CS 115 Lecture 13

Strings

Neil Moore

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

29 October 2015

Strings

We've been using strings for a while. What can we do with them?

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`
- Convert (type-cast) them into ints or floats: `num = int(userin)`

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`
- Convert (type-cast) them into ints or floats: `num = int(userin)`
- Concatenate them with +: `name = first + " " + last`

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`
- Convert (type-cast) them into ints or floats: `num = int(userin)`
- Concatenate them with +: `name = first + " " + last`
- Compare with other strings: `if "A" <= name < "K":`

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`
- Convert (type-cast) them into ints or floats: `num = int(userin)`
- Concatenate them with +: `name = first + " " + last`
- Compare with other strings: `if "A" <= name < "K":`
- Check whether they are all digits: `if mystr.isdigit()`

Strings

We've been using strings for a while. What can we do with them?

- Read them from the user: `mystr = input("Name?  ")`
- Print them to the screen: `print(mystr)`
- Convert (type-cast) them into ints or floats: `num = int(userin)`
- Concatenate them with +: `name = first + " " + last`
- Compare with other strings: `if "A" <= name < "K":`
- Check whether they are all digits: `if mystr.isdigit()`

String in detail

Let's see how to do more things with strings:

- Find the length.
- Get individual characters.
- Extract ranges of characters ("slicing").
- Convert to uppercase/lowercase.
- Search for letters or substrings.
- Search and replace substrings.
- Remove whitespace.

String in detail

Let's see how to do more things with strings:

- Find the length.
- Get individual characters.
- Extract ranges of characters ("slicing").
- Convert to uppercase/lowercase.
- Search for letters or substrings.
- Search and replace substrings.
- Remove whitespace.

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.
- To get the length of a string, use the `len` function:

```
name = "HAL 9000"
numchars = len(name) # 8
```

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.
- To get the length of a string, use the `len` function:

```
name = "HAL 9000"
numchars = len(name) # 8
```

  - Argument type: string
  - Return type: integer
- What is `len("")`?

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.
- To get the length of a string, use the `len` function:

```
name = "HAL 9000"
numchars = len(name) # 8
```

  - Argument type: string
  - Return type: integer
- What is `len("")`?
  - Zero.

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.
- To get the length of a string, use the `len` function:

```
name = "HAL 9000"
numchars = len(name) # 8
```

  - Argument type: string
  - Return type: integer
- What is `len("")`?
  - Zero.
- We'll see later that `len` works with lists too.

String length

The **length** of a string is the number of characters in it.

- Spaces count!
- So do newlines and other special characters.
- To get the length of a string, use the `len` function:

```
name = "HAL 9000"
numchars = len(name) # 8
```

  - Argument type: string
  - Return type: integer
- What is `len("")`?
  - Zero.
- We'll see later that `len` works with lists too.

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

```
first = name[0] # "H"
```

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

```
first = name[0] # "H"
```

  - This is called *subscripting* or *indexing*.

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

```
first = name[0] # "H"
```

  - This is called *subscripting* or *indexing*.
  - The position must be *smaller than* the length:

```
print(name[8]) # ERROR: out of range
```

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

```
first = name[0] # "H"
```

  - This is called *subscripting* or *indexing*.
  - The position must be *smaller than* the length:

```
print(name[8]) # ERROR: out of range
```

- You can subscript with negative numbers, to count from the end.
  - `name[-1]` is the last character (rightmost).
  - `name[-2]` is the next-to-last character.
  - ...
  - `name[-len(name)]` is the first character.

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

  ```
  first = name[0] # "H"
  ```

  - This is called *subscripting* or *indexing*.
  - The position must be *smaller than* the length:

    ```
    print(name[8]) # ERROR: out of range
    ```

- You can subscript with negative numbers, to count from the end.
  - `name[-1]` is the last character (rightmost).
  - `name[-2]` is the next-to-last character.
  - ...
  - `name[-len(name)]` is the first character.
- `name[-i]` is like `name[len(name) - i]`
  - `name[-9]` would still be out of range!

Extracting characters

The characters in a string are numbered from 0 to *length* − 1

```
HAL 9000 (length = 8)
01234567
```

- This number is called the **position** or **index** of the character.
- You can use square brackets to get the character at a given position.

  ```
  first = name[0] # "H"
  ```

  - This is called *subscripting* or *indexing*.
  - The position must be *smaller than* the length:

    ```
    print(name[8]) # ERROR: out of range
    ```

- You can subscript with negative numbers, to count from the end.
  - `name[-1]` is the last character (rightmost).
  - `name[-2]` is the next-to-last character.
  - ...
  - `name[-len(name)]` is the first character.
- `name[-i]` is like `name[len(name) - i]`
  - `name[-9]` would still be out of range!

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`
  - Can leave out either part:
    - ★ Leave out start: start at the beginning.

      `first = name[:3] # "HAL"`

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`
  - Can leave out either part:
    - ★ Leave out start: start at the beginning.
      `first = name[:3] # "HAL"`
    - ★ Leave out stop: go until the end.
      `last = name[4:]  # "9000"`

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`
  - Can leave out either part:
    - ★ Leave out start: start at the beginning.
      `first = name[:3] # "HAL"`
    - ★ Leave out stop: go until the end.
      `last = name[4:]  # "9000"`
    - ★ The whole thing (kind of silly here):
      `copy = name[:]  # "HAL 9000"`

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`
  - Can leave out either part:
    - ★ Leave out start: start at the beginning.
      `first = name[:3] # "HAL"`
    - ★ Leave out stop: go until the end.
      `last = name[4:]  # "9000"`
    - ★ The whole thing (kind of silly here):
      `copy = name[:]  # "HAL 9000"`
- Doesn't change the original string! (Makes a new one).

Extracting whole substrings: **slicing**

The square-bracket notation also lets us extract multiple characters.

```
HAL 9000 (length = 8)
01234567
```

- For example, "The first 3 characters" or "Characters 2 through 4".
- Subscript using a **slice** ("slicing").
  - Syntax: start, a colon ":", and stop (one-past-the-end).
    - ★ Similar semantics to `range(start, stop)`.
  - The first three characters: `name[0:3] # "HAL"`
    - ★ "Start at character 0, stop before character 3."
  - Two through four: `name[2:5] # "L 9"`
  - Can leave out either part:
    - ★ Leave out start: start at the beginning.
      `first = name[:3] # "HAL"`
    - ★ Leave out stop: go until the end.
      `last = name[4:]  # "9000"`
    - ★ The whole thing (kind of silly here):
      `copy = name[:]  # "HAL 9000"`
- Doesn't change the original string! (Makes a new one).

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"
- First letter uppercase:
  `almost = name.capitalize()` – "Albert einstein"

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"
- First letter uppercase:
  `almost = name.capitalize()` – "Albert einstein"
- First letter of each word uppercase:
  `good = name.title()` – Gives "Albert Einstein"

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"
- First letter uppercase:
  `almost = name.capitalize()` – "Albert einstein"
- First letter of each word uppercase:
  `good = name.title()` – Gives "Albert Einstein"
- One use: case-insensitive comparison.
  - For example, a yes-no prompt:
  - The user might type "Y" or "y" or "N" or "n".

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"
- First letter uppercase:
  `almost = name.capitalize()` – "Albert einstein"
- First letter of each word uppercase:
  `good = name.title()` – Gives "Albert Einstein"
- One use: case-insensitive comparison.
  - For example, a yes-no prompt:
  - The user might type "Y" or "y" or "N" or "n".
  - Convert the input to uppercase and compare that!

```
if userin.upper() == "Y":  # handles "y" too
```

Converting case

Python strings have several methods to change their case (capitalization).

- These methods don't change the original string, either.
  - They return a new string, so use them with assignment.
- All lowercase:
  `lazy = name.lower()` – "albert einstein"
- All uppercase:
  `telegraph = name.upper()` – "ALBERT EINSTEIN"
- First letter uppercase:
  `almost = name.capitalize()` – "Albert einstein"
- First letter of each word uppercase:
  `good = name.title()` – Gives "Albert Einstein"
- One use: case-insensitive comparison.
  - For example, a yes-no prompt:
  - The user might type "Y" or "y" or "N" or "n".
  - Convert the input to uppercase and compare that!

```
if userin.upper() == "Y":  # handles "y" too
```

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
```

    - ★ In a loop 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
```

    - ★ In a loop 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
```

    - ★ In a loop 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
    ```

    - ★ In a loop 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 the 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
    ```

    - ★ In a loop 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 the 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:

```
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 : ])
```

`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:

```
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 : ])
```

`words.py`

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.

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!

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.

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.
  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.
  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.
  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`