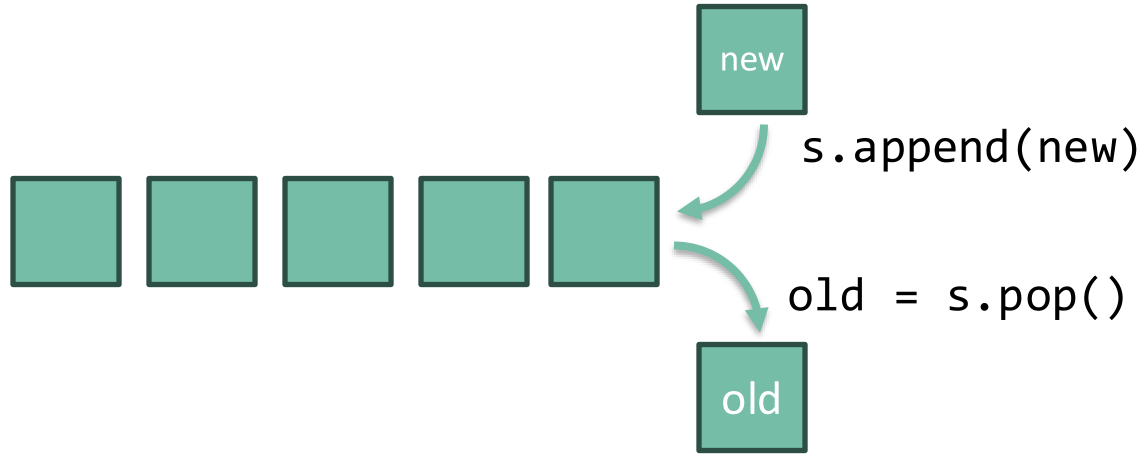
LIFO Stack – Larst-In First-Out

List implementations

`s = list()`



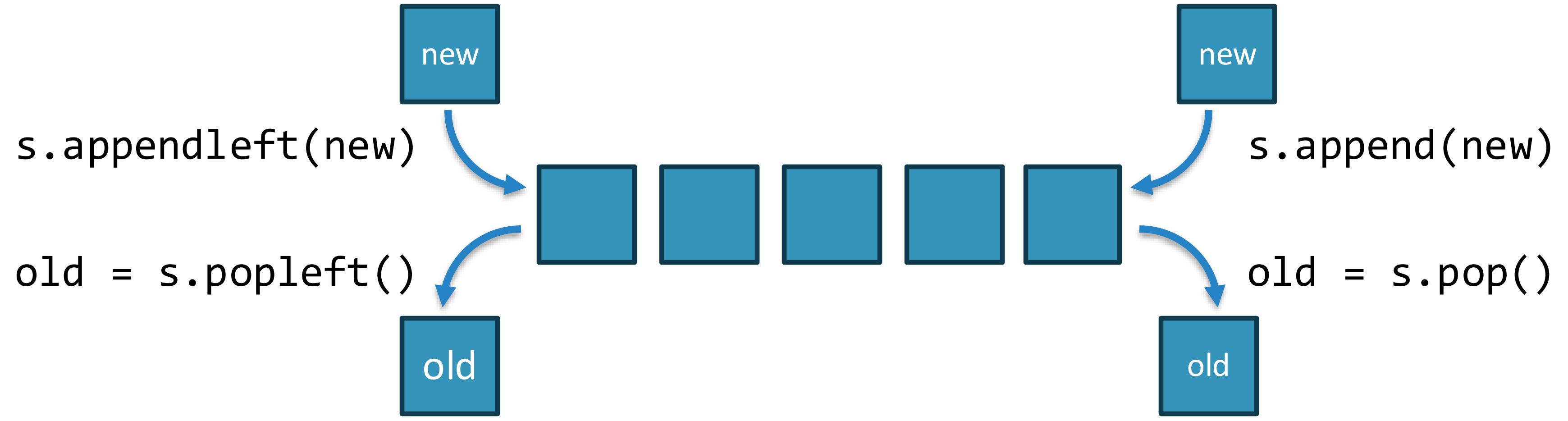
FIFO Queue – First-In First-Out



LIFO Stack – Larst-In First-Out

deque: double-ended queue

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



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)`	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]`	Access element (slower than lists)

`d.index(x), d.insert(i, x),`
`d.remove(x), d.reverse()`
	Same as lists

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

Overview

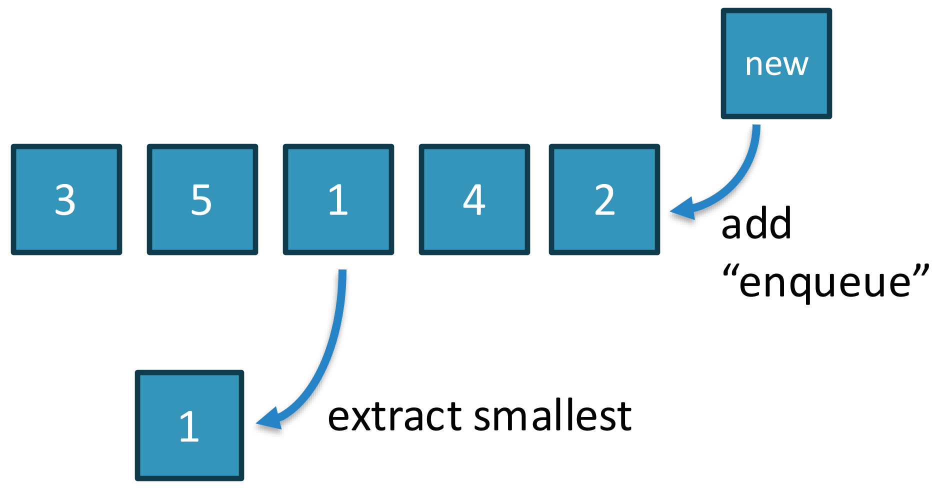
Dictionaries, Maps, Hash Tables	Array Data Structures	Records, Structs, Data Transfer Objects	Sets, Multisets	Stacks (LIFO)	Queues (FIFO)	Priority Queues
`dict`	`list`	`dict`	`set`	`list`	`list`	`list`
`OrderedDict`	`tuple`	`tuple`	`frozenset`	`deque`	`deque`	`heapq`
`defaultdict`	`array`	`class`	`Counter`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

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



- 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__)

heapq – uses plain lists

- `h = []`
- `h = heapify(iterable)`
- `len(h)`
- `len(h)==0`

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

`queue.PriorityQueue`

- `q = queue.PriorityQueue()`

- `q.qsize()`
- `q.empty()`
- `q.full()`
- `q.put(x)`
- `x = q.get_nowait()`

License