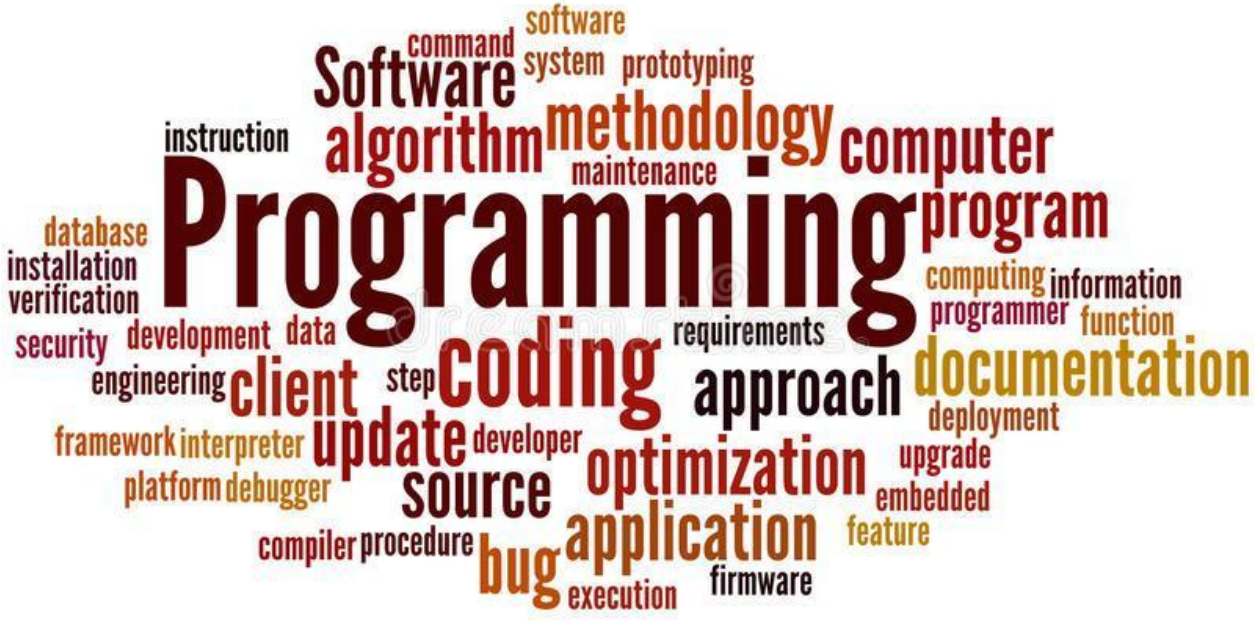
COMP 125 Programming with Python Strings



Mehmet Sayar

Koç University

So Far

- Karel
- Variables
- Conditionals
- Loops
- Functions
- Math and Random modules (will use them again from time to time)

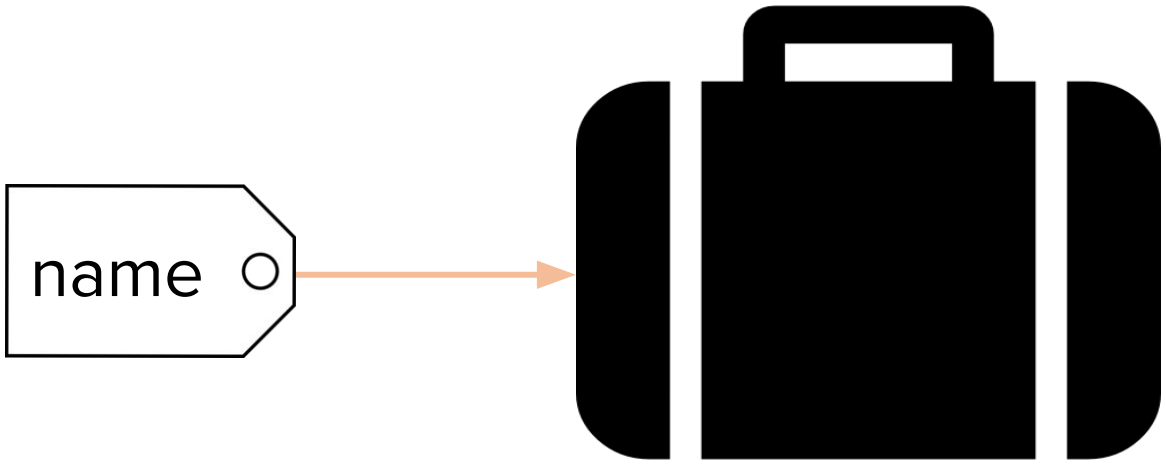
Short Detour: Data Types

- We have seen
  - Booleans
  - Integers
  - Floats
- Sort of talked about strings here and there
- Also saw that functions are objects: Type “function”

- There are many other data types

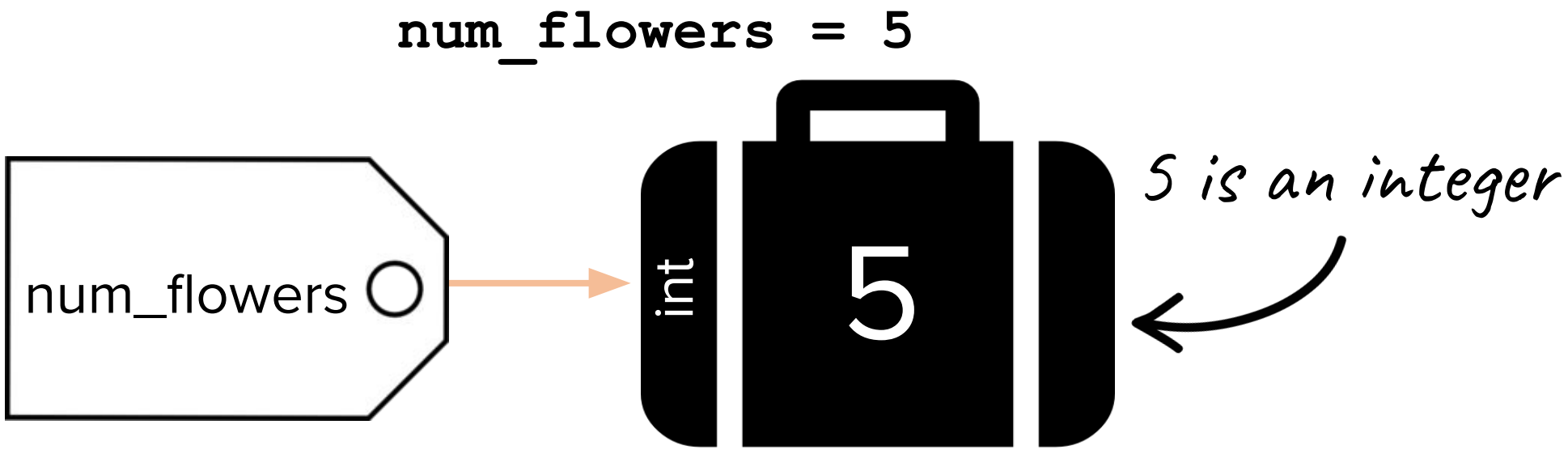
Remember: The Suitcase Analogy

- When you store information in Python, it becomes a Python object
  - Objects come in different sizes and **types**
- You can think about a Python object as a suitcase stored in your computer's memory.
- A variable is a luggage tag for your suitcase that gives it a name!



Remember: The Suitcase Analogy

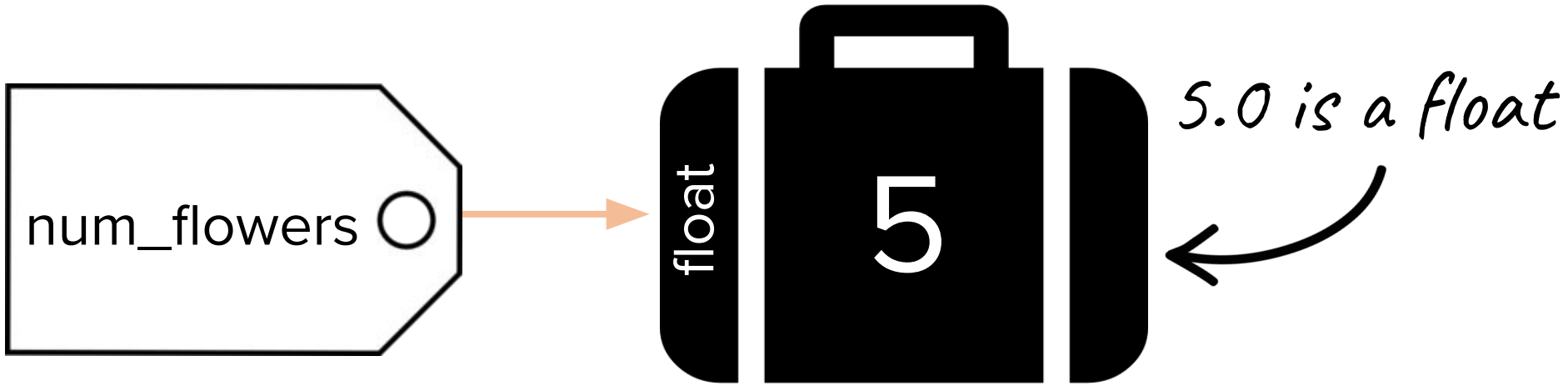
- When you store information in Python, it becomes a Python object
  - Objects come in different sizes and **types**



Remember: The Suitcase Analogy

- When you store information in Python, it becomes a Python object
  - Objects come in different sizes and **types**

```
num_flowers = 5.0
```



All Python Objects Have a Type

- Python automatically figures out the type based on the value
  - Variables are "**dynamically-typed**": you don't specify the type of the Python object they point to

- We will learn more about types as we move on, but let's introduce another built-in Python function: `type`

- This returns the type of a python object

Type Examples

```
def add(a,b):
    return a + b

print(type(add))
```

Output:

```
<class 'function'>
```

```
print(type(5))
print(type(5.0))
print(type(5 > 0))
```

Output:

```
<class 'int'>
<class 'float'>
<class 'bool'>
```

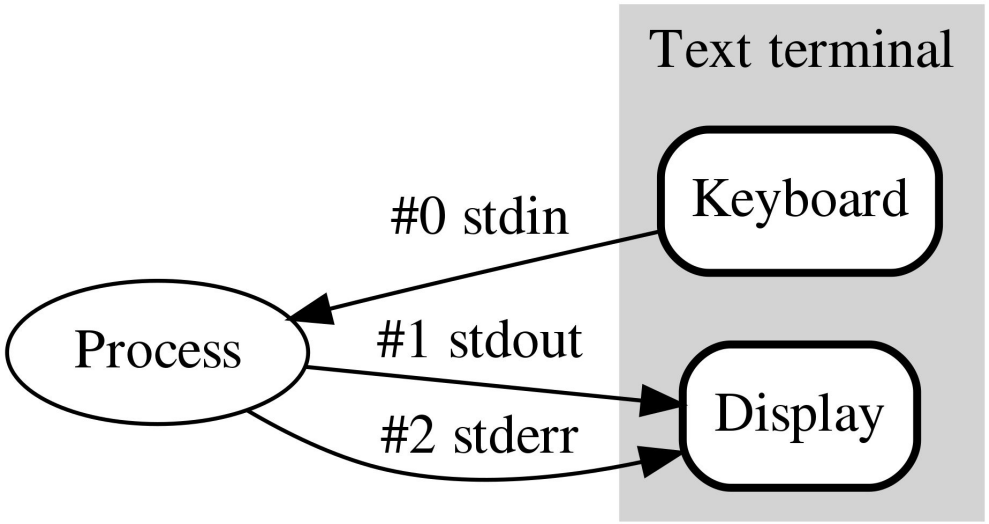
Console

- Older days of computing:
  - Computers were large and thus were housed away from the users
  - Display technology was primitive
- Result: Text terminals or text consoles
  - Text input-output
  - Hacker scenes in movies where there are no mice

- A very useful paradigm to this day
- Homework: Read up on text terminals and command line interfaces
  - An example is the “lower right-hand corner of Spyder”
  - Where we see the print output and give the keyboard input

Console Programs

- Programs designed to be used with text input and output through a console (text terminal)
- We have seen two basic functions for this:
  - `print`
  - `input`



- Anybody need a short demo?

Note that console programs can use other input-output channels as well

Text Data - Characters

- A character is a minimal unit of text
  - Character Data Type: A single character
- Letters of the alphabet, numbers, punctuation, white spaces etc.
- Multiple encodings
  - Bits to characters, e.g. 7-bit ASCII encoding
- In Python there is no separate character data type!

Text Data - Strings

- Sequence of characters
- Use single (') or double quotes (") to define them
  - Any character can be enclosed within two of these

```
my_name = 'Mehmet'
class_name = 'comp125'
secure_password = 'fg^#kwro!@-lm>'
sentence = "I don't want that."
```

Note: Use triple quotes for multi line strings. E.g.

```
"""
This is a multi-line
string
"""
```

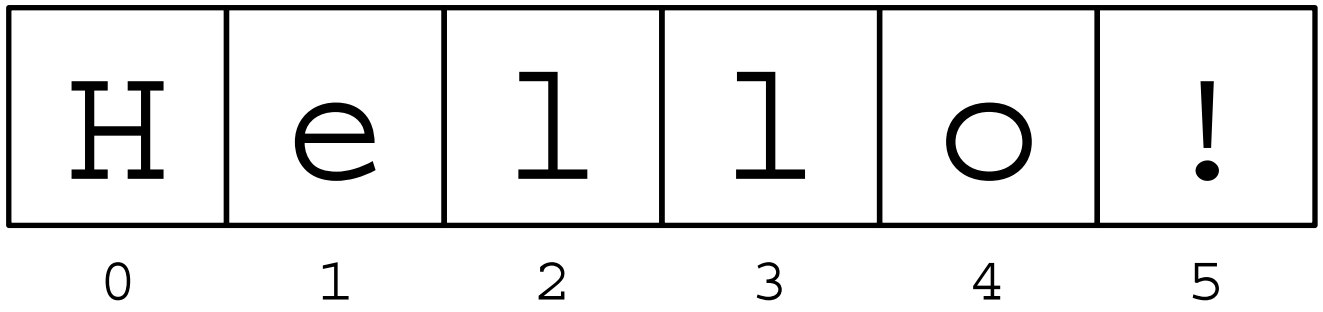
String Fundamentals

- String literals are any string of characters enclosed in single (`' '`) or double quotes (`" "`)
- Each character in the string is associated with an **`index`**
  - **`index`**: An integer representing the location of a character in a string

```
H e l l o !
```

String Fundamentals

- String literals are any string of characters enclosed in single (' ') or double quotes (" ")
- Each character in the string is associated with an **index**
  - **index**: An integer representing the location of a character in a string



String Fundamentals

- String literals are any string of characters enclosed in single (' ') or double quotes (" ")
- Each character in the string is associated with an **index**
  - **index**: An integer representing the location of a character in a string
  - You can access a character in the sequence via its index using **bracket ([ ]) notation**

```
text = 'hello’
a = text[1] #a = ‘e’
```

String Fundamentals

- String literals are any string of characters enclosed in single (' ') or double quotes (" ")
- Each character in the string is associated with an **index**
- Strings can be combined with the **+** operator in a process called **concatenation**

```
intro = 'Hello, '
name = 'Nick'
greeting = intro + name → 'Hello, Nick'
```

String Fundamentals

- String literals are any string of characters enclosed in single (' ') or double quotes (" ")
- Each character in the string is associated with an **index**
- Strings can be combined with the **+** operator in a process called **concatenation**
- Strings are **immutable**
  - They cannot be modified after they are created
  - To change a string, you must first build a new string somehow and then re-assign the string variable
  - Important consequence: If you pass a string into a function, that function cannot modify the string
  - Indexing with immutable sequences can only be used for accessing (or vice versa, indexing can be used for assignment with mutable sequences).

```
a = string[index] #This works!

string[index] = new_character #This gives an exception!
```

  - Concatenation doesn't change the existing string, but rather creates a new string and assigns the new string to the previously used variable

Concatenation Example

Figure 8-4 The string 'Carmen' assigned to `name`

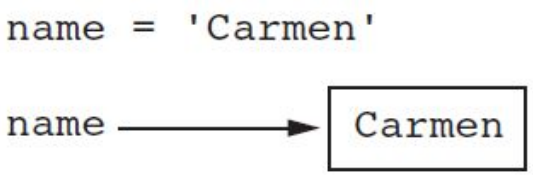
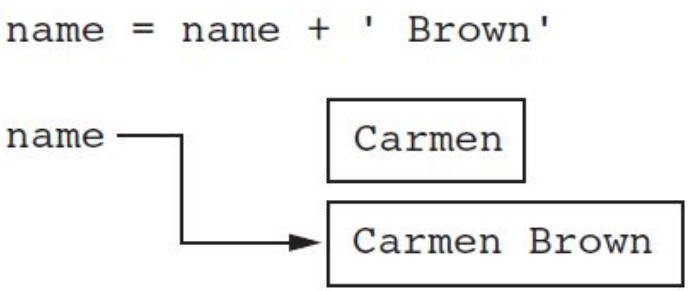


Figure 8-5 The string 'Carmen Brown' assigned to `name`



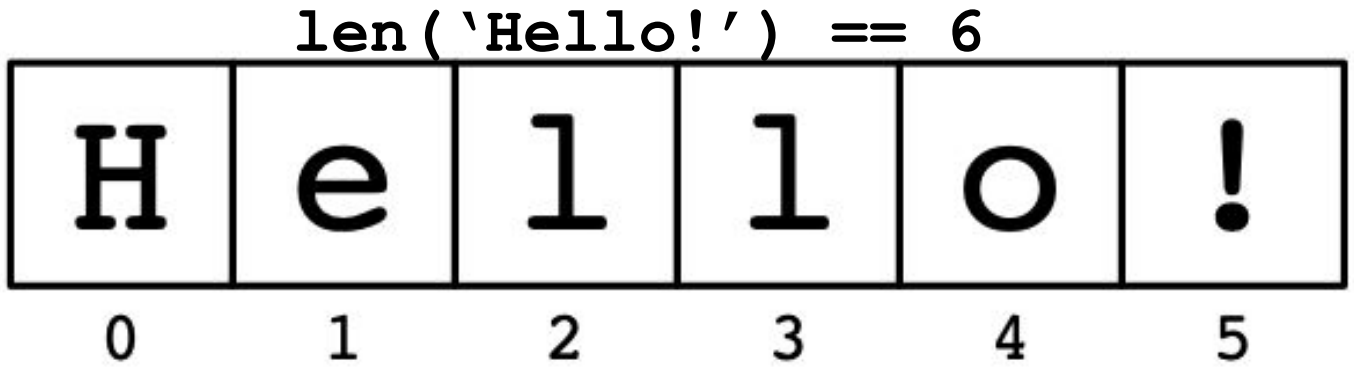
Length, Indexing and Slicing

- Length:
  - The **length** of a string is the number of characters it contains
  - We can use the Python function `len()` to evaluate the length of a string

```
len('banana') → 6
len('') → 0
len('COMP125 rocks my socks') → 22
```

Length, Indexing and Slicing

- Length:
- Zero-based indexing
  - The **first character** of a string exists at **index 0**
  - The **last character** of a string exists at **index len(s)-1**
  - Index `len(s)` is NOT a valid index.
  - Note: This is a very common paradigm in computer science!



Length, Indexing and Slicing

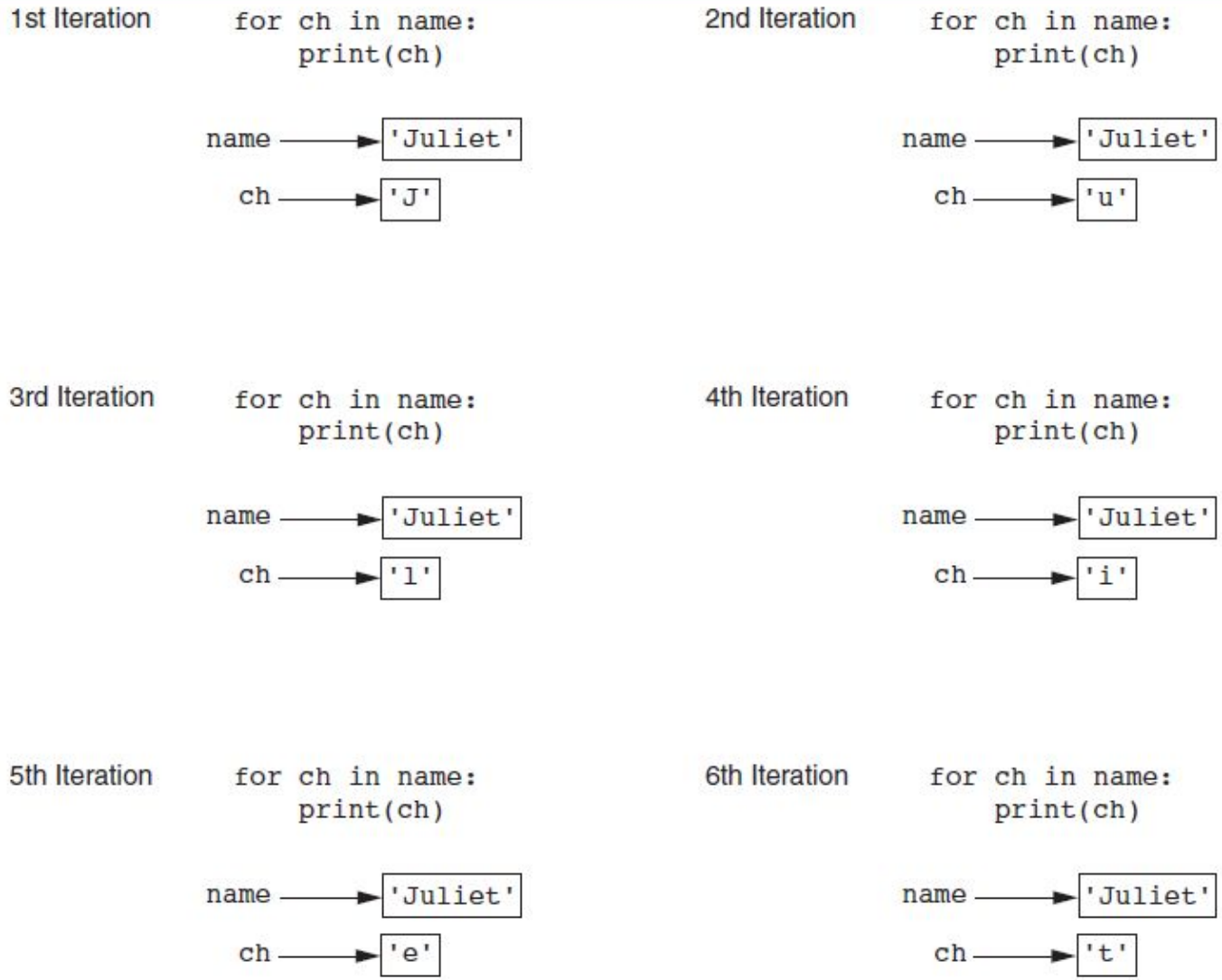
- Length:
- Zero-based indexing
- Recall the bracket ([ ]) notation: The character at index `i` of string `s` can be accessed with the expression `s[i]`
- Slices and substrings
  - A slice (or substring) of a string is a consecutive block of characters that has been extracted from the original string
  - Slicing uses bracket notation with (potentially) more than one number and colon → `s[a:b]`
  - The resulting substring consists of all characters starting from the the first index and going up to, **but not including**, the second index

Accessing Individual Characters In a String

- Use a for each loop
  - Format: `for character in string:`
  - Useful when need to iterate over the whole string, such as to count the occurrences of a specific character

- Use indexing
  - Each character has an index specifying its position in the string, starting at 0
  - Format: `character = my_string[i]`

Figure 8-1 Iterating over the string `'Juliet'`



Accessing Individual Characters In a String

- `IndexError` exception will occur if:
  - You try to use an index that is out of range for the string
  - Likely to happen when loop iterates beyond the end of the string

- `len(string)` function can be used to obtain the length of a string
  - Useful to prevent loops from iterating beyond the end of a string

String Slicing

- Slice: span of items taken from a sequence, known as a *substring*
- Slicing format: `string[start:end]`
  - Expression will return a string containing a copy of the characters from `start` up to, but not including, `end`
  - If start not specified, 0 is used for start index (`string[:end]`)
  - If end not specified, len(string) is used for end index (`string[start:]`)
- Slicing expressions can include a step value and negative indexes relative to end of string - sound familiar?
  - Similar rules as the range function!

Indexing

```
  0 1 2 3 4 5 6
s='Arthur'
```

Slides courtesy of Sam Redmond, CS41

Indexing

Slides courtesy of Sam Redmond, CS41

Indexing


```
s[0] == 'A'
s[1] == 'r'
s[4] == 'u'
s[6] # Bad!
```

Slides courtesy of Sam Redmond, CS41

Slicing

Slides courtesy of Sam Redmond, CS41

Slicing

Slides courtesy of Sam Redmond, CS41

Slicing


```
s[0:2] ==  'Ar'
```

Slides courtesy of Sam Redmond, CS41

Slicing


```
s[0:2] ==  'Ar'
```

Slides courtesy of Sam Redmond, CS41

Slicing


```
s[0:2] ==  'Ar'
s[3:6] == 'hur'
```

Slides courtesy of Sam Redmond, CS41

Slicing


```
s[0:2] ==  'Ar'
s[3:6] == 'hur'
s[1:4] == 'rth'
```

Slides courtesy of Sam Redmond, CS41

Strings

Implicitly starts at 0

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Implicitly ends at the end

Slides courtesy of Sam Redmond, CS41

Strings

Implicitly starts at 0

Implicitly ends at the end

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Slides courtesy of Sam Redmond, CS41

Strings

Implicitly starts at 0

```
s[:2] ==  'Ar'
s[3:] == 'hur'
```

Implicitly ends at the end

Slides courtesy of Sam Redmond, CS41

Basic String Methods

- `str.isupper()`, `str.islower()`
  - Return True if all the characters in str are either uppercase or lowercase, False otherwise
- `str.isalpha()`, `str.isdigit()`
  - Return True if all the characters in str are either letters ('a'-'z', 'A'-'Z') or digits ('0'-'9'), False otherwise
- `str.upper()`, `str.lower()`
  - Returns str with all letters converted to uppercase or lowercase respectively
  - The original string remains unchanged
- Remember Python strings are immutable

Basic String Method Remarks

- Some methods return a copy of the string, to which modifications have been made
  - Simulate strings as mutable objects

- String comparisons are case-sensitive
  - Uppercase characters are distinguished from lowercase characters
  - lower and upper methods can be used for making case-insensitive string comparisons

Basic String Method Remarks

- Some methods return a copy of the string, to which modifications have been made
  - Simulate strings as mutable objects

- String comparisons are case-sensitive
  - Uppercase characters are distinguished from lowercase characters
  - lower and upper methods can be used for making case-insensitive string comparisons

Basic String Method List (for Reference)

Table 8-1 Some string testing methods

Method	Description
`isalnum()`	Returns true if the string contains only alphabetic letters or digits and is at least one character in length. Returns false otherwise.
`isalpha()`	Returns true if the string contains only alphabetic letters and is at least one character in length. Returns false otherwise.
`isdigit()`	Returns true if the string contains only numeric digits and is at least one character in length. Returns false otherwise.
`islower()`	Returns true if all of the alphabetic letters in the string are lowercase, and the string contains at least one alphabetic letter. Returns false otherwise.
`isspace()`	Returns true if the string contains only whitespace characters and is at least one character in length. Returns false otherwise. (Whitespace characters are spaces, newlines (`\n`), and tabs (`\t`).
`isupper()`	Returns true if all of the alphabetic letters in the string are uppercase, and the string contains at least one alphabetic letter. Returns false otherwise.

Basic String Method List (for Reference)

Table 8-2 String Modification Methods

Method	Description
`lower()`	Returns a copy of the string with all alphabetic letters converted to lowercase. Any character that is already lowercase, or is not an alphabetic letter, is unchanged.
`lstrip()`	Returns a copy of the string with all leading whitespace characters removed. Leading whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the beginning of the string.
`lstrip(char)`	The *`char`* argument is a string containing a character. Returns a copy of the string with all instances of *`char`* that appear at the beginning of the string removed.
`rstrip()`	Returns a copy of the string with all trailing whitespace characters removed. Trailing whitespace characters are spaces, newlines (`\n`), and tabs (`\t`) that appear at the end of the string.
`rstrip(char)`	The *`char`* argument is a string containing a character. The method returns a copy of the string with all instances of *`char`* that appear at the end of the string removed.
`strip()`	Returns a copy of the string with all leading and trailing whitespace characters removed.
`strip(char)`	Returns a copy of the string with all instances of *`char`* that appear at the beginning and the end of the string removed.
`upper()`	Returns a copy of the string with all alphabetic letters converted to uppercase. Any character that is already uppercase, or is not an alphabetic letter, is unchanged.

Basic String Method List (for Reference)

Table 8-3 Search and replace methods

Method	Description
`endswith(substring)`	The *substring* argument is a string. The method returns true if the string ends with *substring*.
`find(substring)`	The *substring* argument is a string. The method returns the lowest index in the string where *substring* is found. If *substring* is not found, the method returns –1.
`replace(old, new)`	The *old* and *new* arguments are both strings. The method returns a copy of the string with all instances of *old* replaced by *new*.
`startswith(substring)`	The *substring* argument is a string. The method returns true if the string starts with *substring*.

Recall Types

- What does the following Boolean expression evaluate to?

'123' == 123

- **False**!
  - Left one is a string (`str`) and right one is an integer (`int`) -> different types

- What if we want to really compare them as if they were both numbers?
  - Type conversion!

- In order to convert between the data types we learned so far, we can use built-in Python functions: `str()`, `int()`, `float()`

Type Conversions

- In order to convert between the data types we learned so far, we can use built-in Python functions: `str()`, `int()`, `float()`

```
int('123') == 123
float('24.7') == 24.7
str(12345) == '12345'
str(20.19) == '20.19’
```

- These all evaluate to **True**!