SORTING SEQUENCES

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

What about non-numerical values?

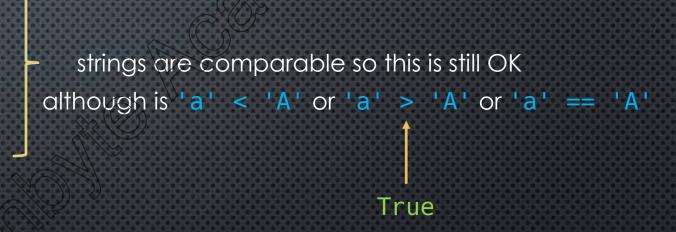
```
'a', 'b', 'c'
'A', 'a', 'B', 'b', 'C', 'c'
'hello', 'python', 'bird', 'parrot'
(0, 0) (1, 1) (2, 2)
(0, 0) (0, 1) (1, 0)
```

rectangle_1, rectangle_2, rectangle_3

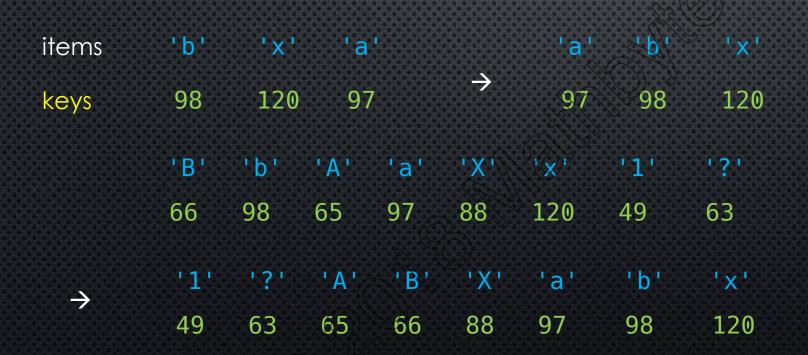
When items are pairwise comparable (< or >) we can use that ordering to sort

but what happens when they are not?

→ Sort Keys



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



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

Let's say we want to sort a list of **Person** objects based on their age

```
pl.age \Rightarrow 30 property)

p2.age \Rightarrow 15 p3.age \Rightarrow 5 p4.age \Rightarrow 32 item p1 p2 p3 p4 p3 p2 p1 p4 keys 30 15 5 15 30 32
```

(assumes the Person

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

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

sort[p1, p2, p3, p4]
key = lambda p: p.age
```

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

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

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

Clast character of each string

```
'bird' 'python' 'hello' 'parrot''
'd' 'n' 'o'
```

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

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) \leftarrow \rightarrow 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 \rightarrow 30
p2.age \rightarrow 15
p3.age \rightarrow 5
p4.age \rightarrow 32
p5.age \rightarrow 15
sorted((p1, p2, p3, p4, p5), key=lambda p: p.age)
      [ p3 p2 p5 p1 \\\\p4 ]
\rightarrow
                 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) \rightarrow 0xFF42
l.sort()
l \rightarrow [2, 3, 5, 10] id(l) \rightarrow 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