

# SORTING SEQUENCES



## Sorting and Sort Keys

Sorting a sequence of numbers is something easily understood

But we do have to consider the **direction** of the sort: **ascending** **descending**

Python provides a **sorted( )** function that will sort a given iterable

The default sort direction is **ascending**

The **sorted( )** function has an optional keyword-only argument called **reverse** which defaults to **False**

If we set it to **True**, then the sort will sort in **descending** order

But one really important thing we need to think about: **ordering**

→ obvious when sorting **real numbers**



## Sorting and Sort Keys

What about non-numerical values?

'a', 'b', 'c'

'A', 'a', 'B', 'b', 'C', 'c'

'hello', 'python', 'bird', 'parrot'

(0, 0) (1, 1) (2, 2)

(0, 0) (0, 1) (1, 0)

rectangle\_1, rectangle\_2, rectangle\_3

When items are pairwise comparable (< or >)  
we can use that ordering to sort

but what happens when they are not?

→ Sort Keys

strings are comparable so this is still OK  
although is 'a' < 'A' or 'a' > 'A' or 'a' == 'A'

True



# Sorting and Sort Keys

'b', 'x', 'a'      ASCII character codes      a → 97  
b → 98      ord('a') → 97  
x → 120

We now associate the ASCII numerical value with each character, and sort based on that value

items	'b'	'x'	'a'		'a'	'b'	'x'	
keys	98	120	97	→	97	98	120	
	'B'	'b'	'A'	'a'	'X'	'x'	'1'	'?'
	66	98	65	97	88	120	49	63
→	'1'	'?'	'A'	'B'	'X'	'a'	'b'	'x'
	49	63	65	66	88	97	98	120

You'll note that the sort keys have a natural sort order



## Sorting and Sort Keys

Let's say we want to sort a list of `Person` objects based on their age (assumes the `Person` class has an `age` property)

```
p1.age → 30  
p2.age → 15  
p3.age → 5  
p4.age → 32
```

item	p1	p2	p3	p4	→	p3	p2	p1	p4
keys	30	15	5	32		5	15	30	32

We could also **generate** the key value, for any given person, using a **function**

```
def key(p):  
    return p.age
```

```
key = lambda p: p.age
```

```
sort [p1, p2, p3, p4]
```

using sort keys generated by the function `key = lambda p: p.age`



## Sorting and Sort Keys

The sort keys need not be numerical → they just need to have a **natural sort order** (< or >)

item	'hello'	'python'	'parrot'	'bird'
keys	'o'	'n'	't'	'd'

← last character of each string

→

'bird'	'python'	'hello'	'parrot'
'd'	'n'	'o'	't'

```
key = lambda s: s[-1]
```



## Python's `sorted` function

That's exactly what Python's `sorted` function allows us to do

Optional keyword-only argument called `key`

if provided, `key` must be a `function` that for any given element in the sequence being sorted returns the `sort key`

The sort key does not have to be numerical → it just needs to be values that are themselves pairwise comparable (such as `<` or `>`)

If `key` is not provided, then Python will sort based on the `natural ordering` of the elements

i.e. they must be pairwise comparable (`<`, `>`)

If the elements are not pairwise comparable, you will get an exception



## Python's `sorted` function

```
sorted(iterable, key=None, reverse=False)
```

keyword-only

The `sorted` function:

- makes a `copy` of the iterable
- returns the sorted elements in a `list`
- uses a sort algorithm called `TimSort`
- a `stable` sort

→ named after Tim Peters      Python 2.3, 2002

<https://en.wikipedia.org/wiki/Timsort>

Side note: for the "natural" sort of elements, we can always think of the keys as the elements themselves

```
sorted(iterable) ↔ sorted(iterable, key=lambda x: x)
```



## Stable Sorts

A stable sort is one that **maintains** the **relative order** of items that have **equal keys** (or values if using natural ordering)

```
p1.age → 30  
p2.age → 15  
p3.age → 5  
p4.age → 32  
p5.age → 15
```

```
sorted((p1, p2, p3, p4, p5), key=lambda p: p.age)
```

```
→ [ p3    p2    p5    p1    p4 ]
```



keys **equal**

**p2** preceded **p5** in original tuple

→ **p2** precedes **p5** in sorted list



## In-Place Sorting

If the iterable is mutable, **in-place** sorting is **possible**

But that will depend on the particular type you are dealing with

Python's **list** objects support in-place sorting

The list class has a **sort( )** **instance method** that does in-place sorting

```
l = [10, 5, 3, 2]          id(l) → 0xFF42
```

```
l.sort( )
```

```
l → [2, 3, 5, 10]         id(l) → 0xFF42
```

Compared to **sorted( )**

- same **TimSort** algorithm
- same keyword-only arg: **key**
- same keyword-only arg: **reverse** (default is **False**)
- **in-place** sorting, does not copy the data
- only works on **lists** (it's a method in the **list** class)



# Code Exercises

© 2018 Matplotlib Academy