CS 115 Lecture 14

Strings part 2

Neil Moore

Department of Computer Science
University of Kentucky
Lexington, Kentucky 40506
neil@cs.uky.edu

3 November 2015

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.

```
if " " in name:  # if name contains a space
```

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.
    ```
    if " " in name:  # if name contains a space
    ```
  - The substring can occur anywhere: beginning, middle, or end.
    ```
    if "CS" in class:  # CS115, SCSI, 1CS
    ```

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.

    ```
    if " " in name:  # if name contains a space
    ```
  - The substring can occur anywhere: beginning, middle, or end.

    ```
    if "CS" in class:  # CS115, SCSI, 1CS
    ```
  - Case-sensitive!

    ```
    if "cs" in "CS115":  # FALSE!
    ```

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.
    ```
    if " " in name:  # if name contains a space
    ```
  - The substring can occur anywhere: beginning, middle, or end.
    ```
    if "CS" in class:  # CS115, SCSI, 1CS
    ```
  - Case-sensitive!
    ```
    if "cs" in "CS115":  # FALSE!
    ```
  - It must be contiguous:
    ```
    if "C1" in "CS115":  # FALSE!
    ```

Searching inside a string

Python has two ways to search inside a string for a substring.

- The **in** operator: `needle in haystack`
  - needle and haystack are both strings (for now).
  - Returns a boolean.
    ```
    if " " in name:  # if name contains a space
    ```
  - The substring can occur anywhere: beginning, middle, or end.
    ```
    if "CS" in class:  # CS115, SCSI, 1CS
    ```
  - Case-sensitive!
    ```
    if "cs" in "CS115":  # FALSE!
    ```
  - It must be contiguous:
    ```
    if "C1" in "CS115":  # FALSE!
    ```

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

```
pos = haystack.find(needle, 4) # start at position 4
```

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

```
pos = haystack.find(needle, 4) # start at position 4
```

    - ★ To "continue", you can use the last match + 1:

```
sp1 = haystack.find(" ") # first space
sp2 = haystack.find(" ", sp1 + 1) # next space
```

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

```
pos = haystack.find(needle, 4) # start at position 4
```

    - ★ To "continue", you can use the last match + 1:

```
sp1 = haystack.find(" ") # first space
sp2 = haystack.find(" ", sp1 + 1) # next space
```

- `rfind` is similar, but searches backwards.
  - So finds the *last* occurrence.

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

```
pos = haystack.find(needle, 4) # start at position 4
```

    - ★ To "continue", you can use the last match + 1:

```
sp1 = haystack.find(" ") # first space
sp2 = haystack.find(" ", sp1 + 1) # next space
```

- `rfind` is similar, but searches backwards.
  - So finds the *last* occurrence.

```
text = "the last space here"
lastsp = text.rfind(" ") # 14
```

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

    ```
    pos = haystack.find(needle, 4) # start at position 4
    ```

    - ★ To "continue", you can use the last match + 1:

      ```
      sp1 = haystack.find(" ") # first space
      sp2 = haystack.find(" ", sp1 + 1) # next space
      ```

- `rfind` is similar, but searches backwards.
  - So finds the *last* occurrence.

    ```
    text = "the last space here"
    lastsp = text.rfind(" ") # 14
    ```

  - To reverse-search from the middle, give both beginning and end:

    ```
    prevsp = text.rfind(" ", 0, lastsp) # 8
    ```

Searching inside a string

Sometimes you need to know now just whether the substring is there, but also *where* it is.

- The **find** method returns the location of a substring.
  `pos = haystack.find(needle)`
  - Find the first occurrence of the needle in the haystack.
  - Returns the position where it was found (0 = beginning, etc).
  - Returns -1 if it was not found.
  - Add another argument to start searching in the middle:

    ```
    pos = haystack.find(needle, 4) # start at position 4
    ```

    - ★ To "continue", you can use the last match + 1:

      ```
      sp1 = haystack.find(" ") # first space
      sp2 = haystack.find(" ", sp1 + 1) # next space
      ```

- `rfind` is similar, but searches backwards.
  - So finds the *last* occurrence.

    ```
    text = "the last space here"
    lastsp = text.rfind(" ") # 14
    ```

  - To reverse-search from the middle, give both beginning and end:

    ```
    prevsp = text.rfind(" ", 0, lastsp) # 8
    ```

Combining `find` and slicing

You can use `find` and slicing to extract part of a string:

```
space = name.find(" ")
if space != -1:
    first = name[ : space] # before the space
    last = name[space + 1 : ]  # after the space
```

Combining `find` and slicing

You can use `find` and slicing to extract part of a string:

```
space = name.find(" ")
if space != -1:
    first = name[ : space] # before the space
    last = name[space + 1 : ]  # after the space
```

Here's a loop to find all the words in a string: `words.py`

Combining `find` and slicing

You can use `find` and slicing to extract part of a string:

```
space = name.find(" ")
if space != -1:
    first = name[ : space] # before the space
    last = name[space + 1 : ]  # after the space
```

Here's a loop to find all the words in a string: `words.py`

```
text = "a string with many words"
prevspace = -1
nextspace = text.find(" ", prevspace + 1)
while nextspace != -1:
    word = text[prevspace + 1 : nextspace]
    print("word:  ", word)
    prevspace = nextspace
    nextspace = text.find(" ", prevspace + 1)
print("last word:  ", text[prevspace + 1 : ])
```

Combining `find` and slicing

You can use `find` and slicing to extract part of a string:

```
space = name.find(" ")
if space != -1:
    first = name[ : space] # before the space
    last = name[space + 1 : ]  # after the space
```

Here's a loop to find all the words in a string: `words.py`

```
text = "a string with many words"
prevspace = -1
nextspace = text.find(" ", prevspace + 1)
while nextspace != -1:
    word = text[prevspace + 1 : nextspace]
    print("word:  ", word)
    prevspace = nextspace
    nextspace = text.find(" ", prevspace + 1)
print("last word:  ", text[prevspace + 1 : ])
```

Search and replace

Often you don't really care where the substrings are, but just want to replace them with something else.

- Use the `replace` method.

```
newstr = str.replace("from", "to")
```

Search and replace

Often you don't really care where the substrings are, but just want to replace them with something else.

- Use the `replace` method.

```
newstr = str.replace("from", "to")
```

  - Finds all the occurrences of "from" and replaces them with "to".
  - Doesn't modify the original: returns a new string.

Search and replace

Often you don't really care where the substrings are, but just want to replace them with something else.

- Use the `replace` method.

```
newstr = str.replace("from", "to")
```

  - Finds all the occurrences of "from" and replaces them with "to".
  - Doesn't modify the original: returns a new string.

- You can tell replace to only replace the first few occurrences.

```
course = "CS 115 introduction to programming"
print(course.replace(" ", "-", 1)) # first occurrence
```

Search and replace

Often you don't really care where the substrings are, but just want to replace them with something else.

- Use the `replace` method.

```
newstr = str.replace("from", "to")
```

  - Finds all the occurrences of "from" and replaces them with "to".
  - Doesn't modify the original: returns a new string.

- You can tell replace to only replace the first few occurrences.

```
course = "CS 115 introduction to programming"
print(course.replace(" ", "-", 1)) # first occurrence
  → "CS-115 introduction to programming"
```

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc. . .
  - Does not affect whitespace in the middle!

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc. . .
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc. . .
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.
- ```
  userin = "␣␣\tCS␣␣115␣\n"
  clean = userin.strip() # "CS␣␣115"
  ```
  ```

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc...
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.
- `userin = "␣␣\tCS␣␣115␣\n"`
  `clean = userin.strip() # "CS␣␣115"`
- Can strip from only the left or right with `lstrip` and `rstrip`:

```
lclean = userin.lstrip() # "CS␣␣115␣\n"
rclean = userin.rstrip() # "␣␣\tCS␣␣115"
```

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc...
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.
- `userin = "␣␣\tCS␣␣115␣\n"`
  `clean = userin.strip() # "CS␣␣115"`
- Can strip from only the left or right with `lstrip` and `rstrip`:

```
lclean = userin.lstrip() # "CS␣␣115␣\n"
rclean = userin.rstrip() # "␣␣\tCS␣␣115"
print(userin) # What does this print?
```

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc. . .
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.
- `userin = "␣␣\tCS␣␣115␣\n"`
  `clean = userin.strip() # "CS␣␣115"`
- Can strip from only the left or right with `lstrip` and `rstrip`:

```
lclean = userin.lstrip() # "CS␣␣115␣\n"
rclean = userin.rstrip() # "␣␣\tCS␣␣115"
print(userin) # What does this print?
```

  - ★ Original doesn't change! `"␣␣\tCS␣␣115␣\n"`

Strip

When getting input from the user or a file, sometimes there is extra whitespace.

- The `strip` method removes whitespace from the beginning and the end of the string.
  - Whitespace: space, tab, newline, etc. . .
  - Does not affect whitespace in the middle!
  - Does *not* change the original string: returns a new one.
- `userin = "␣␣\tCS␣␣115␣\n"`
  `clean = userin.strip() # "CS␣␣115"`
- Can strip from only the left or right with `lstrip` and `rstrip`:

```
lclean = userin.lstrip() # "CS␣␣115␣\n"
rclean = userin.rstrip() # "␣␣\tCS␣␣115"
print(userin) # What does this print?
```

  - ★ Original doesn't change! `"␣␣\tCS␣␣115␣\n"`

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

- Called "iterating over" or **traversing** ("walking across") the string.

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

- Called "iterating over" or **traversing** ("walking across") the string.
- As usual `char` is the name of a new variable.

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

- Called "iterating over" or **traversing** ("walking across") the string.
- As usual `char` is the name of a new variable.
- In each iteration of the loop, `char` will be one character.
  - In order.
  - *Not* a number!

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

- Called "iterating over" or **traversing** ("walking across") the string.
- As usual `char` is the name of a new variable.
- In each iteration of the loop, `char` will be one character.
  - In order.
  - *Not* a number!
- So if `name` is `"Hal"`:
  - The first time, `char = "H"`
  - Then, `char = "a"`
  - Finally, `char = "l"`

Traversing strings

The `for` loop in Python can iterate not just over a range of integers, but also over the characters of a string:

```
for char in name:
```

- Called "iterating over" or **traversing** ("walking across") the string.
- As usual `char` is the name of a new variable.
- In each iteration of the loop, `char` will be one character.
  - In order.
  - *Not* a number!
- So if `name` is `"Hal"`:
  - The first time, `char = "H"`
  - Then, `char = "a"`
  - Finally, `char = "l"`

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?
   - Because that checks if *all* the characters are digits.
   - `hasdigit.py`

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?
   - Because that checks if *all* the characters are digits.
   - `hasdigit.py`
2. Remove the vowels from a string.
   - Remember, we can't modify the original string.

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?
   - Because that checks if *all* the characters are digits.
   - `hasdigit.py`
2. Remove the vowels from a string.
   - Remember, we can't modify the original string.
   - So we'll need to build a new string for the result.
     - ★ We'll assign to this new string to append the letters we want.

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?
   - Because that checks if *all* the characters are digits.
   - `hasdigit.py`
2. Remove the vowels from a string.
   - Remember, we can't modify the original string.
   - So we'll need to build a new string for the result.
     - ★ We'll assign to this new string to append the letters we want.
     - ★ The string will be a kind of accumulator!
   - `devowel.py`

String traversal examples

Let's write a couple of programs using strings and for loops to:

1. Check if a string contains a digit.
   - How is this different from `string.isdigit()`?
   - Because that checks if *all* the characters are digits.
   - `hasdigit.py`
2. Remove the vowels from a string.
   - Remember, we can't modify the original string.
   - So we'll need to build a new string for the result.
     - ★ We'll assign to this new string to append the letters we want.
     - ★ The string will be a kind of accumulator!
   - `devowel.py`

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.
- There are three ways to do this:
  1. Loop over the string and keep a counter.
     - ★ Initialize the counter to zero (start at the beginning).
     - ★ Increment the counter at the end of each iteration.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.
- There are three ways to do this:
  1. Loop over the string and keep a counter.
     - ★ Initialize the counter to zero (start at the beginning).
     - ★ Increment the counter at the end of each iteration.
  2. Loop over the range of indices (plural of "index"):
     - ★ `for i in range(len(name)):`
     - ★ Inside the loop, `name[i]` gives the character at that index.
     - ★ Lab 8.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.
- There are three ways to do this:
  1. Loop over the string and keep a counter.
     - ★ Initialize the counter to zero (start at the beginning).
     - ★ Increment the counter at the end of each iteration.
  2. Loop over the range of indices (plural of "index"):
     - ★ `for i in range(len(name)):`
     - ★ Inside the loop, `name[i]` gives the character at that index.
     - ★ Lab 8.
  3. Use `enumerate` to get both at the same time.
     - ★ `for i, char in enumerate(name):`
     - ★ Each iteration, `i` will be the index
     - ★ ...and `char` the character at that index.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.
- There are three ways to do this:
  1. Loop over the string and keep a counter.
     - ★ Initialize the counter to zero (start at the beginning).
     - ★ Increment the counter at the end of each iteration.
  2. Loop over the range of indices (plural of "index"):
     - ★ `for i in range(len(name)):`
     - ★ Inside the loop, `name[i]` gives the character at that index.
     - ★ Lab 8.
  3. Use `enumerate` to get both at the same time.
     - ★ `for i, char in enumerate(name):`
     - ★ Each iteration, `i` will be the index
     - ★ ...and `char` the character at that index.

Iterating with an index

Traversing a string gives you the characters, but not their positions.

- If I'm traversing "HAL 9000", the body of the loop has no way to know which "0" it's currently looking at.
- That's fine for many uses, but sometimes you do care.
- There are three ways to do this:
  1. Loop over the string and keep a counter.
     - ★ Initialize the counter to zero (start at the beginning).
     - ★ Increment the counter at the end of each iteration.
  2. Loop over the range of indices (plural of "index"):
     - ★ `for i in range(len(name)):`
     - ★ Inside the loop, `name[i]` gives the character at that index.
     - ★ Lab 8.
  3. Use `enumerate` to get both at the same time.
     - ★ `for i, char in enumerate(name):`
     - ★ Each iteration, `i` will be the index
     - ★ ...and `char` the character at that index.

Iterating with an index

Let's change our "hasdigit" function to "finddigit" in three ways.

1. `finddigit-counter.py`
2. `finddigit-range.py`
3. `finddigit-enumerate.py`

Strings to lists to strings

There are two string methods that work with lists of strings:

- `split` splits a string into words or other parts.
  - And returns a list of strings.

Strings to lists to strings

There are two string methods that work with lists of strings:

- `split` splits a string into words or other parts.
  - And returns a list of strings.
- `join` takes a list of strings and combines them.
  - And returns a single string.

Strings to lists to strings

There are two string methods that work with lists of strings:

- `split` splits a string into words or other parts.
  - And returns a list of strings.
- `join` takes a list of strings and combines them.
  - And returns a single string.

Splitting strings

The `split` method breaks a string apart and returns a list of the pieces. There are two ways to call `split`.

Splitting strings

The `split` method breaks a string apart and returns a list of the pieces. There are two ways to call `split`.

- No arguments: `name.split()`
  - Splits the string on sequences whitespace.

Splitting strings

The `split` method breaks a string apart and returns a list of the pieces. There are two ways to call `split`.

- No arguments: `name.split()`
  - Splits the string on sequences whitespace.
  - Gives you a list of "words":

```
phrase = "attention CS 115 students"
words = phrase.split()
  → [ "attention", "CS", "115", "students" ]
```

Splitting strings

The `split` method breaks a string apart and returns a list of the pieces. There are two ways to call `split`.

- No arguments: `name.split()`
  - Splits the string on sequences whitespace.
  - Gives you a list of "words":

    ```
    phrase = "attention CS 115 students"
    words = phrase.split()
      → [ "attention", "CS", "115", "students" ]
    ```

  - Multiple spaces in a row are skipped, as is leading/trailing space:

    ```
    phrase = "␣CS␣␣115-001\t"
    words = sphrase.split()
      → [ "CS", "115-001" ]
    ```

Splitting strings

The `split` method breaks a string apart and returns a list of the pieces. There are two ways to call `split`.

- No arguments: `name.split()`
  - Splits the string on sequences whitespace.
  - Gives you a list of "words":

    ```
    phrase = "attention CS 115 students"
    words = phrase.split()
      → [ "attention", "CS", "115", "students" ]
    ```

  - Multiple spaces in a row are skipped, as is leading/trailing space:

    ```
    phrase = "␣CS␣␣115-001\t"
    words = sphrase.split()
      → [ "CS", "115-001" ]
    ```

Splitting with a separator

You can also pass an arbitrary separator as an argument to `split`.

- It will break the string apart on that separator:

```
date = "04/02/2015"
parts = date.split("/")
  → [ "04", "02", "2015" ]
```

Splitting with a separator

You can also pass an arbitrary separator as an argument to `split`.

- It will break the string apart on that separator:

```
date = "04/02/2015"
parts = date.split("/")
  → [ "04", "02", "2015" ]
```

- But there are a few differences from word-splitting:
  - Multiple separators in a row aren't combined. Instead, you get an empty string in the resulting list:

```
parts = "A,,B,C".split(",")
  → [ "A", "", "B", "C" ]
```

Splitting with a separator

You can also pass an arbitrary separator as an argument to `split`.

- It will break the string apart on that separator:

```
date = "04/02/2015"
parts = date.split("/")
  → [ "04", "02", "2015" ]
```

- But there are a few differences from word-splitting:
  - Multiple separators in a row aren't combined. Instead, you get an empty string in the resulting list:

```
parts = "A,,B,C".split(",")
  → [ "A", "", "B", "C" ]
```

  - Separators at the beginning/end also give empty strings:

```
parts = ":A:2:".split(":")
  → [ "", "A", "2", "" ]
```

Splitting with a separator

You can also pass an arbitrary separator as an argument to `split`.

- It will break the string apart on that separator:

```
date = "04/02/2015"
parts = date.split("/")
  → [ "04", "02", "2015" ]
```

- But there are a few differences from word-splitting:
  - Multiple separators in a row aren't combined. Instead, you get an empty string in the resulting list:

```
parts = "A,,B,C".split(",")
  → [ "A", "", "B", "C" ]
```

  - Separators at the beginning/end also give empty strings:

```
parts = ":A:2:".split(":")
  → [ "", "A", "2", "" ]
```

Joining strings

What if we want to do the opposite of split?

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`
- Python has a built-in method to do this: `join`

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`
- Python has a built-in method to do this: `join`
  - But calling it is a little funny...

```
result = "-".join(parts)
```

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`
- Python has a built-in method to do this: `join`
  - But calling it is a little funny...

    ```
    result = "-".join(parts)
    ```
  - The *separator*, not the list, comes before the dot!
    - ★ We ask the separator to join the list of strings together.

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`
- Python has a built-in method to do this: `join`
  - But calling it is a little funny...
    ```
    result = "-".join(parts)
    ```
  - The *separator*, not the list, comes before the dot!
    - ★ We ask the separator to join the list of strings together.
  - `parts` is a sequence of strings (usually a list)

Joining strings

What if we want to do the opposite of split?

- That is, take a list of strings...
- ...and join them together with a separator.
- First, let's write the code to do this by hand:
  - `join.py`
- Python has a built-in method to do this: `join`
  - But calling it is a little funny...
    ```
    result = "-".join(parts)
    ```
  - The *separator*, not the list, comes before the dot!
    - ★ We ask the separator to join the list of strings together.
  - `parts` is a sequence of strings (usually a list)

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.
- Slots refer to arguments in order:

```
print("{}:{} {}:{}".format(userid, first, last, salted))
```

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.
- Slots refer to arguments in order:

```
print("{}:{} {}:{}".format(userid, first, last, salted))
```

- Or out of order, by index:

```
author = "{1}, {0}".format(first_name, last_name)
```

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.
- Slots refer to arguments in order:

```
print("{}:{} {}:{}".format(userid, first, last, salted))
```

- Or out of order, by index:

```
author = "{1}, {0}".format(first_name, last_name)
```

- Or with keyword arguments (like `print`'s `sep=`)

```
madlib = "The {noun} {verb}s the {noun2}".format(
    noun = "programmer", noun2 = "bug", verb = "cause"
)
```

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.
- Slots refer to arguments in order:

```
print("{}:{} {}:{}".format(userid, first, last, salted))
```

- Or out of order, by index:

```
author = "{1}, {0}".format(first_name, last_name)
```

- Or with keyword arguments (like `print`'s `sep=`)

```
madlib = "The {noun} {verb}s the {noun2}".format(
    noun = "programmer", noun2 = "bug", verb = "cause"
)
```

- Don't mix these in a single format string! Pick one.

Filling in blanks: `format`

- The `format` method builds a string by "filling in the blanks".
  - You could use concatenation, but `format` is often simpler.
- Call it on a **format string** with slots marked with braces {}
  - Usually a literal string: `"...".format(...)`
  - Returns a new string, so use in an expression.
- Slots refer to arguments in order:

```
print("{}:{} {}:{}".format(userid, first, last, salted))
```

- Or out of order, by index:

```
author = "{1}, {0}".format(first_name, last_name)
```

- Or with keyword arguments (like `print`'s `sep=`)

```
madlib = "The {noun} {verb}s the {noun2}".format(
    noun = "programmer", noun2 = "bug", verb = "cause"
)
```

- Don't mix these in a single format string! Pick one.