CSCI 141

Lecture 17
String Manipulation

Happenings

CS Mentors Present: GITHUB Pages Workshop
Tuesday, November 5th 5:00 PM CF 165

Tech Talk: Pacific Northwest National Labs (PNNL)
Tabling Wednesday, Nov 6th 10:00-3:30 PM CF 4th Floor Foyer
Tech Talk Wednesday, Nov 6th 5:00-6:00 PM CF 115

ACM Hosts: Fast Enterprises
Career Prep Presentation Wednesday, Nov 6th 6:00-7:00 PM CF 316

Group Advising Session for CS Premajors
Thursday, November 7th 3:00-4:30 PM

Announcements

Announcements

- Midterm grades are out - see Canvas announcement for full details.

Announcements

- Midterm grades are out - see Canvas announcement for full details.
  - Review your exam on Gradescope by Wednesday night for 2 bonus points on your exam score

Announcements

- Midterm grades are out - see Canvas announcement for full details.
  - Review your exam on Gradescope by Wednesday night for 2 bonus points on your exam score
  - If you do better on the final, it will replace your midterm grade.

Feedback Survey Results

Feedback Survey Results

Lecture Pace

On average, how would you describe the pace of lectures?

Way too slow	3 respondents	2 %	
Somewhat too slow	31 respondents	18 %	
Just right	85 respondents	49 %	
Somewhat too fast	49 respondents	28 %	
Way too fast	4 respondents	2 %	

Feedback Survey Results

QOTD Review

1	Not helpful at all	1 respondents	1 %	
2		17 respondents	10 %	
3		42 respondents	24 %	
4		64 respondents	37 %	
5	Extremely helpful	48 respondents	28 %	

What shouldn't change?

Common themes, approximately in order of frequency:

- QOTDs
- In-class demos
- Socrative

What should change?

Common themes, approximately in order of frequency:

- Reviewing the QOTD is helpful but takes too much time, so new material is rushed.
- A3 was too hard; I gave too little guidance.
- Demos are helpful, often more so than slides.
- You need more in-class practice, especially coding on paper.

What am I doing about it?

Reviewing the QOTD is helpful but takes too much time, so new material is rushed.

- Short term: Spend less time on the QOTD
- Short term experiment: Written explanations of QOTD
- Long term: Video explanations of QOTD

What am I doing about it?

A3 was too hard; I gave too little guidance.

- Short term: Friday's lecture was all about how to approach A4.

- Long term: Adjust A3 difficulty and give more tips for how to approach each problem.

What am I doing about it?

Demos are helpful, often more so than slides.

You need more in-class practice, especially coding on paper.

- Strive to make slides more concise, talk a little less, and allow time for:
  - More frequent demos with more examples per demo.
  - More in-class exercises, including on-paper coding.

Other

- Many people don't like spending class time answering out-of-scope questions.
- Many people do not like taking CS exams on paper.
- Many people would like me to go over assignment solutions.

Goals

- Review what we know already about strings:
  - the `str` type, `+` and `*` operators, `len` function
- Know how to iterate over tuples and strings using `for` loops
- Know how to index into a string
- Know how Python interprets negative indices into strings.
- Know how to use slicing to get substrings

QOTD

```
import math
def square(x):
    return x ** 2
```

Which of the following are local variables belonging to the `discriminant` function?

`x, a, b, c, disc, b2`

```
def quadratic(a, b, c):
    disc = discriminant(a, b, c)
    return (-b + disc) / (2 * a)

def discriminant(a, b, c):
    b2 = square(b)
    return math.sqrt(b2 - 4 * a * c)

print(quadratic(4, 6, 2))
```

Last time: tuples

- A tuple is a sequence of values, optionally enclosed in parens.

(of any types!)

```
(1, 4, “Mufasa”)
```

- You can “pack” and “unpack” them using assignment statements:

```
v = (1, 4, "Mufasa") # "packing"

(a, b, c) = v # "unpacking"
```

QOTD

Run the following code. If a line causes an error, skip it and continue execution. Which lines, if any, cause errors?

```
1    a, b, c = 6, 4, 2
2    (z, x) = c, b
3    print((x, z))
4    v = (x, z, c)
5    print(v)
6    a, b = v
```

QOTD

What does line 5 print?

```
1    a, b, c = 6, 4, 2
2    (z, x) = c, b
3    print((x, z))
4    v = (x, z, c)
5    print(v)
6    a, b = v
```

A. (2, 4, 2)
B. (2, 2, 4)
C. (6, 4, 2)
D. (4, 2, 2)
E. (4, 4, 2)

fun fact:

Tuples are sequences,

so they can be used in `for` loops just like lists and ranges.

These two loops do the same thing:

```
for number in [1, 3]:
    print(number, end=" ")
```

```
for number in (1, 3):
    print(number, end=" ")
```

fun fact:

Tuples are sequences,

so they can be used in `for` loops just like lists and ranges.

These two loops do the same thing:

```
for number in [1, 3]:
    print(number, end=" ")
```

```
for number in (1, 3):
    print(number, end=" ")
```

Exercise: write a `for` loop that uses a `range` to print the same thing.

Today: Strings

Don’t we already know about strings?

Today: Strings

Don’t we already know about strings?

```
type("hello")
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>
```

```
print("Hello")
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>
```

```
print("Hello")    # prints Hello to the console
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”

len("abc")
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”

len("abc")         # => 3
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”

len("abc")         # => 3

"na" * 16 + " Batman!"
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”

len("abc")         # => 3

"na" * 16 + " Batman!"

# => …
```

Today: Strings

Don’t we already know about strings?

```
type("hello")    # => <class ‘str’>

print("Hello")   # prints Hello to the console

"Hello" + “World"  # => “HelloWorld”

len("abc")         # => 3

"na" * 16 + " Batman!"

# => …"nanananananananananananananananana Batman!"
```

Strings: What else is there?

Strings: What else is there?

```
def house_number(address_line):
```

Strings: What else is there?

```
def house_number(address_line):
    """ Return the house number portion of
        the given address line.
```

Strings: What else is there?

```
def house_number(address_line):
    """ Return the house number portion of
        the given address line.
        Examples:
```

Strings: What else is there?

```
def house_number(address_line):
    """ Return the house number portion of
        the given address line.
        Examples:
            house_number("1600 Pennsylvania Ave")
            => 1600
```

Strings: What else is there?

```
def house_number(address_line):
    """ Return the house number portion of
        the given address line.
        Examples:
            house_number("1600 Pennsylvania Ave")
              => 1600
            house_number("221B Baker St")
              => 221
    """
```

Strings: What else is there?

```
def house_number(address_line):
    """ Return the house number portion of
        the given address line.
        Examples:
            house_number("1600 Pennsylvania Ave")
              => 1600
            house_number("221B Baker St")
              => 221
    """
    # ????
    return result
```

Strings: What else is there?

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with and following
        the first instance of '#' emoved. If there
        is no # in the string, return input unchanged.
    """
```

Strings: What else is there?

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with and following
        the first instance of '#' emoved. If there
        is no # in the string, return input unchanged.
    """
    # ????
```

Strings: What else is there?

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with and following
        the first instance of '#' emoved. If there
        is no # in the string, return input unchanged.
    """
    # ????
    return result
```

fun fact:

Strings are sequences,

so they can be used in for loops just like lists and ranges.

Check this out:

```
for letter in "Bellingham":
    print(letter, "-", sep="", end="")
```

fun fact:

Strings are sequences,

so they can be used in for loops just like lists and ranges.

Check this out:

```
for letter in "Bellingham":
    print(letter, "-", sep="", end="")
```

Demo?

fun fact:

Strings are sequences,

so they can be used in for loops just like lists and ranges.

Check this out:

```
for letter in "Bellingham":
    print(letter, "-", sep="", end="")
```

fun fact:

Strings are sequences,

so they can be used in for loops just like lists and ranges.

Check this out:

```
for letter in "Bellingham":
    print(letter, "-", sep="", end="")
```

What does this print?

A. `Bellingham`
B. `B-e-l-l-i-n-g-h-a-m`
C. `-B-e-l-l-i-n-g-h-a-m`
D. `B-e-l-l-i-n-g-h-a-m-`

Exercise (worksheet #1)

Write a function that **prints** a string with all vowels removed.

```
def remove_vowels(string):
    """ Print string, but with no vowels.
        Don't count y as a vowel. """
```

Exercise (worksheet #1)

Write a function that **prints** a string with all vowels removed.

```
def remove_vowels(string):
    """ Print string, but with no vowels.
        Don't count y as a vowel. """
```

Possible modification: Return the result instead of printing it.

Indexing into Strings

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?

Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?

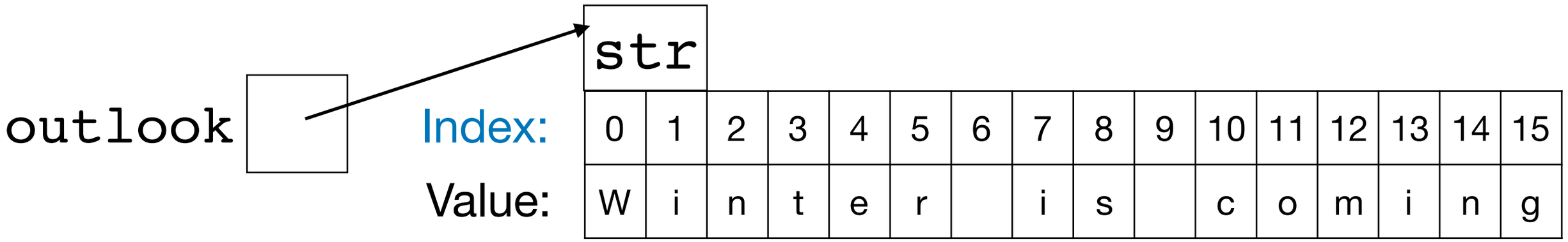
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



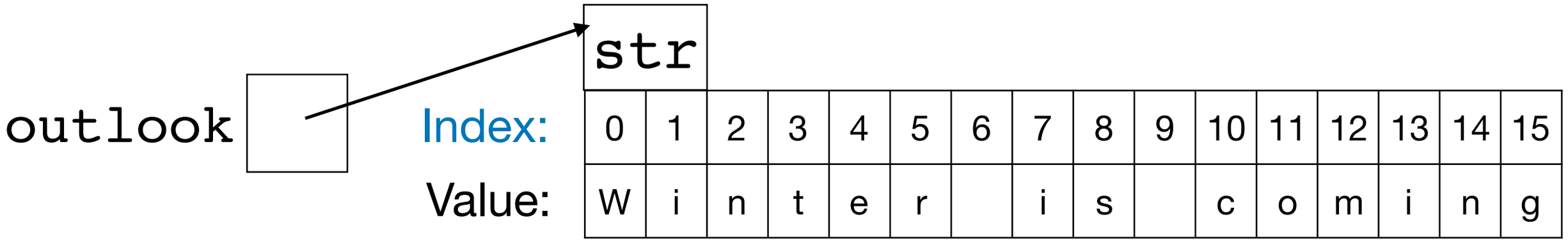
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

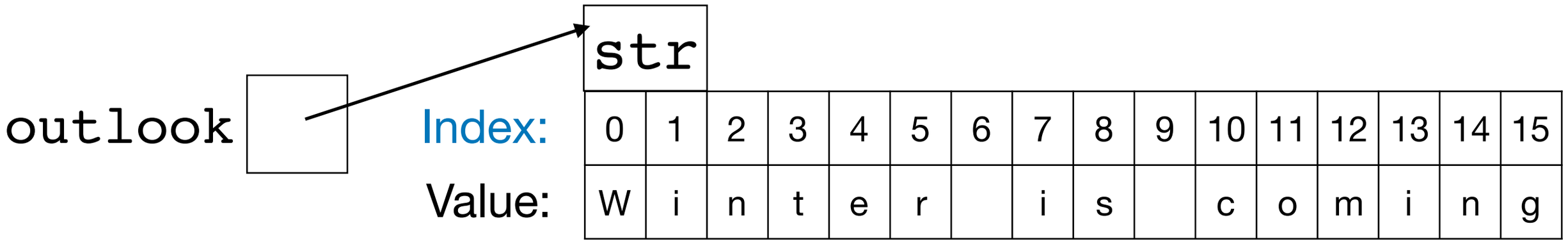
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

Syntax:

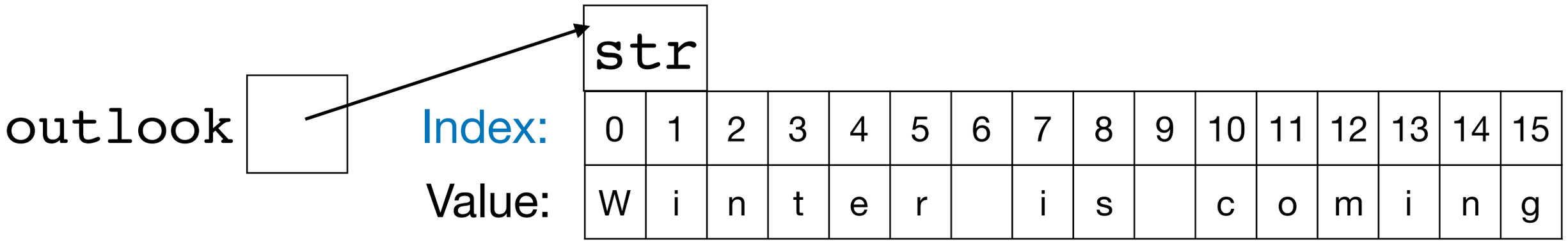
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

Syntax:

```
outlook[0] # => "W"
```

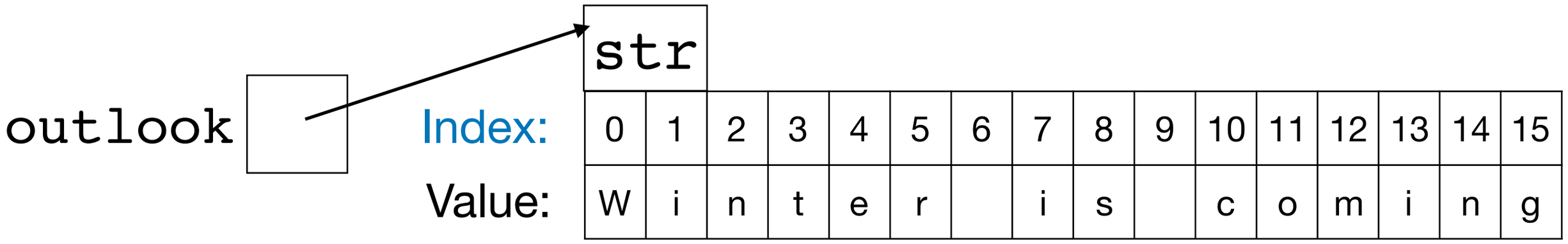
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

Syntax:

```
outlook[0] # => "W"
outlook[4] # => "e"
```

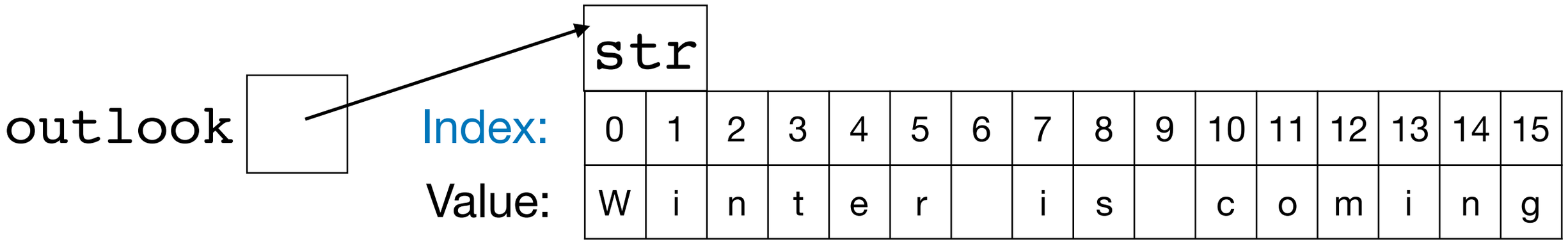
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

Spaces are characters too!

Syntax:

```
outlook[0] # => "W"
outlook[4] # => "e"
```

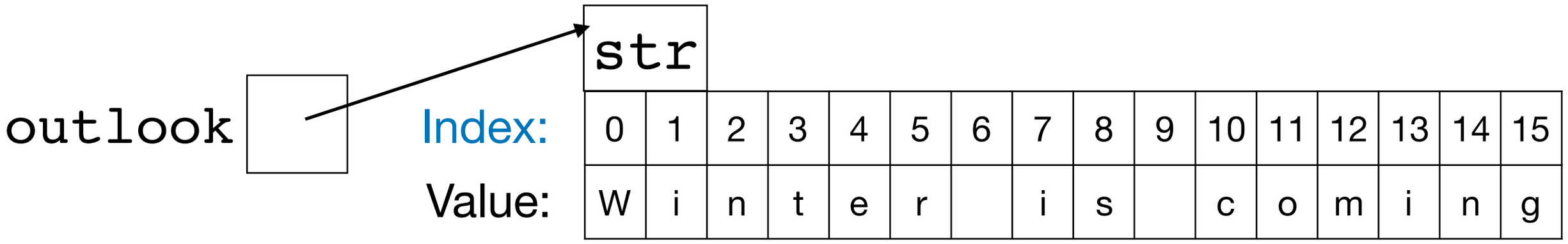
Indexing into Strings

(just smaller strings!)

Strings are collections of individual characters.
We can get access to an individual character by index.

```
outlook = “Winter is coming”
```

How is this stored in memory?



Indices in Python begin at 0.

Spaces are characters too!

Syntax:

```
outlook[0] # => "W"
outlook[4] # => "e"
outlook[6] # => " "
```

Indexing into Strings

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value:	W	i	n	t	e	r		i	s		c	o	m	i	n	g

Assume `s` is a variable that refers to the above string object.
How would I access the letter `'r'`?

```
A. s[5]
B. s(5)
C. s[6]
D. s(6)
```

Indexing into Strings

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Value:	W	i	n	t	e	r		i	s		c	o	m	i	n	g

What is the index of the last character of a string `s`?
(not the specific string above - this should work for **any** string)

```
A. len(s) - 1
B. len(s)
C. len(s) + 1
D. 42
```

A consequence of indexing - Another way to loop through strings:

```
for letter in a_string:
    print(letter, "-", sep="", end="")
```

is equivalent to

```
for i in range(len(a_string)):
    print(a_string[i], "-", sep="", end="")
```

Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:

Also Index:

Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	W	i	n	t	e	r		i	s		c	o	m	i	n	g
Also Index:	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	W	i	n	t	e	r		i	s		c	o	m	i	n	g
Also Index:	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Two possible ways to remember how this works:

Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	W	i	n	t	e	r		i	s		c	o	m	i	n	g
Also Index:	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Two possible ways to remember how this works:

```
a_string[-5]
```

is equivalent to

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

Nifty Python Feature: Negative Indices

Negative indices count backwards from len(s):

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	W	i	n	t	e	r		i	s		c	o	m	i	n	g
Also Index:	-16	-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

Two possible ways to remember how this works:

-1 is always the last character, and indices count backwards from there.

```
a_string[-5]
```

is equivalent to

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

Negative Indices!

For which assignment of `a` and `b` does the above **not** print `True`?

A.
```
a = 1
b = 5
```

B.
```
a = 1
b = 7
```

C.
```
a = -8
b = -4
```

D.
```
a = -2
b =  6
```

Negative Indices!

```
last_name = "wehrwein"
```

For which assignment of `a` and `b` does the above **not** print `True`?

A.
```
a = 1
b = 5
```

B.
```
a = 1
b = 7
```

C.
```
a = -8
b = -4
```

D.
```
a = -2
b =  6
```

Negative Indices!

```
last_name = "wehrwein"
```

For which assignment of `a` and `b` does the above **not** print `True`?

A.
```
a = 1
b = 5
```

B.
```
a = 1
b = 7
```

C.
```
a = -8
b = -4
```

D.
```
a = -2
b =  6
```

Negative Indices!

```
last_name = "wehrwein"

print(last_name[a] == last_name[b])
```

For which assignment of `a` and `b` does the above **not** print `True`?

A.
```
a = 1
b = 5
```

B.
```
a = 1
b = 7
```

C.
```
a = -8
b = -4
```

D.
```
a = -2
b =  6
```

Negative Indices!

```
last_name = "wehrwein"

print(last_name[a] == last_name[b])
```

For which assignment of `a` and `b` does the above **not** print `True`?

A.
```
a = 1
b = 5
```

B.
```
a = 1
b = 7
```

C.
```
a = -8
b = -4
```

D.
```
a = -2
b =  6
```

Indexing

gives us other ways to loop through strings:

```
for letter in a_string:
    print(letter, end="")
```

is equivalent to

```
for i in range(len(a_string)):
    print(a_string[i], end="")
```

and also

```
i = 0
while i < len(a_string):
    print(a_string[i], end="")
    i += 1
```

Exercise (worksheet #2)

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with and following
        the first instance of '#' emoved. If there
        is no # in the string, return input unchanged.
    """


# Example:

remove_comments("a = b # assign b to a"))
# => "a = b "
```

Exercise (worksheet #2)

```
def remove_comments(string):
    """ Return a copy of string, but with
        all characters starting with and following
        the first instance of '#' emoved. If there
        is no # in the string, return input unchanged.
    """
```

Hint: use a `while` loop!

```
# Example:

remove_comments("a = b # assign b to a"))
# => "a = b "
```

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

Slicing syntax: `string[start:end]`

1 + index of last character

just like the `range` function:
the end index is **not** included

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character → `start`

1 + index of last character → `end`

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character — 1 + index of last character

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

```
alph[0:5] # => “abcde"
```

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character — 1 + index of last character

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

```
alph[0:5] # => “abcde"

alph[0:10] # => "abcdefghij"
```

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character — 1 + index of last character

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

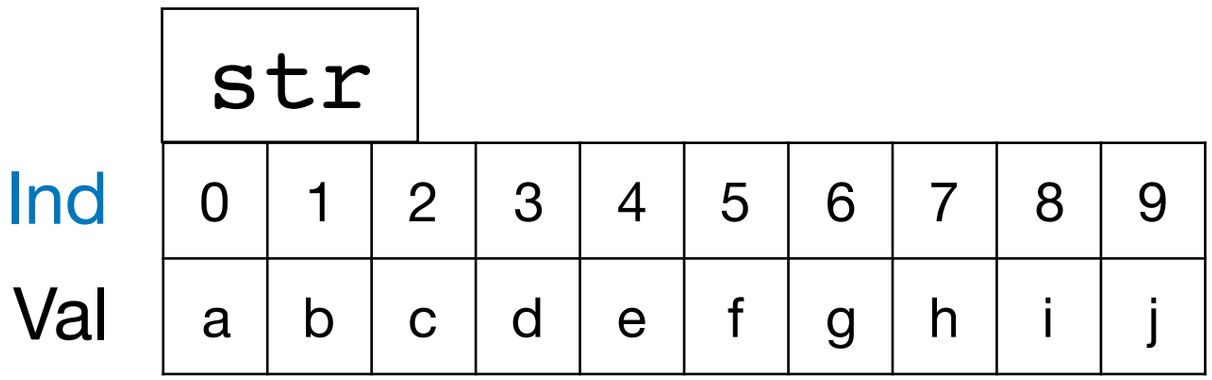
```
alph[0:5] # => “abcde"

alph[0:10] # => "abcdefghij"

alph[5:-2]
```

Slicing: indexing substrings

```
alph = "abcdefghij"
alph[0] # => "a"
alph[4] # => "e"
```



index of first character — 1 + index of last character

Slicing syntax: `string[start:end]`

just like the `range` function:
the end index is **not** included

```
alph[0:5] # => “abcde"

alph[0:10] # => "abcdefghij"

alph[5:-2] # => "fgh"
```

Slicing: indexing substrings

```
alph = "abcdefghij"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character → `start`

1 + index of last character → `end`

Slicing syntax: `string[start:end]`

Slicing: indexing substrings

```
alph = "abcdefghij"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

index of first character — 1 + index of last character

Slicing syntax: `string[start:end]`

If omitted, *`start`* defaults to `0`

Slicing: indexing substrings

```
alph = "abcdefghij"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

Slicing syntax: `string[start:end]`

`start`: index of first character

`end`: 1 + index of last character

If omitted, `start` defaults to `0`

If omitted, `end` defaults to `len(string)`

Slicing: indexing substrings

```
alph = "abcdefghij"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

Slicing syntax: `string[start:end]`

`start`: index of first character

`end`: 1 + index of last character

If omitted, `start` defaults to `0`

If omitted, `end` defaults to `len(string)`

```
alph[:4] # => “abcd"
```

Slicing: indexing substrings

```
alph = "abcdefghij"
```

	str									
Ind	0	1	2	3	4	5	6	7	8	9
Val	a	b	c	d	e	f	g	h	i	j

Slicing syntax: `string[start:end]`

- `start`: index of first character
- `end`: 1 + index of last character

If omitted, `start` defaults to `0`

If omitted, `end` defaults to `len(string)`

```
alph[:4] # => “abcd"
alph[5:] # => "fghij"
```

String Slicing: Exercise

	str							
Ind	0	1	2	3	4	5	6	7
Val	W	e	h	r	w	e	i	n

Which of these evaluates to `"in"`?

A. `last_name[7:8]`
B. `last_name[6:-1]`
C. `last_name[-3:]`
D. `last_name[-2:8]`

String Slicing: Exercise

```
last_name = "Wehrwein"
```

	str							
Ind	0	1	2	3	4	5	6	7
Val	W	e	h	r	w	e	i	n

Which of these evaluates to `"in"`?

A. `last_name[7:8]`
B. `last_name[6:-1]`
C. `last_name[-3:]`
D. `last_name[-2:8]`