CSCI 141

Lecture 19
String Comparisons and Ordering
Introduction to Lists

Announcements

Announcements

- A4 due tonight.

Announcements

- A4 due tonight.
- I have office hours 2-3:30 today.

Announcements

- A4 due tonight.
- I have office hours 2-3:30 today.
- Remember: you have a total of 3 slip days to spend on assignments throughout the quarter.

Announcements

- A4 due tonight.
- I have office hours 2-3:30 today.
- Remember: you have a total of 3 slip days to spend on assignments throughout the quarter.
- A5 out circa Wednesday 5/22, due Friday 5/31

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.

Why are we talking about this?

From my Winter 2019 course evaluations:

Why are we talking about this?

From my Winter 2019 course evaluations:

"Genuinely appreciated the diversity talk he gave at the beginning of the first class. I was tempted to drop the class after repeated offensive remarks in mentor hours by students but I kept thinking about his talk."

Why are we talking about this?

From my Winter 2019 course evaluations:

"Genuinely appreciated the diversity talk he gave at the beginning of the first class. I was **tempted to drop the class** after repeated offensive remarks in mentor hours by students but I kept thinking about his talk."

Why are we talking about this?

From my Winter 2019 course evaluations:

"Genuinely appreciated the diversity talk he gave at the beginning of the first class. I was tempted to drop the class after repeated **offensive remarks in mentor hours** by students but I kept thinking about his talk."

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.

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

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.

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

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

What can you do:

What can you do:

straight cis middle class white male (etc.) edition

What can you do:

straight cis middle class white male (etc.) edition

- **Recognize** that this is a problem.

What can you do:

straight cis middle class white male (etc.) edition

- **Recognize** that this is a problem.
- **Listen** to people in underrepresented groups

What can you do:

straight cis middle class white male (etc.) edition

- **Recognize** that this is a problem.
- **Listen** to people in underrepresented groups
  - Understand their experiences.

What can you do:

straight cis middle class white male (etc.) edition

- **Recognize** that this is a problem.

- **Listen** to people in underrepresented groups
  - Understand their experiences.

  - If someone gives you feedback, **listen**. Resist the temptation to get **defensive.** Thank them for the feedback, and think about it.

What can you do:

straight cis middle class white male (etc.) edition

- **Recognize** that this is a problem.
- **Listen** to people in underrepresented groups
  - Understand their experiences.
  - If someone gives you feedback, **listen**. Resist the temptation to get **defensive.** Thank them for the feedback, and think about it.
- **Speak up** if you witness discrimination, harassment, or any inappropriate comments or behavior.

What can you do:

underrepresented group member edition

What can **you** do:

underrepresented group member edition

(I'm horribly underqualified to give advice on this...)

What can **you** do:

underrepresented group member edition

(I'm horribly underqualified to give advice on this...)

- **Recognize** that this is a problem.

What can **you** do:

underrepresented group member edition

(I'm horribly underqualified to give advice on this...)

- **Recognize** that this is a problem.
- Seek **mentorship**

What can **you** do:

underrepresented group member edition

(I'm horribly underqualified to give advice on this...)

- **Recognize** that this is a problem.
- Seek **mentorship**
- Find **community**

What can **you** do:

underrepresented group member edition

(I'm horribly underqualified to give advice on this...)

- **Recognize** that this is a problem.
- Seek **mentorship**
- Find **community**

A good place to start:

- WWU Association for Women in Computing (AWC)
  (not just for women!)

CS Stories: What's it like to be a female professor?

Who: Dr. Sharmin, Dr. Liu, Dr. Islam, AWC professional guests from industry, alumni, friends, YOU!

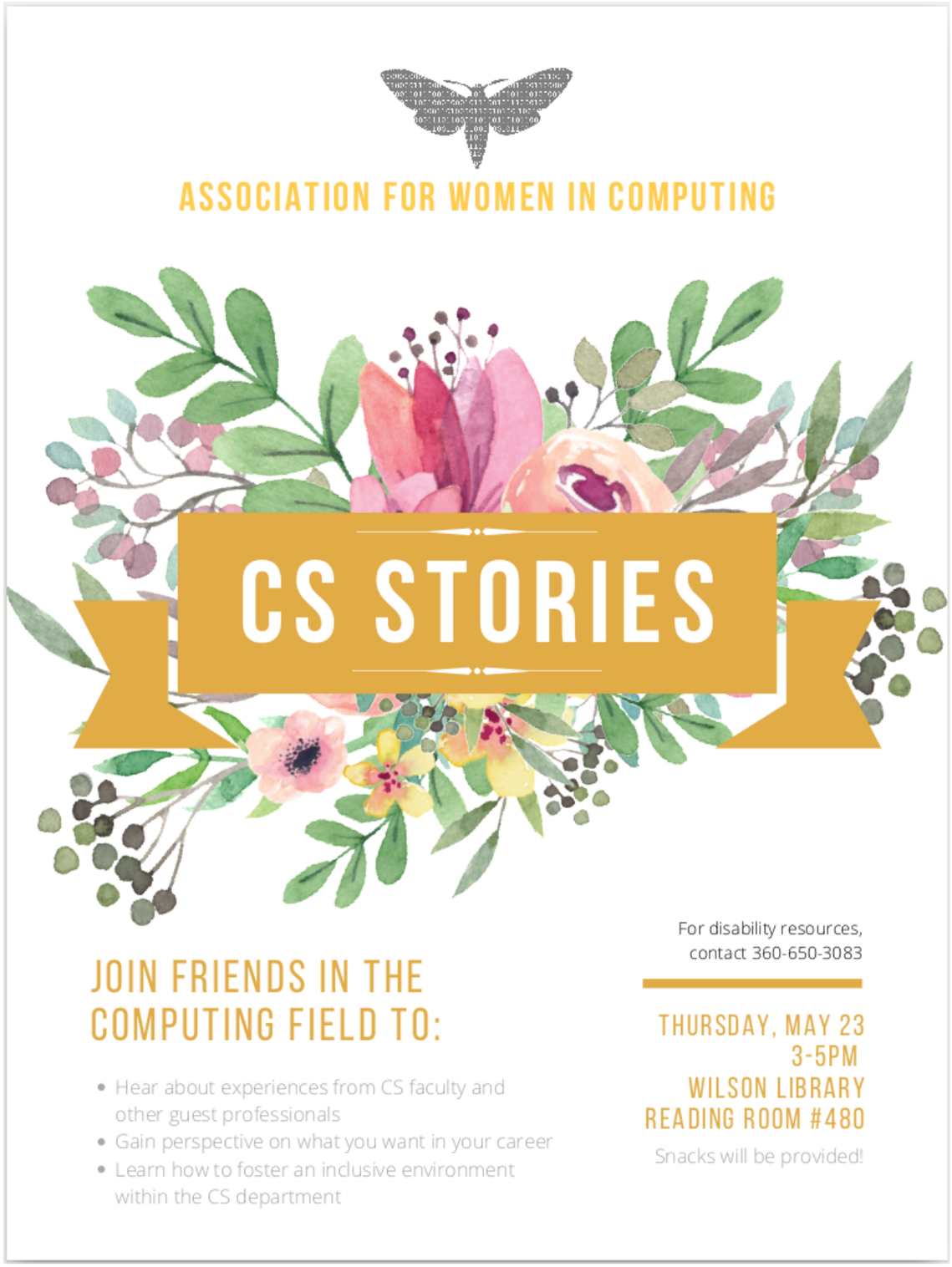
What: Creating the space to open about experiences as students in education with various career goals in addition to equipping our friends to be allies for underrepresented friends.

When: Thursday May 23rd from 3-5pm. Doors open @2:45pm

Where: Wilson Library Reading Room #480 (yes the Harry Potter Reading Room)

Contact: awc.wwu@gmail.com for more info or questions!

See you there!



Happenings

Tuesday, 5/21 – Peer Lecture Series: Math in CS
– 5 pm in CF 165

Wednesday, 5/22 – Tech Talk: OSNEXUS
– 5 pm in CF 115

Thursday, 5/23 – AWC Presents: CS Stories
– 3 – 5 pm in WL 480

Saturday and Sunday, 5/25 & 5/26 – Spring Game Jam
– 10 am in CF 105

Goals

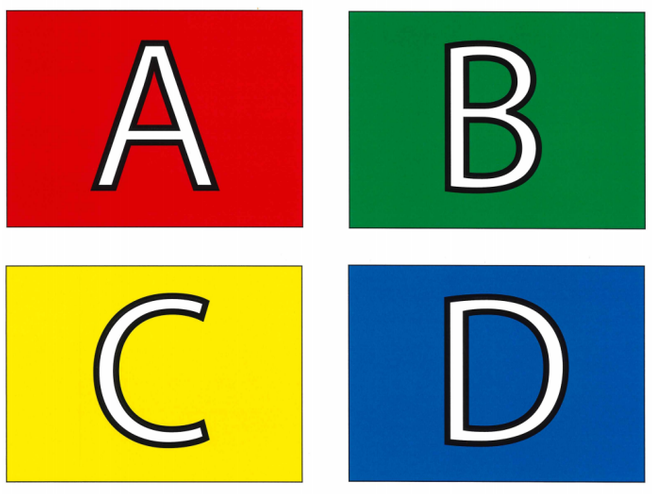
- Understand the behavior of the following operators on strings:
  - `<`, `>`, `==`, `!=`, `in`, and `not in`
    - Understand how Python orders strings using lexicographic ordering
- Know how to create, index, slice, and check for membership in lists.
- Understand the behavior of the `+`, `*`, `in`, `not in`, operators on lists.

Last time...

Know how Python interprets negative indices into strings.

Know how to use slicing to get substrings

ABCD: Which of the these does **not** evaluate to `"king"`?




```
s = "Viking"

A. s[-4:]
B. s[2:6]
C. s[1:][1:]
D. s[4:]
```

Slicing/Indexing: Out of range

Slicing/Indexing: Out of range

```
s = "four"
```

Slicing/Indexing: Out of range

```
s = "four"

s[0] # => "f"
```

Slicing/Indexing: Out of range

```
s = "four"

s[0] # => "f"

s[5] # IndexError: string index out of range
```

Slicing/Indexing: Out of range

```
s = "four"

s[0] # => "f"

s[5] # IndexError: string index out of range

s[-1] # => "r"
```

Slicing/Indexing: Out of range

```
s = "four"

s[0] # => "f"

s[5] # IndexError: string index out of range

s[-1] # => "r"

s[-5] # IndexError: string index out of range
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

```
s[3:0:-1] # => "ruo"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

```
s[3:0:-1] # => "ruo"
```

Negative step size: from start down to but not including end.

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

```
s[3:0:-1] # => "ruo"
```

Negative step size: from start down to but not including end.

```
s[::-1] # => "ruof"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

(like `range`!)

```
s[3:0:-1] # => "ruo"
```

Negative step size: from start down to but not including end.

```
s[::-1] # => "ruof"
```

Slicing/Indexing: Advanced

This will not be tested, but might be useful!

```
s = "four"

s[:2] # => "fo"

s[:7] # => "four" (!?)
```

Slice ends beyond the length are OK!

```
s[1:4:2] # => "or"
```

Slices can take a step size!

(like `range`!)

```
s[3:0:-1] # => "ruo"
```

Negative step size: from start down to but not including end.

```
s[::-1] # => "ruof"
```

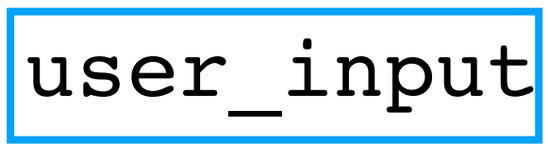
This idiom concisely reverses a string.

Last time...

- Know how to use a few of the basic methods of string objects:
  - `s.upper()` - convert s to upper case
  - `s.lower()` - convert s to lower case
  - `s.find(t)` - return the (start) index of t in s
    or -1 if it's not in s
  - `s.replace(p, q)` - replace all instances of p with q in s
- All these (except `find`) return a **new** string with the given modifications.

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.



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`

`=> " 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(" ", "")
```

```
=> " 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(" ", "")
```

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

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(" ", "")
```

```
=> " Y  eS ".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()
```

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

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()
```

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

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

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

Today’s Quiz

- 3 minutes

Today's Quiz

- 3 minutes

For reference:

- `s.upper()` - convert s to upper case
- `s.lower()` - convert s to lower case
- `s.find(t)` - return the (start) index of t in s
  or -1 if it's not in s
- `s.replace(p, q)` - replace all instances of p
  with q in string s

Today's Quiz

- 3 minutes
- Working with a neighbor: do your answers agree? (2 minutes)

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.

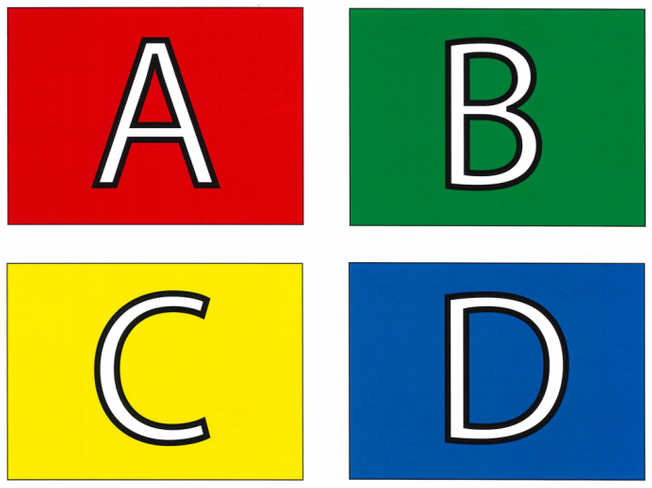
The chr function takes a numerical (ASCII) code and returns the corresponding character.

```
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!"`

Lists

A list is an object that contains a sequence of values.

We've seen them before.

Values can be of any type(s)!

Lists

A list is an object that contains a sequence of values.

We've seen them before.

```
for value in [1, 16, 4]:
    print(value)
```

Values can be of any type(s)!

Lists

A list is an object that contains a sequence of values.

We've seen them before.

```
for value in [1, 16, 4]:
    print(value)
```

Syntax:

Values can be of any type(s)!

Lists

A list is an object that contains a sequence of values.

We've seen them before.

```
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

Values can be of any type(s)!

Lists

A list is an object that contains a sequence of values.

We've seen them before.

```
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

comma-separated list of values

Values can be of any type(s)!

Lists

A list is an object that contains a sequence of values.

We've seen them before.

```
for value in [1, 16, 4]:
    print(value)
```

Syntax:

```
[val0, val1, val2, val3]
```

comma-separated list of values

surrounded by square brackets

Values can be of any type(s)!

What can we do with Lists?

A lot of this should look familiar.

These things work analogously to strings:

- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators

What can we do with Lists?

A lot of this should look familiar.

```
a_list = ["Scott", 34, 27.7]
```

These things work analogously to strings:

- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators

Demo

A lot of this should look familiar.

These things work analogously to strings:

- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators

Demo

A lot of this should look familiar.

```
a_list = ["Scott", 34, 27.7]
```

These things work analogously to strings:

- Indexing
- Slicing
- The len function
- in and not in operators
- + and * operators

Demo

A lot of this should look familiar.

make 'em

index 'em

index 'em

slice 'em

Demo

A lot of this should look familiar.

```
a_list = ["Scott", 34, 27.7]
```
make 'em

```
a_list[0]
```
index 'em

```
a_list[-1]
```
index 'em

```
a_list[1:]
```
slice 'em

Demo

A lot of this should look familiar.

Demo

A lot of this should look familiar.

```
a_list = ["Scott", 34, 27.7]

len(a_list)

len(["abc"])

len([])

34 in a_list

"34" not in a_list

a_list + ["Wehrwein", "WWU"]

["na"] * 16 + ["Batman"]
```

Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?

Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?

Tuples are also objects that hold a sequence of values of any type(s).

Lists vs Strings: What's the difference?

1. Strings hold only characters, while lists can hold values of any type(s).

...haven't we seen this before?

Tuples are also objects that hold a sequence of values of any type(s).

```
("alpaca", 14, 27.6)
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
a_list[1] = 0 # a_list is now ["a", 0, 27.6]
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
a_list[1] = 0 # a_list is now ["a", 0, 27.6]
```

Lists vs Tuples: What's the difference?

Tuples are *also* objects that hold a sequence of values of any type(s).

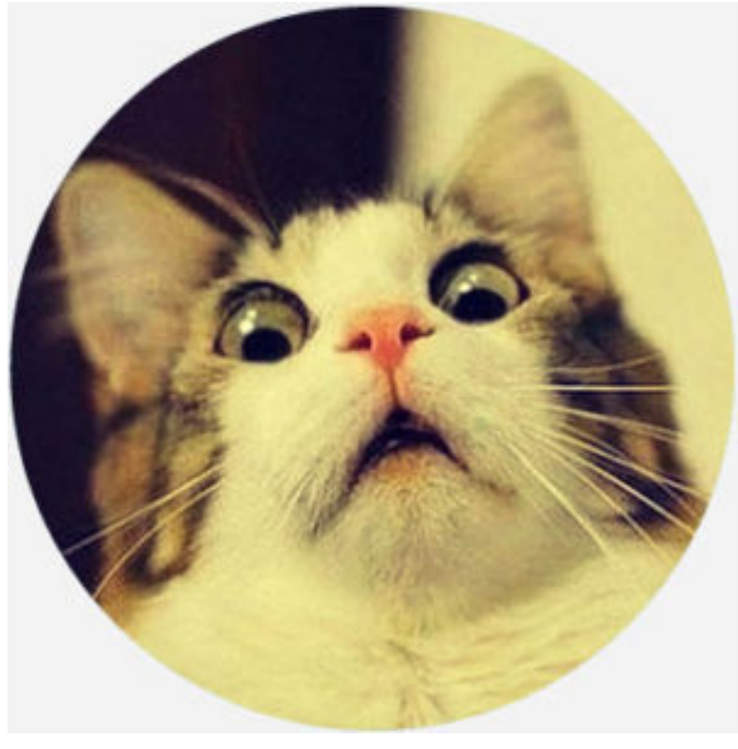
Tuples are immutable: their contents **cannot** be changed.

Lists are mutable: their contents **can** be changed.

```
a_tuple = ("a", 14, 27.6)
a_list = ["a", 14, 27.6]

a_tuple[1] # => 14
a_list[1] # => 14

a_tuple[1] = 0 # causes an error
a_list[1] = 0 # a_list is now ["a", 0, 27.6]
```



A model of how lists are stored