# CSCI 141

## Lecture 18
## Strings: Slicing, String Methods, Comparison and `in` operators

# Announcements

# Announcements

- QOTD is out of 5 points (whoops)

# Announcements

- QOTD is out of 5 points (whoops)
- QOTD explanation is linked from the explanation of the question on Canvas

# Announcements

- QOTD is out of 5 points (whoops)
- QOTD explanation is linked from the explanation of the question on Canvas
- A4 is due next Wednesday.

# Announcements

- QOTD is out of 5 points (whoops)
- QOTD explanation is linked from the explanation of the question on Canvas
- A4 is due next Wednesday.
    - Monday's a holiday, so (I assume) no mentor hours that day!

# Announcements

- QOTD is out of 5 points (whoops)
- QOTD explanation is linked from the explanation of the question on Canvas
- A4 is due next Wednesday.
  - Monday's a holiday, so (I assume) no mentor hours that day!
  - Monday's a holiday, so no labs next week!

# Goals

- Know how to use slicing to get substrings
- Know how to use a few of the basic methods of string objects:
  - `upper, lower, find, replace`
- Understand the behavior of the following operators on strings:
  - <, >, ==, !=, in, and not in
  - Know strings are compared using lexicographic ordering
- Understand the meaning and implications of strings being immutable objects.

# Inclusive Learning Environment: Redux

# Inclusive Learning Environment: Redux

- Remember Lecture 1?

# Inclusive Learning Environment: Redux

- Remember Lecture 1?

# Inclusive Learning Environment: Redux

- Remember Lecture 1?

- Anyone felt like this at any point in the course?

# Inclusive Learning Environment: Redux

- Remember Lecture 1?

- Anyone felt like this at any point in the course? (I have...)

# Inclusive Learning Environment: Redux

- My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
    - You learn by **doing**.
    - This involves **making mistakes** and **asking questions**.
    - **Nobody** writes perfect code on the first try, not even professionals.
- Keep this in mind when:

# Inclusive Learning Environment: Redux

- My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
  - You learn by **doing**.
  - This involves **making mistakes** and **asking questions**.
  - **Nobody** writes perfect code on the first try, not even professionals.
- Keep this in mind when:

This is you.

# Inclusive Learning Environment: Redux

- My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
    - You learn by **doing**.
    - This involves **making mistakes** and **asking questions**.
    - **Nobody** writes perfect code on the first try, not even professionals.
- **Also** keep this in mind when:

# Inclusive Learning Environment: Redux

- My goal: A learning environment in which everyone feels comfortable, curious, and excited to learn.
  - You learn by **doing.**
  - This involves **making mistakes** and **asking questions.**
  - **Nobody** writes perfect code on the first try, not even professionals.
- **Also** keep this in mind when:

This is you.

# People from underrepresented groups face extra obstacles.

# People from underrepresented groups face extra obstacles.

This claim is (heavily) backed by scientific research.

# People from underrepresented groups face extra obstacles.

Disclaimer: I am not a psychologist

This claim is (heavily) backed by scientific research.

# People from underrepresented groups face extra obstacles.

Disclaimer: I am not a psychologist

This claim is (heavily) backed by scientific research.

**Stereotype threat**:
stereotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

# People from underrepresented groups face extra obstacles.

Disclaimer: I am not a psychologist

This claim is (heavily) backed by scientific research.

**Stereotype threat**:
stereotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

**Implicit bias:**
well-intentioned people exhibit biases that they're not even aware they have.

# People from underrepresented groups face extra obstacles.

Disclaimer: I am not a psychologist

This claim is (heavily) backed by scientific research.

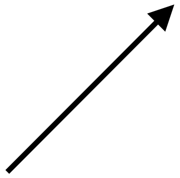
**Stereotype threat**:
stereotypes become self-fulfilling when the subjects of the stereotype are conscious of them.

**Implicit bias:**
well-intentioned people exhibit biases that they're not even aware they have.

**Impostor syndrome**:
Successes are attributed to luck
Failures are attributed to ability

# Happenings

Computer Science EID Community Office Hours



**Equity, Inclusion, and Diversity**

Dr. Moushumi Sharmin is new Community Ambassador for CS.

Community office hours for Fall 2019 are **Wednesdays from 10:30-11:30 am** (CF 465). Students, faculty and staff are welcome to discuss issues related to **equity, inclusion, and diversity**.

**VIKING UNION - MPR**

**NOVEMBER 14**

**5 - 7:30 PM**

- Free Food
- Photobooth
- Hands on Science
- Story Gallery
- Raffle Prizes

MIX IT UP

SCIENCE PAST

THE GOOD AND THE BAD

MIX IT UP: Inclusion in STEM Mixer

# Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| | S | u | m | m | e | r | | i | s | | n | e | a | r |
| Index: | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

# Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  | S | u | m | m | e | r |  | i | s |  | n | e | a | r |
| Index: | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

**Two possible ways to remember how this works:**

# Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| | S | u | m | m | e | r | | i | s | | n | e | a | r |
| Index: | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

**Two possible ways to remember how this works:**

`a_string[-5]`

is equivalent to

`a_string[len(a_string)-5]`

# Last time: Indexing Strings, Negative Indices

Negative indices count backwards from len(s):

| | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| | S | u | m | m | e | r | | i | s | | n | e | a | r |
| Index: | -14 | -13 | -12 | -11 | -10 | -9 | -8 | -7 | -6 | -5 | -4 | -3 | -2 | -1 |

**Two possible ways to remember how this works:**

-1 is always the last character, and indices count backwards from there.

`a_string[-5]`

is equivalent to

`a_string[len(a_string)-5]`

# QOTD

```
def flop(value, number):
    output = ""

    for i in range(number, 0, -1):
        output = output + value[i-1]

    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]

    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if `a` is an integer?

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

`mit one` `t onime` `emit on`

`nomite` `on time` `timeno time`

# QOTD

```python
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

[✓ `mit one`]    `t onime`    `emit on`

`nomite`    `on time`    `timeno time`

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

✓`mit one`

`t onime`

`emit on`

✗`nomite`

`on time`

`timeno time`

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

✓ mit one

✓ t onime

emit on

✗ nomite

on time

timeno time

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

✓`mit one`

✓`t onime`

`emit on`

✗`nomite`

✓`on time`

`timeno time`

# QOTD

```python
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

✓`mit one` ✓`t onime` ✓`emit on`

✗`nomite` ✓`on time` `timeno time`

# QOTD

```
def flop(value, number):
    output = ""
    # Reverse the first "number" characters in value:
    for i in range(number, 0, -1):
        output = output + value[i-1]
    # Append the substring from number to the end:
    for i in range(number, len(value)):
        output = output + value[i]
    return output
```

Which of the following is **not** a possible return value of `flop("no time", a)` if a is an integer?

✓ `mit one`　　✓ `t onime`　　✓ `emit on`

✗ `nomite`　　✓ `on time`　　✓ `timeno time`

# Worksheet - Exercise 1

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with the first
        # symbol removed. If there is no # in
        the string, return input unchanged.
    """
```

Hint: try a `while` loop!

```
# Example:

remove_comments("a = b # assign b to a”))
# => "a = b “
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Val | a | b | c | d | e | f | g | h | i | j |

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

What if I want to "index" more than one character at a time?

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

What if I want to "index" more than one character at a time?

```
alph[???] # => "cdef"
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

**Slicing syntax:** `string[`*start*`:`*end*`]`

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character ↘

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

str

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** string[*start*:*end*]

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

str

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[start:end]`

just like the `range` function:
the end index is **not** included

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*start*`:`*end*`]`

just like the `range` function: the end index is **not** included

```
alph[2:6] # => "cdef"
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[start:end]`

just like the `range` function:
the end index is **not** included

```
alph[2:6] # => "cdef"

alph[0:10] # => "abcdefghij"
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

just like the `range` function:
the end index is **not** included

```
alph[2:6] # => "cdef"

alph[0:10] # => "abcdefghij"

alph[5:-2]
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

str

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

just like the `range` function: the end index is **not** included

```
alph[2:6] # => "cdef"

alph[0:10] # => "abcdefghij"

alph[5:-2]  # => "fgh"
```

# Slicing: indexing substrings

`alph = "abcdefghij"`

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[start:end]`

# Slicing: indexing substrings

`alph = "abcdefghij"`

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

If omitted, *`start`*
defaults to 0

# Slicing: indexing substrings

```
alph = "abcdefghij"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

If omitted, *`start`* defaults to 0

If omitted, *`end`* defaults to `len(string)`

# Slicing: indexing substrings

```
alph = "abcdefghij"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

If omitted, *`start`* defaults to 0

If omitted, *`end`* defaults to `len(string)`

```
alph[:4] # => “abcd"
```

# Slicing: indexing substrings

```
alph = "abcdefghij"
```

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Val | a | b | c | d | e | f | g | h | i | j |

index of first character

1 + index of last character

**Slicing syntax:** `string[`*`start`*`:`*`end`*`]`

If omitted, *`start`* defaults to 0

If omitted, *`end`* defaults to `len(string)`

```
alph[:4] # => “abcd"
alph[5:] # => "fghij"
```

# String Slicing: Demo

# String Slicing: Demo

- `s = "fibonacci"`
- Positive indices: s[1:3]
- Negative indices!? s[-4:9]
- Leaving out start/endpoint: s[:6], s[4:]
- Indices past the end in a slice: s[1:21]
- Single indices past the end: s[9], s[21]
- Loop over a slice of a string

```
for c in s[2:6]:
    print(c, "!", sep="", end="")
```

# String Slicing: Exercise

`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Which of these evaluates to `"in"`?

A. `last_name[7:8]`
B. `last_name[6:-1]`
C. `last_name[-3:]`
D. `last_name[-2:8]`

# String Slicing: Exercise

```
last_name = "Wehrwein"
```

str

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

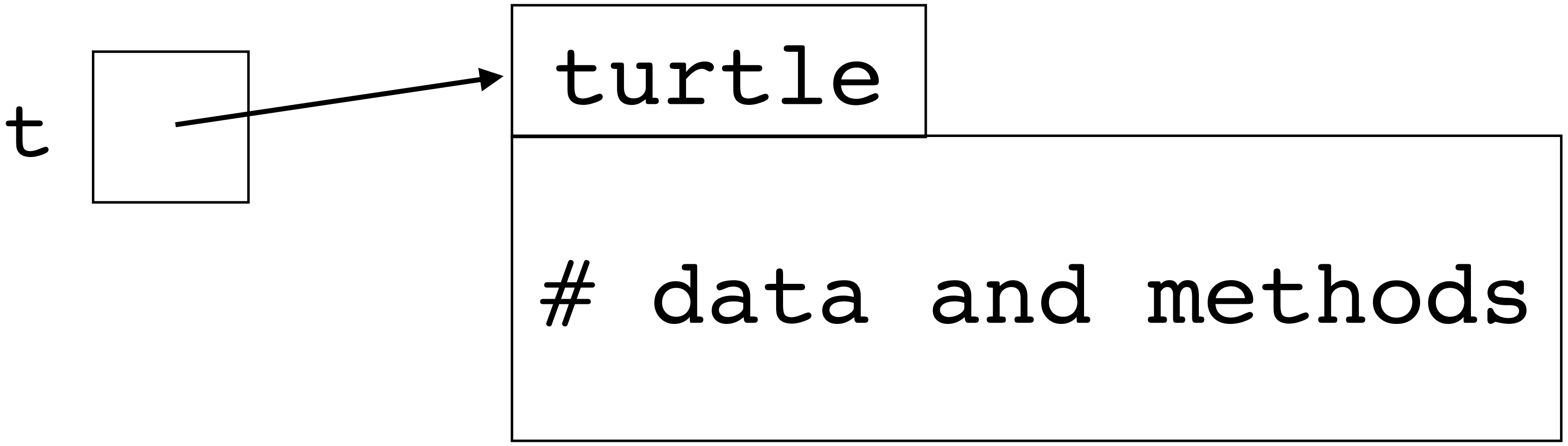
Which of these evaluates to `"in"`?

A. `last_name[7:8]`
B. `last_name[6:-1]`
C. `last_name[-3:]`
D. `last_name[-2:8]`

# Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:



t

turtle

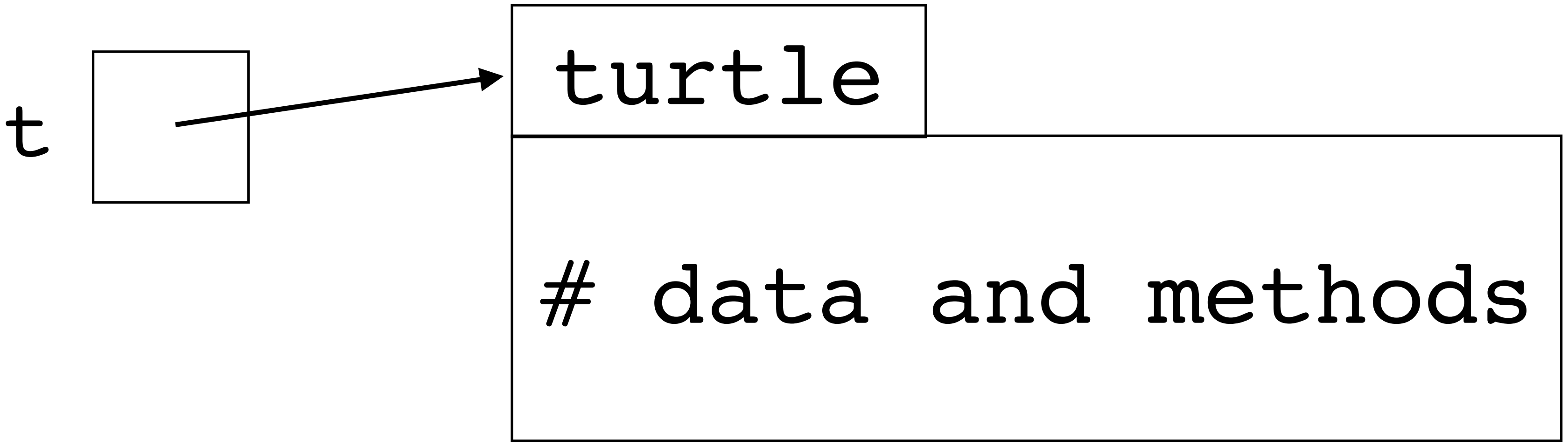
`# data and methods`

# Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```
t = turtle.Turtle()
```



t

turtle

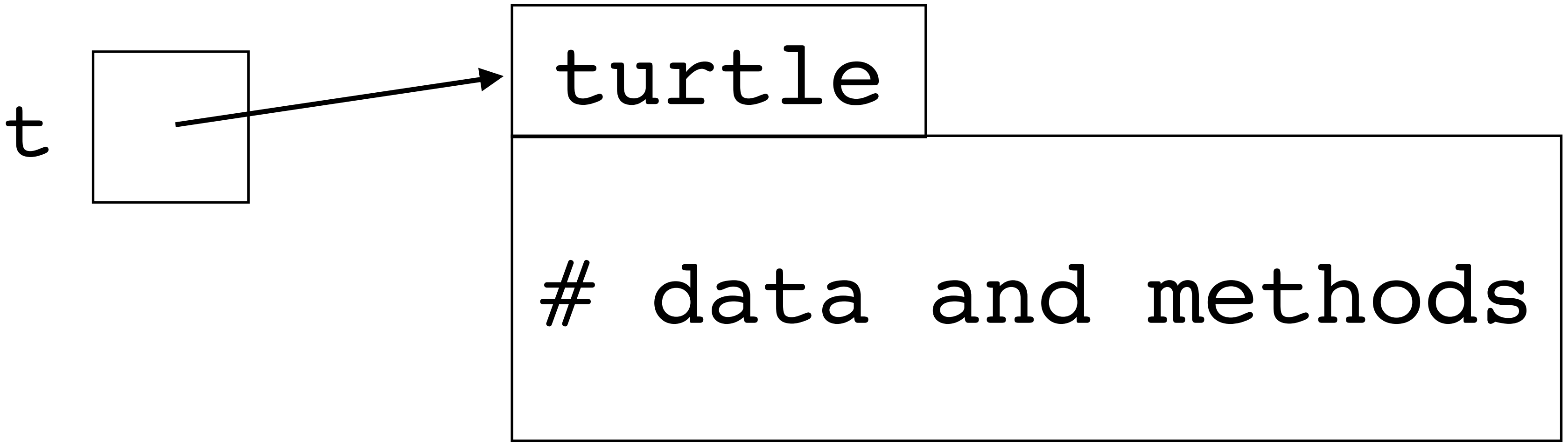
# data and methods

# Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

```
t = turtle.Turtle()
t.forward(100)
```



t

turtle

# data and methods

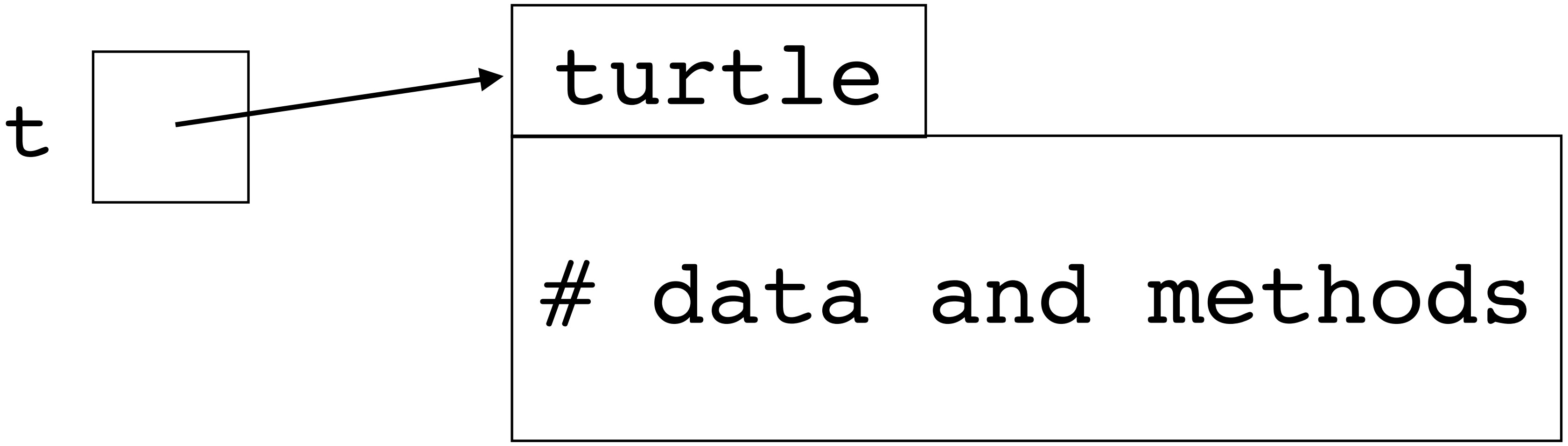
# Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:

turtle module

```
t = turtle.Turtle()
t.forward(100)
```



# Strings are objects.

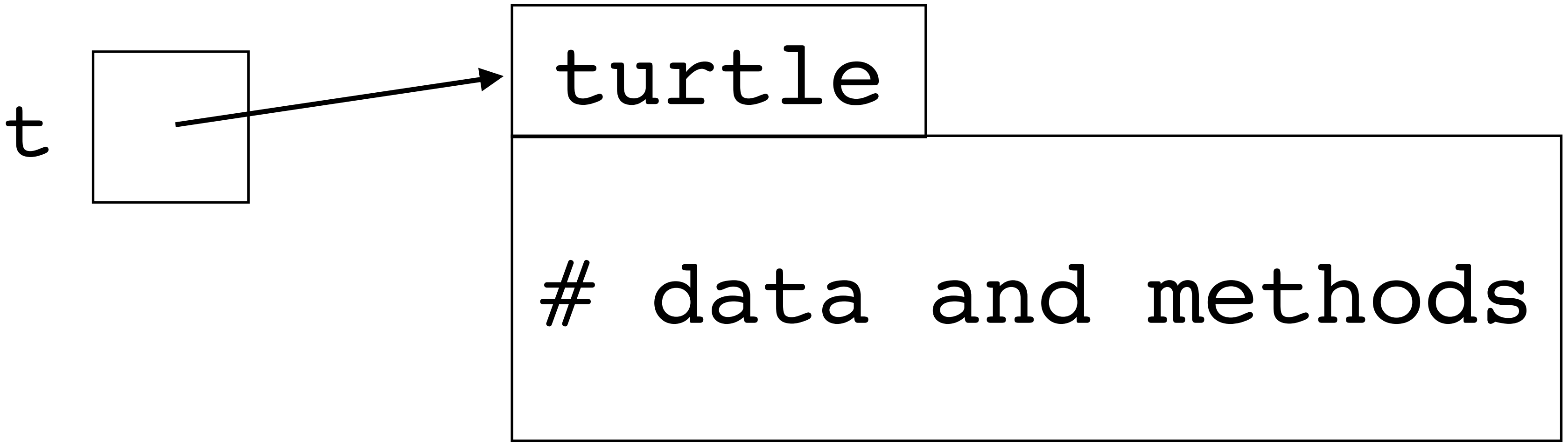
We've seen other objects before: turtles!

Turtles had methods:

turtle module

module function
(turtle constructor)

```
t = turtle.Turtle()
t.forward(100)
```



t

turtle

`# data and methods`

# Strings are objects.

We’ve seen other objects before: turtles!

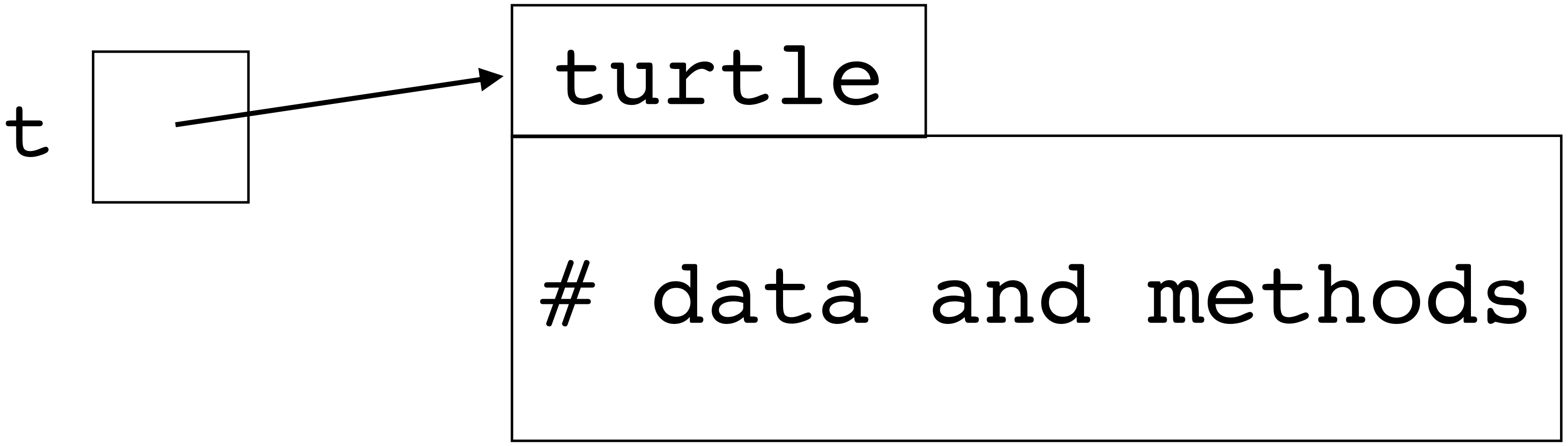
Turtles had methods:

turtle module

module function
(turtle constructor)

```
t = turtle.Turtle()
t.forward(100)
```

variable that refers to
a turtle object



t

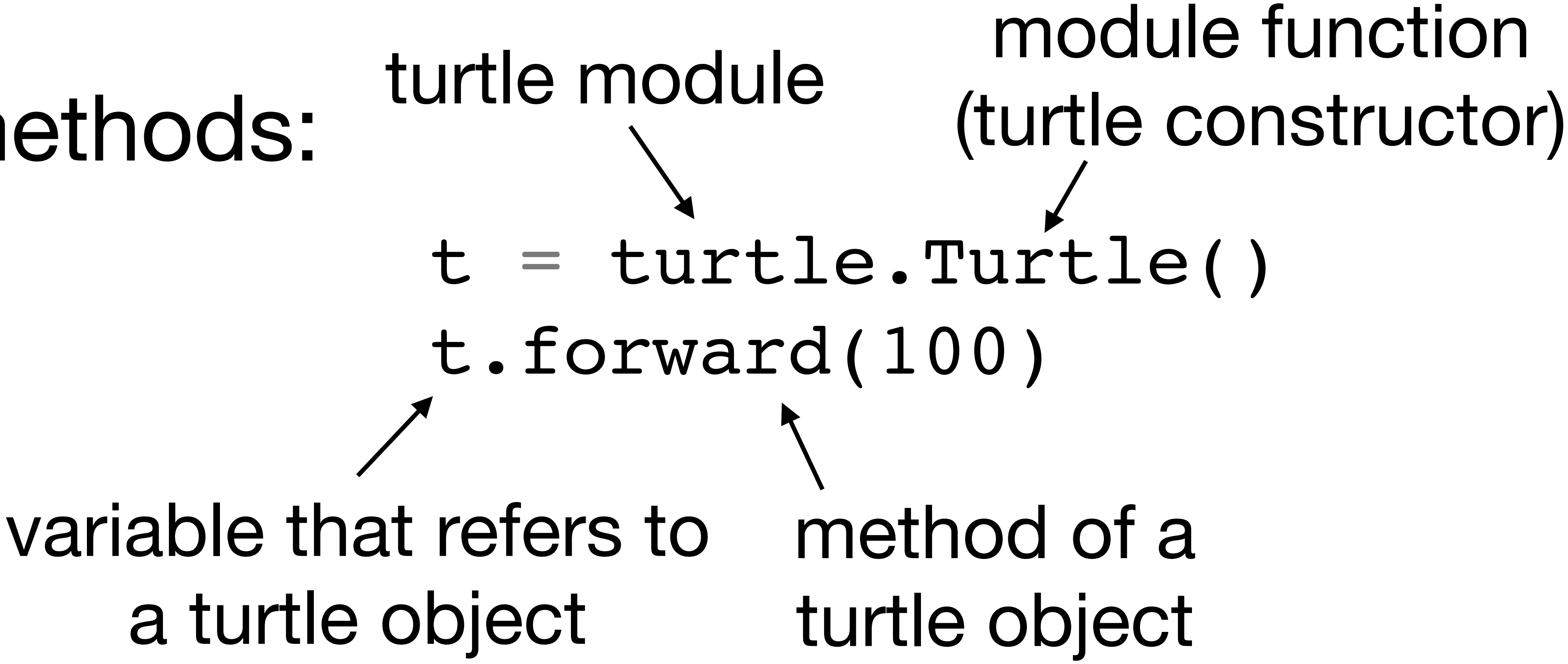
turtle

# data and methods

# Strings are objects.

We’ve seen other objects before: turtles!

Turtles had methods:



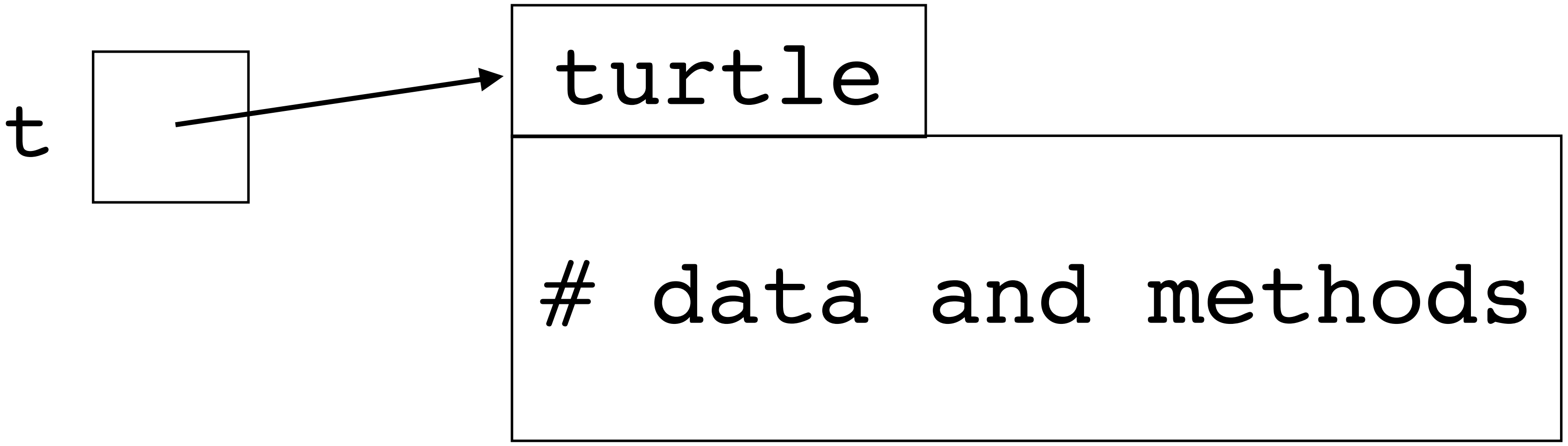
turtle module

module function (turtle constructor)

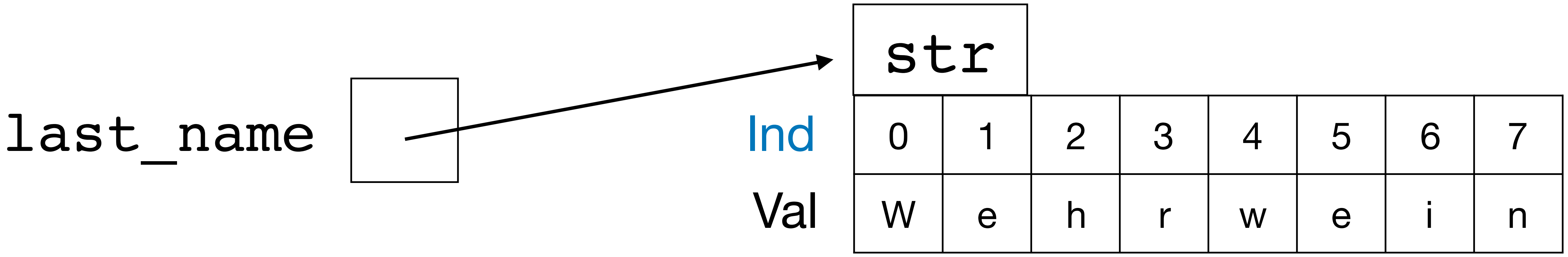
```
t = turtle.Turtle()
t.forward(100)
```

variable that refers to a turtle object

method of a turtle object



# Strings are objects.



`last_name`

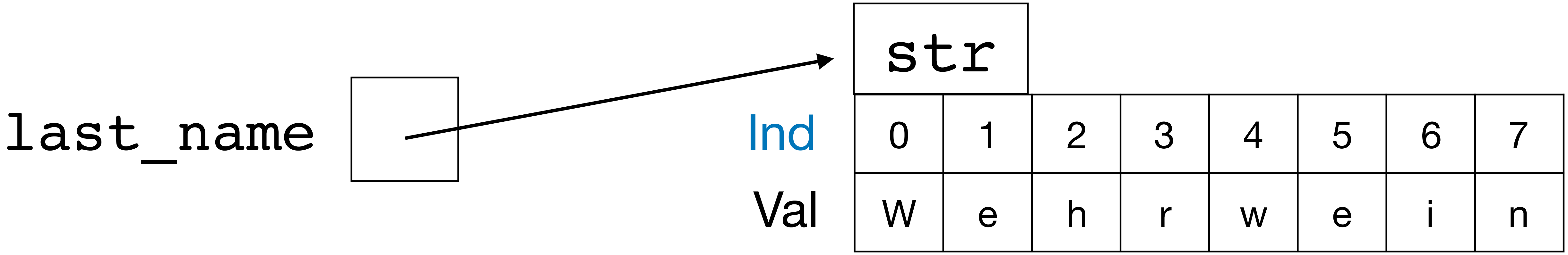
`str`

| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Strings are objects too - they also have methods.

```
last_name = "Wehrwein"
```

# Strings are objects.



`last_name`

`str`

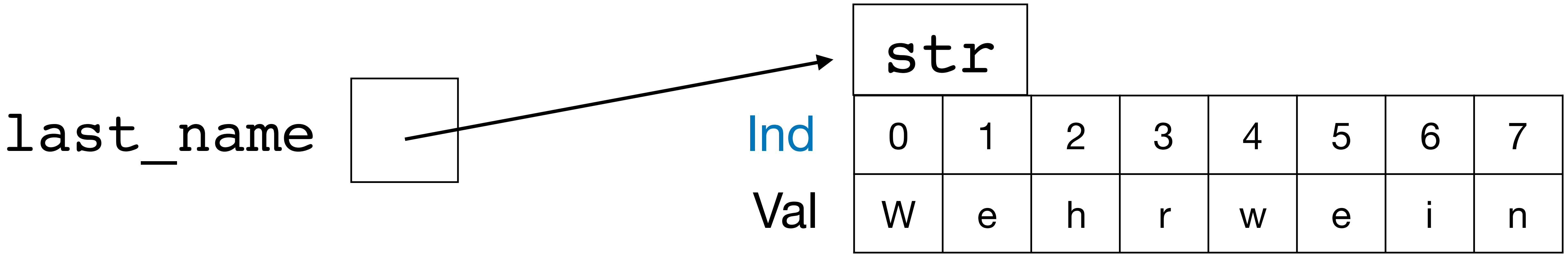
| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Strings are objects too - they also have methods.

variable that refers to
a string object

```
last_name = "Wehrwein"
```

# Strings are objects.



|  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|
| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Val | W | e | h | r | w | e | i | n |

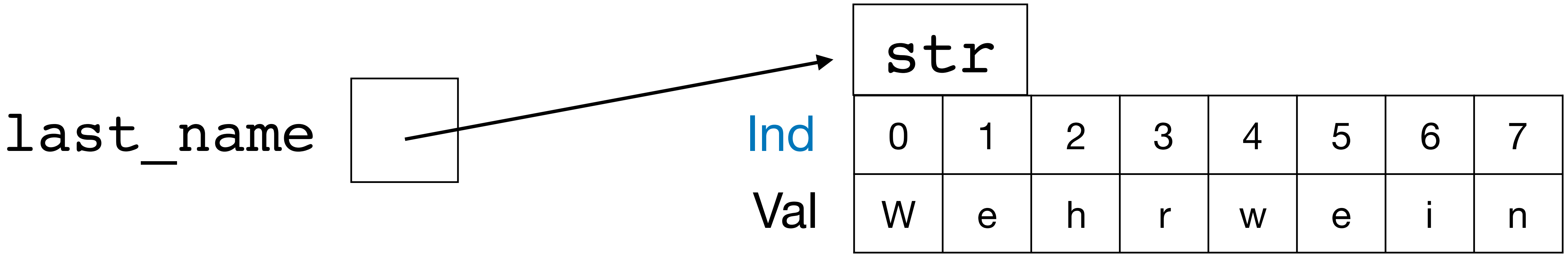
Strings are objects too - they also have methods.

variable that refers to a string object

a string literal

```
last_name = "Wehrwein"
```

# Strings are objects.



| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Strings are objects too - they also have methods.

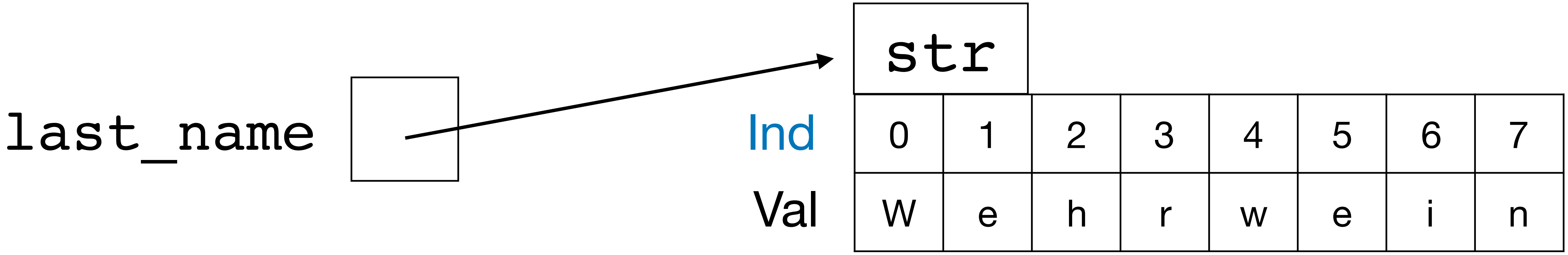
variable that refers to a string object

a string literal

```
last_name = "Wehrwein"

last_name.upper()
```

# Strings are objects.



| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Strings are objects too - they also have methods.

variable that refers to a string object

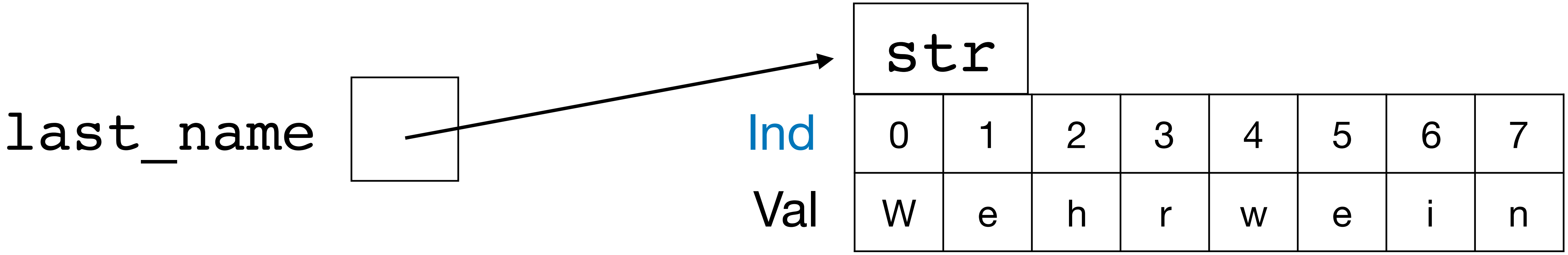
a string literal

```
last_name = "Wehrwein"

last_name.upper()
```

method of a string object

# Strings are objects.



| Ind | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| Val | W | e | h | r | w | e | i | n |

Strings are objects too - they also have methods.

variable that refers to a string object

a string literal

```
last_name = "Wehrwein"
```

```
last_name.upper()
```

method of a string object

Methods can be called directly on the literal string, too:

```
"Wehrwein".upper()
```

# Strings have many methods

here are a few of them:

| Method | Parameters | Description |
|---|---|---|
| upper | none | Returns a string in all uppercase |
| lower | none | Returns a string in all lowercase |
| strip | none | Returns a string with the leading and trailing whitespace removed |
| count | item | Returns the number of occurrences of item |
| replace | old, new | Replaces all occurrences of old substring with new |
| find | item | Returns the leftmost index where the substring item is found, or -1 if not found |

# String methods: demo

`upper, lower, count, replace, find, strip`

# String methods: demo

`upper, lower, count, replace, find, strip`

```
word = "Banana"
word.upper()
word.lower()
word.count("a")
word.replace("a", "A")

line = " snails are out "
line.find("s")
line.find("snails")
line.find("banana")
line.strip()
line.strip().upper()

word = "Bellingham"
word = word[:9] + word[9].upper()
```

# String Methods: More

The textbook (Section 9.5) has a more complete listing of string methods:
http://interactivepython.org/runestone/static/thinkcspy/Strings/StringMethods.html

The Python documentation has full details of the `str` type and all its methods:

https://docs.python.org/3/library/stdtypes.html#str

You should know how to use `upper`, `lower`, `replace`, and `find`.

# Worksheet - Exercise 2

```
phrase = "WWU is in Bellingham"
phrase = phrase[:19] + phrase[19].upper()
```

Write a function that capitalizes the last letter of **any** string:

```
def capitalize_last(in_str):
    """ Return a copy of in_str with its
        last letter capitalized.
    """
```

```
# Example:
capitalize_last(“Mix")) # => "MiX"
```

# String Methods: Evaluation

**Problem**: write an expression to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input
```

# String Methods: Evaluation

**Problem**: write an expression to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input
```

```
=> " Y  eS "
```

# String Methods

**Problem**: write an expression to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "")
```

```
=> "YeS"
```

# String Methods

**Problem**: write an expression to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "").lower()
```

```
=> "yes"
```

# String Methods

**Problem**: write an expression to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "").lower()
```

```
=> "yes"
```

**dot (method call) operators are evaluated left-to-right!**

# String Methods

**Problem**: write an *expression* to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "").lower() == "yes"
```

```
=> " Y  eS " .replace(" ", "").lower() == "yes"
=> "YeS".lower() == "yes"
=> "yes" == "yes"
```

**dot (method call) operators are evaluated left-to-right!**

# String Methods

**Problem**: write an *expression* to determine if a string `user_input` contains the word `"yes"`, with any capitalization and with any amount of spaces.

```
user_input.replace(" ", "").lower() == "yes"
```

```
=> " Y  eS " .replace(" ", "").lower() == "yes"

=> "YeS".lower() == "yes"

=> "yes" == "yes"

=> True
```

**dot (method call) operators are evaluated left-to-right!**

# Worksheet - Exercise 3

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with the first
        # symbol removed. If there is no # in
        the string, return input unchanged.
    """
```

Do this without a loop!

For reference:

| Method | Description |
|---|---|
| s.upper() | Returns s in all uppercase |
| s.lower() | Returns s in all lowercase |
| s.strip() | Returns s with the leading and trailing whitespace removed |
| s.count(t) | Returns the number of occurrences of t in s |
| s.replace(u, v) | Replaces all occurrences of substring u with v in s |
| s.find(t) | Returns the leftmost index where the substring item is found, or -1 if not found |

# Operators on Strings

**Familiar:**

| | |
|---|---|
| `+` | concatenation |
| `*` | repetition |
| `[ ]` | indexing, slicing |
| `==` | equals |
| `!=` | not equals |

# Operators on Strings

**Familiar:**

| | | |
|---|---|---|
| `+` | concatenation | `"a" + "b" => "ab"` |
| `*` | repetition | |
| `[ ]` | indexing, slicing | |
| `==` | equals | |
| `!=` | not equals | |

# Operators on Strings

**Familiar:**

| | | |
|---|---|---|
| `+` | concatenation | `"a" + "b" => "ab"` |
| `*` | repetition | `"ha" * 3 => "hahaha"` |
| `[ ]` | indexing, slicing | |
| `==` | equals | |
| `!=` | not equals | |

# Operators on Strings

**Familiar:**

| | | |
|---|---|---|
| `+` | concatenation | `"a" + "b" => "ab"` |
| `*` | repetition | `"ha" * 3 => "hahaha"` |
| `[ ]` | indexing, slicing | `"batman"[:3] => "bat"` |
| `==` | equals | |
| `!=` | not equals | |

# Operators on Strings

**Familiar:**

| | | |
|---|---|---|
| `+` | concatenation | `"a" + "b" => "ab"` |
| `*` | repetition | `"ha" * 3 => "hahaha"` |
| `[ ]` | indexing, slicing | `"batman"[:3] => "bat"` |
| `==` | equals | `"antman" == "natman" => False` |
| `!=` | not equals | |

# Operators on Strings

**Familiar:**

| | | |
|---|---|---|
| `+` | concatenation | `"a" + "b" => "ab"` |
| `*` | repetition | `"ha" * 3 => "hahaha"` |
| `[ ]` | indexing, slicing | `"batman"[:3] => "bat"` |
| `==` | equals | `"antman" == "natman" => False` |
| `!=` | not equals | `"antman" != natman" => True` |

# String operators

# String operators

**Unfamiliar, but intuitive:**

# String operators

**Unfamiliar, but intuitive:**

`in`

# String operators

**Unfamiliar, but intuitive:**

```
in    "a" in "abc".          # => True
```

# String operators

**Unfamiliar, but intuitive:**

```
in    "a" in "abc".         # => True
      "dab" in "abacadabra" # => True
```

# String operators

**Unfamiliar, but intuitive:**

`in`

```
"a" in "abc".                # => True
"dab" in "abacadabra" # => True
"A" in "abate"               # => False
```

# String operators

**Unfamiliar, but intuitive:**

```
in     "a" in "abc".        # => True
       "dab" in "abacadabra" # => True
       "A" in "abate"       # => False
       "eye" in "team"      # => False
```

# String operators

**Unfamiliar, but intuitive:**

`in`

```
"a" in "abc".             # => True
"dab" in "abacadabra"     # => True
"A" in "abate"            # => False
"eye" in "team"           # => False
```

`not in`: exactly what you think (opposite of `in`)

# String operators

much like in a dictionary

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all `True`:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```

# String operators

Familiar, but (possibly) unintuitive:

much like in a dictionary

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all `True`:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```

# String operators

Familiar, but (possibly) unintuitive:

`<, >`

much like in a dictionary

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all `True`:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```

# String operators

Familiar, but (possibly) unintuitive:

`<, >`

much like in a dictionary

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all `True`:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```

# String operators

Familiar, but (possibly) unintuitive:

`<, >`

much like in a dictionary

Inequality comparisons follow lexicographic ordering:

- Order based on the first character
- If tied, use the next character,
- and so on

These are all `True`:

```
"a" < "b"
"ab" < "ac"
"a" < "aa"
"" < "a"
```

# String operators

**Familiar, but (a little) unintuitive:**

`<, >`

**Caveat:** lexicographic ordering is case-sensitive, and ALL upper-case characters come before ALL lower-case letters:

These are all `True`:

```
"A" < "a"
"Z" < "a"
"Jello" < "hello"
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

Tie - next character

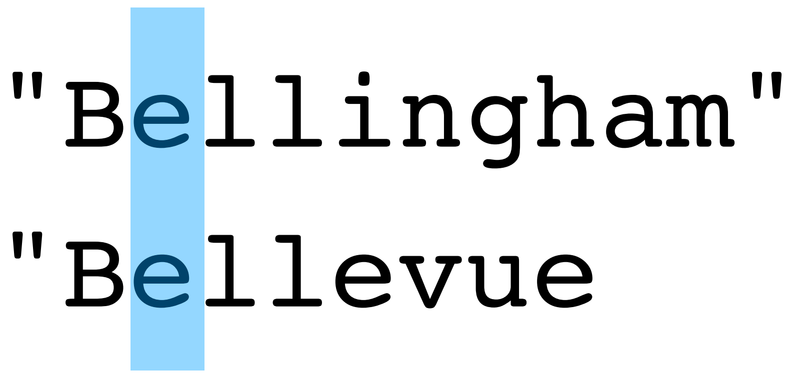
# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`



```
"Bellingham"
"Bellevue
```

Tie - next character

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

Tie - next character

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

Tie - next character

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

`i > e`, so `"Bellingham" > "Bellevue"`

# Lexicographic Ordering

Example: `"Bellingham" > "Bellevue"`

```
"Bellingham"
"Bellevue
```

`i > e`, so `"Bellingham" > "Bellevue"`

Aside:
`"Bell" < "Bellingham" => True`

When all letters are tied, the shorter word comes first.

# Lexicographic Ordering: Aside

`"?" < "!" # =>` ???

# Lexicographic Ordering: Aside

```
"?" < "!" # => ???
```

The ord function takes a character and returns its numerical (ASCII) code, which determines its ordering.

# Lexicographic Ordering: Aside

```
"?" < "!" # => ???
```

The ord function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The chr function takes a numerical (ASCII) code and returns the corresponding character.

# Lexicographic Ordering: Aside

```
"?" < "!" # => ???
```

The ord function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The chr function takes a numerical (ASCII) code and returns the corresponding character.

```
ord("?") # => 63
ord("!") # => 33
```

# Lexicographic Ordering: Aside

```
"?" < "!" # => ???
```

The ord function takes a character and returns its numerical (ASCII) code, which determines its ordering.

The chr function takes a numerical (ASCII) code and returns the corresponding character.

```
ord("?") # => 63
ord("!") # => 33
```

```
"?" < "!" # => False
```

# Lexicographic Ordering

**ABCD**: Which of the these evaluates to `True`?

A. `"bat" > "rat"`

B. `"tap" < "bear"`

C. `"Jam" < "bet"`

D. `"STEAM" > "STEP!"`