# Programming for Data Science: String and Lists

Instructor: Jakramate Bootkrajang

## Outlines

- String
- List
- Exercises

## String

- A sequence of characters
- Wrapped in single/double/triple quote
  - ‘this is string’
  - “this is also string”
  - ‘’’so is this’’’

## Indexing

- Because the elements of a string are a sequence, we can associate each element with an ***index***, a location in the sequence:
- positive values count up from the left, beginning with index 0
- negative values count down from the right, starting with -1

## Indexing Example

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| characters | H | e | l | l | o | | W | o | r | l | d |
| index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| | | | | | | | | | ... | −2 | −1 |

**FIGURE 4.1** The index values for the string `'Hello World'`.

## Accessing an element

- A particular element of the string is accessed by the index of the element surrounded by square brackets [ ]
- `hello_str = 'Hello World'`
- `print(hello_str[1])` => prints e
- `print(hello_str[-1])` => prints d
- `print(hello_str[11])` => ERROR

## Slicing

- slicing is the ability to select a subsequence of the overall sequence
  - uses the syntax `[start : finish]`, where:
  - `start` is the index of where we start the subsequence
  - `finish` is the index of **<u>one after</u>** where we end the subsequence
- if either `start` or `finish` are not provided, it defaults to the beginning of the sequence for `start` and the end of the sequence for `finish`

## Slicing Example

```
helloString[6:10]
```

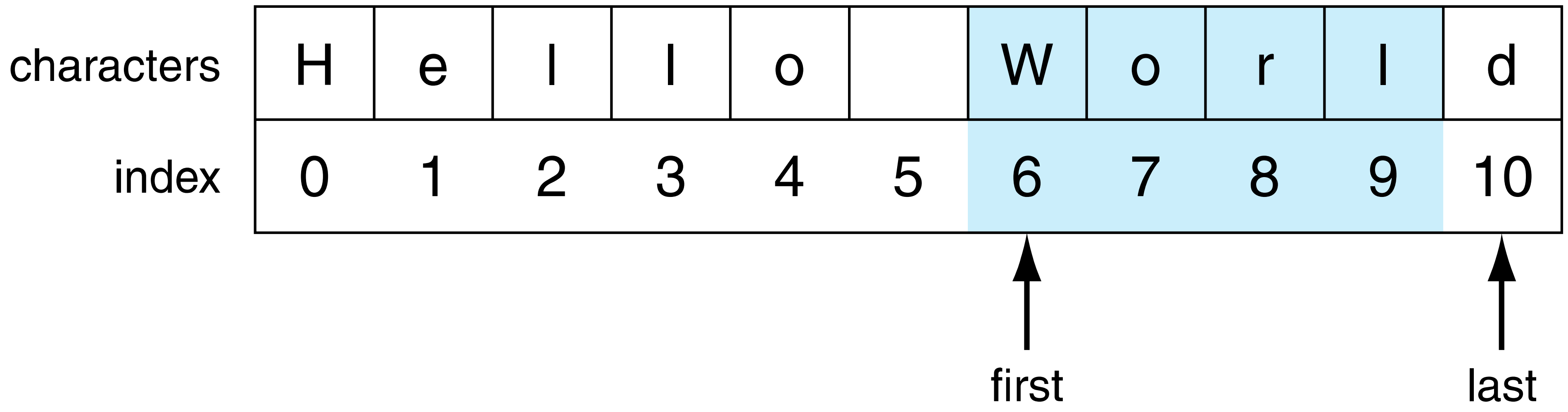


FIGURE 4.2 Indexing subsequences with slicing.

# Slicing Example [2]

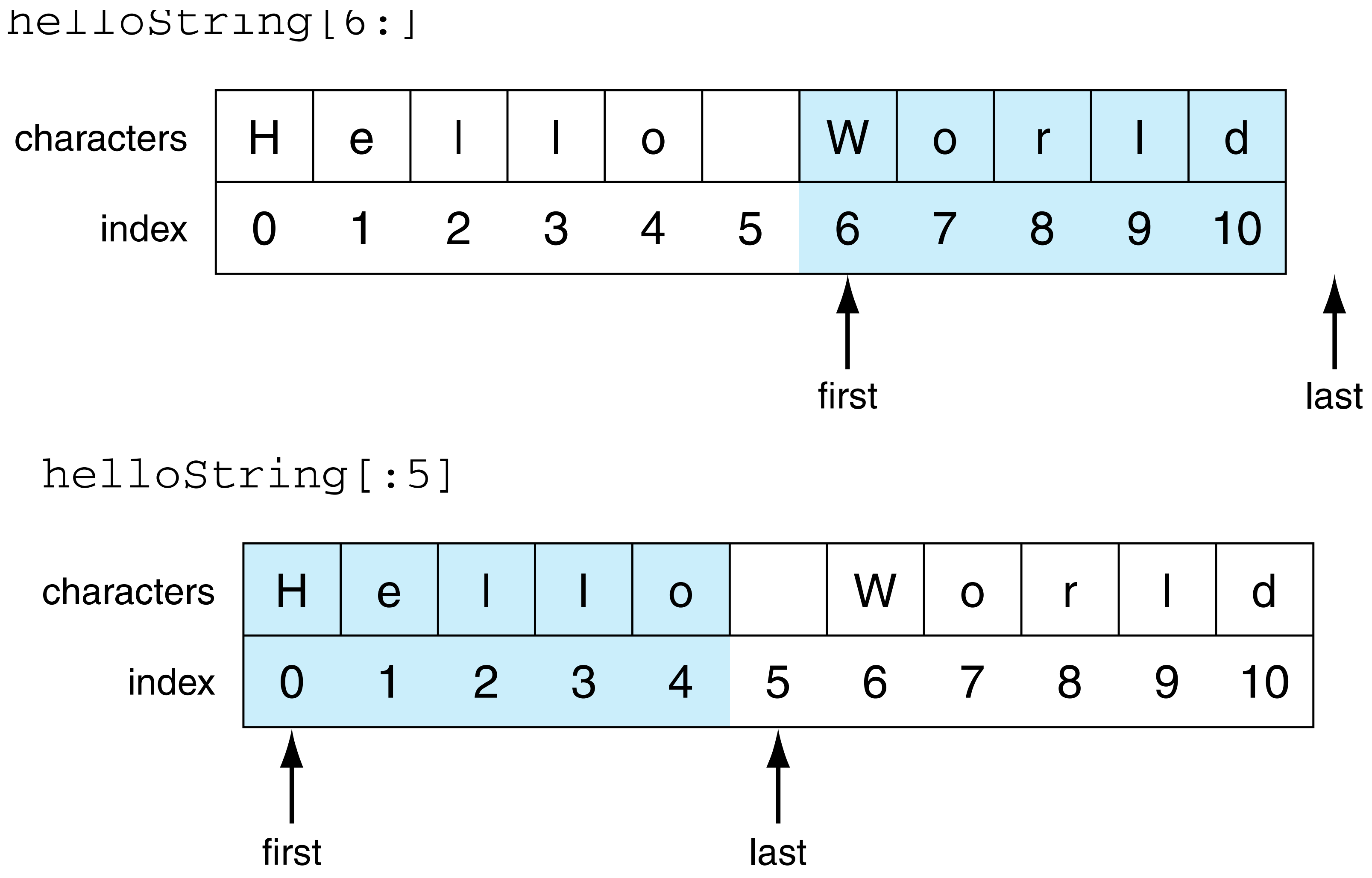


**FIGURE 4.3** Two default slice examples.

# Slicing example [3]

```
helloString[-1]
```

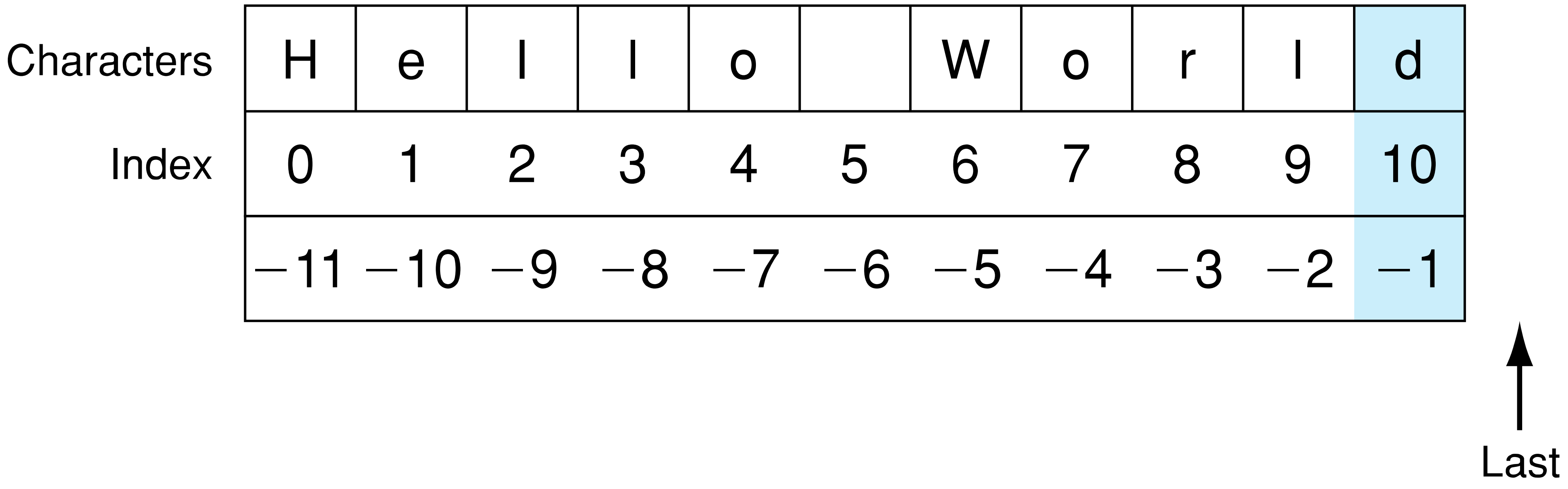


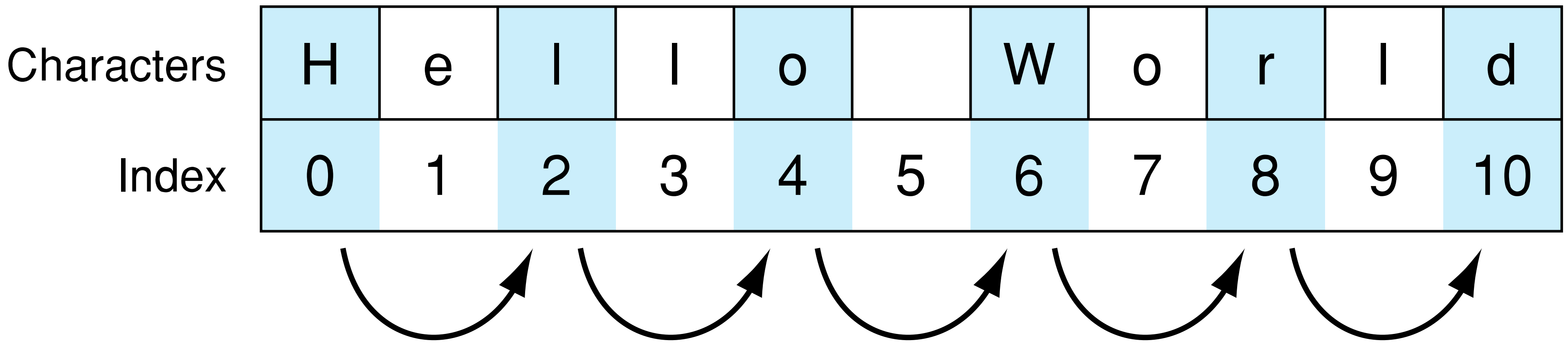
**FIGURE 4.4** Negative indices.

## Extended Slicing

- also takes three arguments:
  `[start:finish:countBy]`
- defaults are:
  `start` is beginning, `finish` is end, `countBy` is 1
- `my_str = 'hello world'`
- `my_str[0:11:2] ⇒ 'hlowrd'`
  every other letter

# Extended slicing example

```
helloString[::2]
```



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Characters | H | e | l | l | o | | W | o | r | l | d |
| Index | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**FIGURE 4.6** Slicing with a step.

## Python string idioms

- idioms are python “phrases” that are used for a common task that might be less obvious to non-python folk
- how to make a copy of a string:

```
my_str = 'hi mom'
new_str = my_str[:]
```

- how to reverse a string

```
my_str = "madam I'm adam"
reverseStr = my_str[::-1]
```

## Strings are iterable

- The for loop iterates through each element of a sequence in order. For a string, this means character by character:

```
>>> for char in 'Hi mom':
        print(char, type(char))


H <class 'str'>
i <class 'str'>
  <class 'str'>
m <class 'str'>
o <class 'str'>
m <class 'str'>
>>>
```

## Basic String Operations

- `s = 'spam'`
  length operator len()
- `len(s)` ⇒ 4
  + is concatenate
- `new_str = 'spam' + '-' + 'spam-'`
- `print(new_str)` ⇒ spam-spam-
  * is repeat, the number is how many times
- `new_str * 3` ⇒
  `'spam-spam-spam-spam-spam-spam-'`

## String comparisons, single char

- Python 3 uses the Unicode mapping for characters.
- Allows for representing non-English characters
- UTF-8, subset of Unicode, takes the English letters, numbers and punctuation marks and maps them to an integer.
- Single character comparisons are based on that number

## Comparison example

- It makes sense to compare within a sequence (lower case, upper case, digits).

```
'a' < 'b'    → True
'A' < 'B'    → True
'1' < '9'    → True
```

- Can be weird outside of the sequence

```
'a' < 'A'    → False
'a' < '0'    → False
```

## The whole string

- Compare the first element of each string
  - if they are equal, move on to the next character in each
  - if they are not equal, the relationship between those to characters are the relationship between the string
  - if one ends up being shorter (but equal), the shorter is smaller

## Example

- `'a' < 'b'` → `True`
- `'aaab' < 'aaac'`

  first difference is at the last char. `'b'<'c'` so `'aaab'` is less than `'aaac'`. `True`
- `'aa' < 'aaz'`

  The first string is the same but shorter. Thusit is smaller. `True`

# Membership operations

- can check to see if a substring exists in the string, the `in` operator. Returns True or False
- `my_str = 'aabbccdd'`
- `'a' in my_str` ⇒ `True`
- `'abb' in my_str` ⇒ `True`
- `'x' in my_str` ⇒ `False`

# Strings are immutable

- strings are immutable, that is you cannot change one once you make it:

  ```
  a_str = 'spam'
  a_str[1] = 'l' → ERROR
  ```

- However, you can use it to make another string (copy it, slice it, etc.)

  ```
  new_str = a_str[:1] + 'l' + a_str[2:]
  a_str →  'spam'
  new_str →'slam'
  ```

# String methods

| | |
|---|---|
| capitalize( ) | lstrip( [*chars*] ) |
| center( *width* [, *fillchar*] ) | partition( *sep*) |
| count( *sub* [, *start* [, *end*] ] ) | replace( *old*, *new* [, *count*] ) |
| decode( [*encoding* [, *errors*] ] ) | rfind( *sub* [,*start* [,*end*] ] ) |
| encode( [*encoding* [,*errors*] ] ) | rindex( *sub* [, *start* [, *end*] ] ) |
| endswith( *suffix* [, *start* [, *end*] ] ) | rjust( *width* [, *fillchar*] ) |
| expandtabs( [*tabsize*] ) | rpartition(*sep*) |
| find( *sub* [, *start* [, *end*] ] ) | rsplit( [*sep* [,*maxsplit*] ] ) |
| index( *sub* [, *start* [, *end*] ] ) | rstrip( [*chars*] ) |
| isalnum( ) | split( [*sep* [,*maxsplit*] ] ) |
| isalpha( ) | splitlines( [*keepends*] ) |
| isdigit( ) | startswith( *prefix* [, *start* [, *end*] ] ) |
| islower( ) | strip( [*chars*] ) |
| isspace( ) | swapcase( ) |
| istitle( ) | title( ) |
| isupper( ) | translate( *table* [, *deletechars*] ) |
| join(*seq*) | upper( ) |
| lower( ) | zfill( *width*) |
| ljust( width [, *fillchar*] ) | |

**TABLE 4.2** Python String Methods

## List

- A list is an ordered sequence of items.
- Items need not be of the same types
  - List of integer and character is possible
- Items are wrapped in square brackets [ ], separated by commas
  - A = [1,2,3,4,”five”]

# Constructing a list

- Using square brackets
  - L = [‘a’,’b’,’c’]
- Using list(x) function
  - X must be iterable: for example string
  - list(“hello”)
    - Result is [‘h’,’e’,’l’,’l’,’o’]

## List and String are similar

- Concatenate using +
- Repeat using *
- indexing (the [ ] operator)
- slicing ([:])
- membership (the in operator)
- len (the length operator)

## List operators

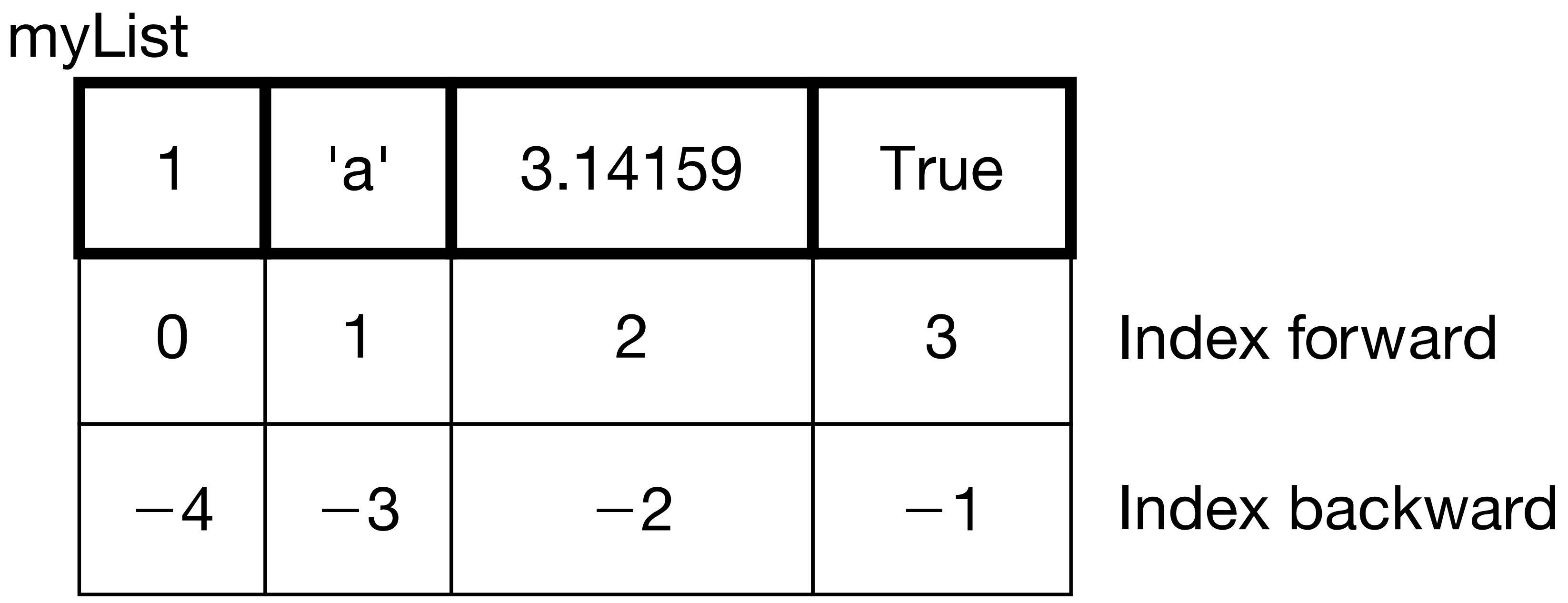
- `[1, 2, 3] + [4]` ⇒ `[1, 2, 3, 4]`
- `[1, 2, 3] * 2` ⇒ `[1, 2, 3, 1, 2, 3]`
- `1 in [1, 2, 3]` ⇒ `True`
- `[1, 2, 3] < [1, 2, 4]` ⇒ `True`
  compare index to index, first difference determines the result

## Difference between list and string

- lists can contain a mixture of any python object, strings can only hold characters
  - 1,"bill",1.2345, True
- lists are mutable, their values can be changed, while strings are immutable

## List indexing

```
myList = [1, 'a', 3.14159, True]
```



myList

|  |  |  |  |  |
|---|---|---|---|---|
| 1 | 'a' | 3.14159 | True |  |
| 0 | 1 | 2 | 3 | Index forward |
| −4 | −3 | −2 | −1 | Index backward |

```
myList[1] ⟶ 'a'

myList[:3] ⟶ [1, 'a', 3.14159]
```

**FIGURE 7.1** The structure of a list.

## List of lists

- `my_list = ['a', [1, 2, 3], 'z']`
  - What is the second element (index 1) of that list? Another list.
- `my_list[1][0] # apply left to right`
- `my_list[1] ⇒ [1, 2, 3]`
- `[1, 2, 3][0] ⇒ 1`

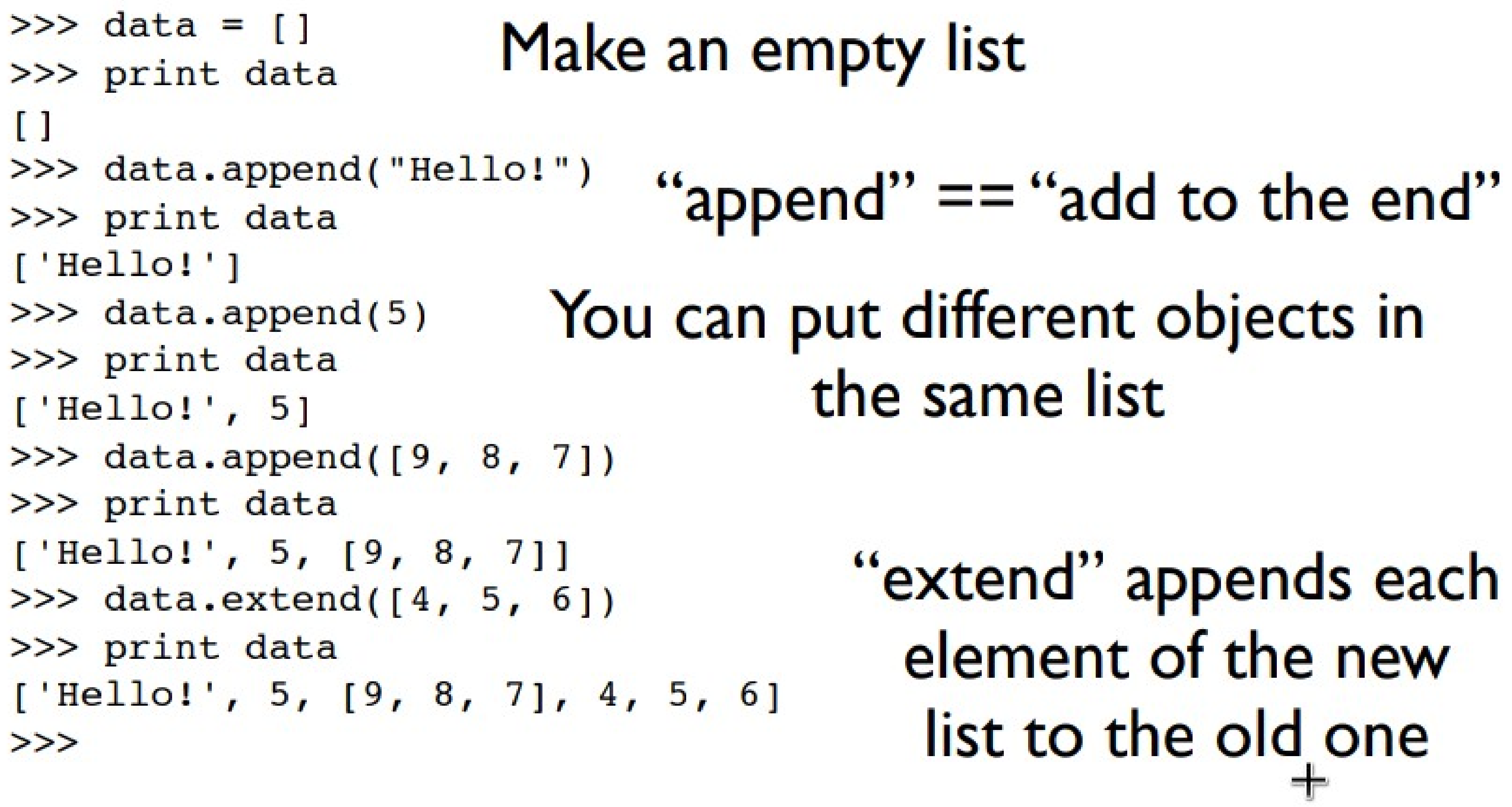
# Function on list

- `len(lst)`: number of elements in list (top level). `len([1, [1, 2], 3])` ⇒ `3`
- `min(lst)`: smallest element. Must all be the same type!
- `max(lst)`: largest element, again all must be the same type
- `sum(lst)`: sum of the elements, numeric only

## Lists’ methods

- `my_list.append()`, `my_list.extend()`
- `my_list.pop()`
- `my_list.insert()`, `my_list.remove()`
- `my_list.sort()`
- `my_list.reverse()`

## In details



```
>>> data = []
>>> print data
[]
>>> data.append("Hello!")
>>> print data
['Hello!']
>>> data.append(5)
>>> print data
['Hello!', 5]
>>> data.append([9, 8, 7])
>>> print data
['Hello!', 5, [9, 8, 7]]
>>> data.extend([4, 5, 6])
>>> print data
['Hello!', 5, [9, 8, 7], 4, 5, 6]
>>>
```

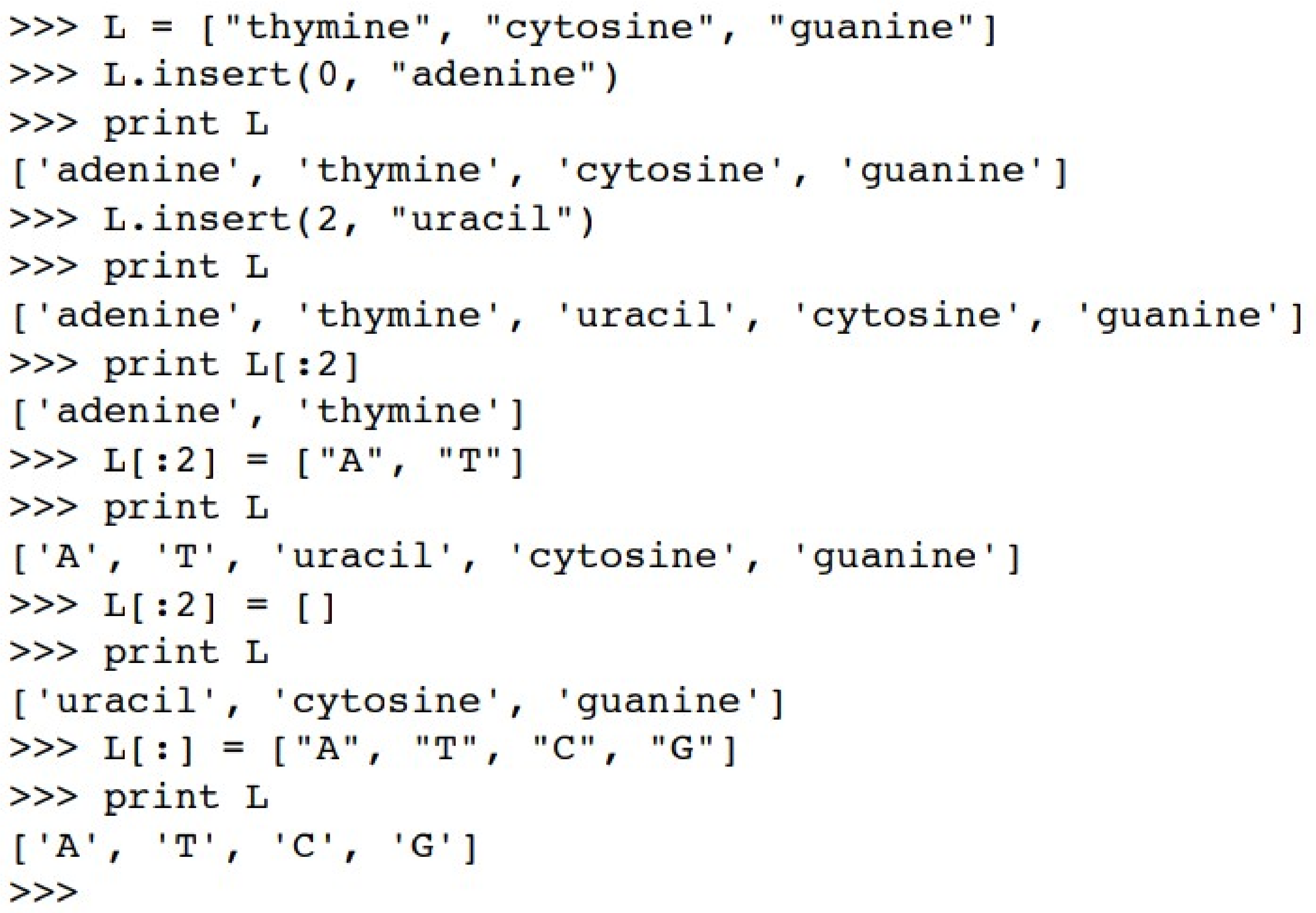
Make an empty list

“append” == “add to the end”

You can put different objects in the same list

“extend” appends each element of the new list to the old one

## In details [2]



```
>>> L = ["thymine", "cytosine", "guanine"]
>>> L.insert(0, "adenine")
>>> print L
['adenine', 'thymine', 'cytosine', 'guanine']
>>> L.insert(2, "uracil")
>>> print L
['adenine', 'thymine', 'uracil', 'cytosine', 'guanine']
>>> print L[:2]
['adenine', 'thymine']
>>> L[:2] = ["A", "T"]
>>> print L
['A', 'T', 'uracil', 'cytosine', 'guanine']
>>> L[:2] = []
>>> print L
['uracil', 'cytosine', 'guanine']
>>> L[:] = ["A", "T", "C", "G"]
>>> print L
['A', 'T', 'C', 'G']
>>>
```

## Iteration

- You can iterate through the elements of a list like you did with a string:

```
>>> my_list = [1,3,4,8]
>>> for element in my_list:     # iterate through list elements
        print(element ,end=' ') # prints on one line

1 3 4 8
>>>
```

## Lists are mutable

- Unlike strings, lists are mutable. You ***can*** change the object's contents!

```
my_list = [1, 2, 3]
my_list[0] = 127
print(my_list) ⇒ [127, 2, 3]
```

## Split

- The string method split() generates a sequence of characters by splitting the string at certain split-characters.

```
split_list = 'this is a test'.split()
split_list
    ⇒ ['this', 'is', 'a', 'test']
```

## Join

- A string join() method takes list as input argument and join each element in the list with the string

```
split_list = ['this', 'is', 'a', 'test']
“+”.join(split_list)
     ⇒ “this+is+a+test”
```

## Sorting

- Only lists have a built in sorting method. Thus you often convert your data to a list if it needs sorting

```
my_list = list('xyzabc')
my_list →['x','y','z','a','b','c']
my_list.sort()   # no return
my_list →
['a', 'b', 'c', 'x', 'y', 'z']
```

## Exercise 1: Anagram

- Anagrams are words that contain the same letters arranged in a different order. For example: 'iceman' and 'cinema'
- Write a program that asks user for two strings and check if they are anagram

## Exercise 2: Sentiment Analysis of reviews

- Suppose we want to automatically check if user’s review is positive or negative
- Positive review usually contains these positive keyword
  - Great, good, nice, fun, enjoy
- Negative review contains
  - Boring, bad, poor, worse, fail

## Exercise 2 (cont)

- Write a program that takes as input a (long) string representing one review and check if the review is positive or negative based on the keywords given.
- For each positive keyword found the review get +1 score (count the first time the keyword is found only)
- For each negative keyword found the review get -1 score

## Exercise 2 (cont)

- A review with more than +2 is considered positive
- A review with less than -2 is considered negative
- Else a review is considered neutral

## Data

- I just discovered this show a few days ago and totally binge-watched every episode. It's sweet and concise, celebrating love for movies and the creative process behind making them, from actors, directors, producers, sfx artists, you name it. I think the hosts are terrific and the joy they bring into each episode, whether they're on location somewhere in Hollywood or interviewing guests, made me want to keep watching. They are able to bring out a youthful and excited side from each guest, and it's fun to see filmmakers talk about movies and tv shows they love and then geek out playing movie games. Guests not only talk about current work, but the hosts also talk with them about past projects; for example, it was really cool when Tim immediately out of the gate asked Alessandro Nivola about working on Face/Off, and watching Kerri smoke a cigar with Norman Lear was badass and had me cracking up. It's fun to watch an episode and then turn around and watch trailers for movies they talked about that I hadn't heard of before. Added to the watch list. Done. This show is a great way to spend a lighthearted ten-fifteen minutes. I hope there are many more seasons ahead.

## Data

- Twenty plus years after his boring character debut on Mall Rats, the same turgid character continues to linger on, pretending to be interesting, but failing horribly. Unfortunately for the world, watching Kevin Smith is like watching fungus grow on a rotten log. Very boring and ponderous. Nothing that Smith says is fun or exciting. It is just Kevin Smith being his homie character from the 1990s. Same backwards cap, same goofy grin, same empty observations. A big waste of time.