Strings in Python

Jerry Cain
CS 106AX
October 23, 2024

slides leveraged from those constructed by Eric Roberts

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:

h	e	l	l	o	,		w	o	r	l	d
0	1	2	3	4	5	6	7	8	9	10	11

- 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:

h	**e**	**l**	**l**	**o**	**,**		**w**	**o**	**r**	**l**	**d**
–12	–11	–10	–9	–8	–7	–6	–5	–4	–3	–2	–1

- 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)
```

Exercise: String Repetition

- Use string repetition to encode the following songs in as short a Python program as possible:

Let it be
Let it be
Let it be
Let it be
Whisper words of wisdom
Let it be

—Paul McCartney, "Let it Be", 1970

I can feel your halo halo halo
I can see your halo halo halo
I can feel your halo halo halo
I can see your halo halo halo

—Beyoncé, "Halo", 2008

- In 1984, computer Don Knuth published a paper on "The Complexity of Songs" in which he concludes that Casey and the Sunshine Band holds the record for longest song lyric.

```
u = " uh huh" * 2
s = ("That's the way" + u + " I like it" + u + "\n") * ∞
```

Exercise: `removeDoubledLetters`

- In the early part of the 20th century, there was considerable interest in both England and the United States in simplifying the rules used for spelling English words, which has always been a difficult proposition. One suggestion advanced as part of this movement was to eliminate all doubled letters, so that *bookkeeper* would be written as *bokeper* and *committee* would become *comite*. Write a function

  ```
  removeDoubledLetters(s)
  ```

  that returns a new string in which any neighboring duplicated characters in `s` have been replaced by a single copy.

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).

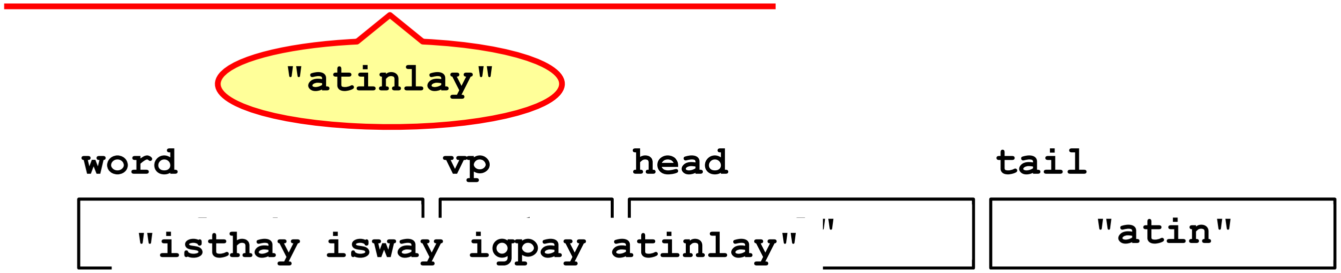
str.**isidentifier()**

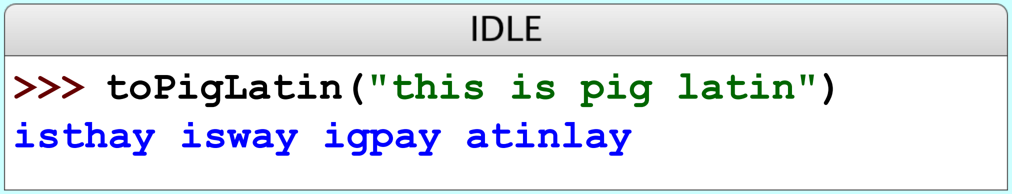
Returns **True** if this string is a legal Python identifier.

Revisiting Pig Latin in Python

```
def toPigLatin(line):
```

```
def wordToPigLatin(word):
    vp = findFirstVowel(word)
    if vp == -1:
        return word
    elif vp == 0:
        return word + "way"
    else:
        head = word[0:vp]
        tail = word[vp:]
        return tail + head + "ay"
```





```
>>> toPigLatin("this is pig latin")
isthay isway igpay atinlay
```

Exercise: Implement `translate`

Pig Latin is just one example of a language game designed to render spoken words to be incomprehensible to the untrained ear. B-Language is another such language game—known primarily in Germany—where words are transformed such that every vowel cluster is reduplicated with a leading `"b"`.

Implement of `translate`, which accepts a lowercase (English) word and returns its B-Language translation.

Examples:

```
translate("quick") ⇒ "quibuick"
translate("spaghetti") ⇒ "spabaghebettibi"
translate("adieu") ⇒ "abadieubieu"
translate("audiophile") ⇒ "aubaudiobiophibilebe"
translate("queueing") ⇒ "queueibueueing"
```

Exercise: Implement `translate`

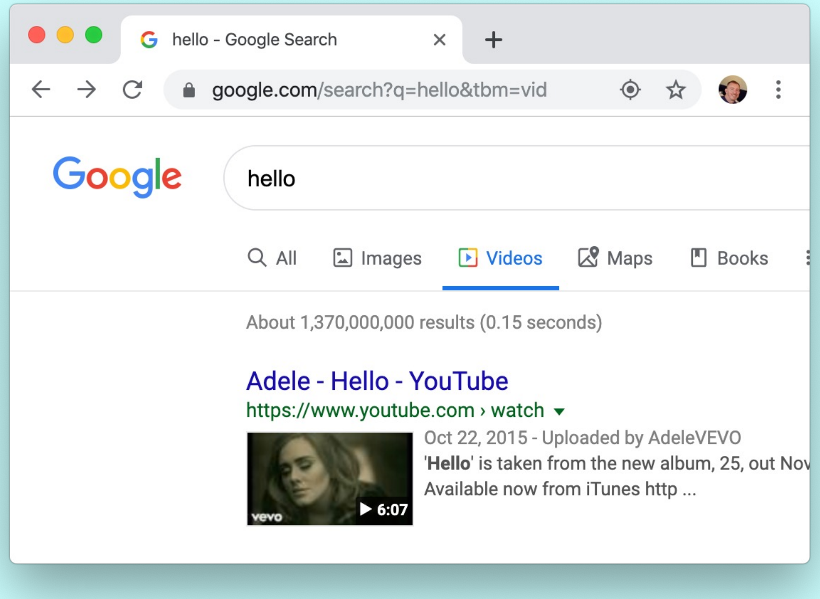
```python
def translate(word, sep='b'):
  """
  Accepts the provided word, assumed to be comprised of
  lowercase letters, and returns its B-Language translation.
  An optional second argument can be supplied to override
  the 'b' with any other string.
  """
  start = -1
  translation = ''
  for i in range(len(word)):
    ch = word[i]
    if isEnglishVowel(ch):
      if start == -1: start = i
    else:
      if start >= 0:
        translation += word[start:i] + sep + word[start:i]
        start = -1
      translation += ch
  if start >= 0:
    translation += word[start:] + sep + word[start:]
  return translation
```

Exercise: Implement `extract`

***U**niform **R**esource **L**ocators*, or *URLs*, are string expressions used to identify where documents can be found on the Internet. Many URLs include *query strings*, which are substrings appearing after a `"?"` that further refine what portion of a document should be retrieved. For example:

```
https://www.google.edu/search?q=hello&tbm=vid
```

lists public *vid*eos associated with `"hello"` , which unsurprisingly includes a whole lot of Adele.



About 1,370,000,000 results (0.15 seconds)

Adele - Hello - YouTube
https://www.youtube.com › watch ▾
Oct 22, 2015 - Uploaded by AdeleVEVO
'**Hello**' is taken from the new album, 25, out Nov
Available now from iTunes http ...

Exercise: Implement **`extract`**

Query strings are really just string serializations of maps in the sense you learned about JavaScript aggregates, where keys are separated from values by equal signs, and key-value pairs are separated from one another by ampersands.

Implement **`extract`**, which lists all key value pairs, one per line, embedded within the query string of a URL. For example:

```
extract("https://explorecourses.stanford.edu/search?view=catalog&"\
        "academicYear=&page=0&q=CS&filter-departmentcode-CS=on&"\
        "filter-coursestatus-Active=on&filter-term-Winter=on")
```

would print the following in the Terminal

```
Key: "view", Value: "catalog"
Key: "academicYear", Value: ""
Key: "page", Value: "0"
Key: "q", Value: "CS"
Key: "filter-departmentcode-CS", Value: "on"
Key: "filter-coursestatus-Active", Value: "on"
Key: "filter-term-Winter", Value: "on"
```

Exercise: Implement **extract**

```
def extract(url):
   """
   Parses the supplied URL and prints out all
   of the key value pairs, one per line.
   """
   qmpos = url.find('?')
   if qmpos == -1: return
   query = url[qmpos + 1:]
   hpos = query.find('#')
   if hpos != -1: query = query[:hpos]
   start = 0
   while start < len(query):
      epos = query.find('=', start)
      key = query[start:epos]
      apos = query.find('&', epos + 1)
      if apos == -1: apos = len(query)
      value = query[epos + 1: apos]
      print('Key: \"{}\", Value: \"{}\"'.format(key, value))
      start = apos + 1
```

The End