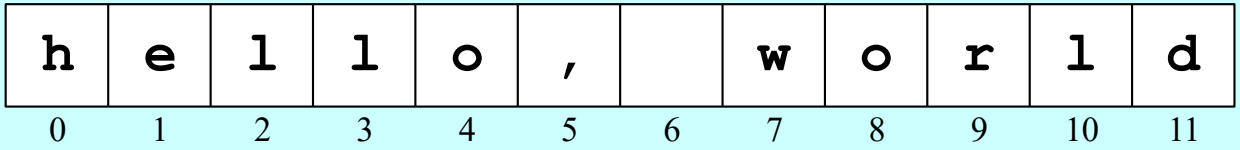
Selecting Characters from a String

- A string is (still) an ordered collection of characters. The character positions in a Python string are, as in most computer languages, identified by an ***index*** beginning at 0.
- For example, if `s` is initialized as

```
s = "hello, world"
```

  the characters in `s` are arranged like this:



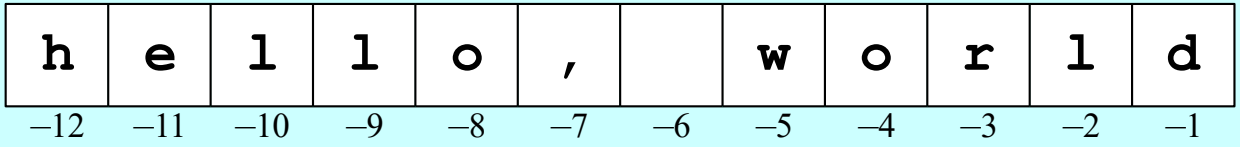
- You can select an individual character using the syntax *str*`[`*k*`]`, where *k* is the index of the desired character. The expression

```
s[7]
```

  returns the one-character string `"w"` that appears at index 7.

Negative Indexing

- Unlike JavaScript, Python allows you to specify a character position in a string by using negative index numbers, which count backwards from the end of the string. The characters in the `"hello, world"` string on the previous slide can therefore be numbered using the following indices:



- You can select the `"w"` toward the end of this string using the expression

```
s[-5]
```

  which is shorthand for the positive indexing expression

```
s[len(s) - 5]
```

Concatenation

- One of the more familiar operations available to Python strings is ***concatenation***, which consists of combining two strings end to end with no intervening characters.
- Concatenation is built into Python in the form of the **+** operator. This is consistent with how JavaScript and most other languages support concatenation.
- Noteworthy difference between Python and JavaScript: Python interprets the **+** operator as concatenation only if **both** operands are strings. If one of the operands is something other than a string, then string concatenation isn't applied. Restated, Python doesn't automatically convert numbers to strings as JavaScript does.

Repetition

- In much the same way that Python redefines the `+` operator to indicate string concatenation, it also redefines the `*` operator for strings to indicate repetition, so that the expression `s * n` indicates `n` copies of the string `s` concatenated together.
- The expression `"la" * 3` therefore returns `"lalala"`, which is three copies of the string `"la"` concatenated together.
- Note that this interpretation is consistent with the idea that multiplication is repeated addition:

```
"la" * 3  →  "la" + "la" + "la"
```

- You can use this feature, for example, to print a line of 80 hyphens like this:

```
print("-" * 80)
```

Slicing

- Python allows you to extract a substring by specifying a range of index positions inside the square brackets. This operation is known as ***slicing***.
- The simplest specification of a slice is `[`*start*`:`*stop*`]`, where *start* is the index at which the slice begins, and *stop* is the past-the-end index where the slice ends.
- The *start* and *stop* components of a slice are optional, but the colon must be present. If *start* is missing, it defaults to 0, and if *stop* is missing, it defaults to the length of the string.
- A slice specification may also contain a third component called a ***stride***, as with `[`*start*`:`*stop*`:`*stride*`]`. Strides indicate how many positions are omitted between selected characters.
- The *stride* component can be negative, in which case the selection occurs backwards from the end of the string.

Exercise: Slicing

- Suppose that you have initialized **`ALPHABET`** as

  `ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"`

  so that the index numbers (in both directions) run like this:

A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
–26	–25	–24	–23	–22	–21	–20	–19	–18	–17	–16	–15	–14	–13	–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

- What are the values of the following slice expressions?

(a) **`ALPHABET[7:9]`**

(b) **`ALPHABET[-3:-1]`**

(c) **`ALPHABET[:3]`**

(d) **`ALPHABET[-1:]`**

(e) **`ALPHABET[14:-12]`**

(f) **`ALPHABET[1:-1]`**

(g) **`ALPHABET[0:5:2]`**

(h) **`ALPHABET[::-1]`**

(i) **`ALPHABET[5:2:-1]`**

(j) **`ALPHABET[14:2:-3]`**

Methods for Finding Patterns

str`.find(`*pattern*`)` Returns the first index of *pattern* in *str,* or –1 if it does not appear.
str`.find(`*pattern*`, `*k*`)` Same as the one-argument version but starts searching from index *k*.
str`.rfind(`*pattern*`)` Returns the last index of *pattern* in *str,* or –1 if it does not appear.
str`.rfind(`*pattern*`, `*k*`)` Same as the one-argument version but searches backward from index *k*.
str`.startswith(`*prefix*`)` Returns `True` if this string starts with *prefix*.
str`.endswith(`*suffix*`)` Returns `True` if this string ends with *suffix*.

Methods for Transforming Strings

str`.lower()` Returns a copy of *str* with all letters converted to lowercase.
str`.upper()` Returns a copy of *str* with all letters converted to uppercase.
str`.capitalize()` Capitalizes the first character in *str* and converts the rest to lowercase.
str`.strip()` Removes whitespace characters from both ends of *str*.
str`.replace(`*old*`, `*new*`)` Returns a copy of *str* with all instances of *old* replaced by *new*.

Methods for Classifying Characters

ch`.isalpha()` Returns `True` if *ch* is a letter.
ch`.isdigit()` Returns `True` if *ch* is a digit.
ch`.isalnum()` Returns `True` if *ch* is a letter or a digit.
ch`.islower()` Returns `True` if *ch* is a lowercase letter.
ch`.isupper()` Returns `True` if *ch* is an uppercase letter.
ch`.isspace()` Returns `True` if *ch* is a ***whitespace character*** (space, tab, or newline).
str`.isidentifier()` Returns `True` if this string is a legal Python identifier.