15-112
Lecture 2

Strings

Instructor: Pat Virtue

Tuesday Logistics

As you walk in

Quiz will start at the beginning of lecture

- Have pencil/pen ready
- Silence phones

Quiz

Before we start

- Don’t open until we start
- Make sure your name and Andrew ID are on the front
- Read instruction page
- No questions (unless clarification on English)

Additional info

- 25 min

Announcements

Quiz

- Grades
- Regrade requests, same as last week
- Fix-its, same as last week

Canvas

- We’re still organizing and getting basics setup
- Participation is next to setup
  - Participation will start to include recitation attendance

Course

- It will keep ramping up
- Come get help

Announcements

Weekly Rhythm Assignments/Quizzes

- Today, HW3 released
- Thu, Pre-reading 4 released
- Sat, 8 pm: HW 4
- Mon, 8 pm: Pre-reading 4
- Next Tue, in-lec: Quiz 3

Thursday Logistics

Announcements

Quiz

- Review quiz results in Gradescope
- Watch solution session recording if you missed the live zoom session
- Regrade requests
  - See Piazza for details
- Fix-its!
  - See Piazza for details

Canvas

- Work in progress: we're getting scripts setup to sync Canvas Grades
- TODO: Participation: Lecture Polls + Recitation Attendance

Announcements

Weekly Rhythm Assignments/Quizzes

- Today, Pre-reading 4 released soon
- Fri: Fix-its due
- Sat, 8 pm: HW 3
- Mon, 8 pm: Pre-reading 4
- Next Tue, in-lec: Quiz 3

Strings

Post-quiz Exercise

What is the correct response to the following?

```
             01234
pet = "manatee"
s = pet[:4]*2
```

temp= pet[:4] ← "mana"

s = "mana" *2

Then Google search: s

Poll 1

What does this print?

A. A

B. B

C. C

D. D

[✓ E. E]

F. F

G. G

H. None of the above

```
def ct(s):
    n = ord(s)
    n += 2
    return chr(n)

print(ct('C'))
```

ord

str → int

chr

int → str

$n = \cancel{67}\ 69$

Ascii, Unicode, and Emojis!

(and a tiny bit of hexadecimal)

Viewing invisible characters

```
repr(s)
```

Poll 2

What does this code print?

```
# 1234567890
print(len('\noodles\\'))
```

1 2 3 4 5 6 7 8

A. 6

B. 7

C. 8 43%

D. 9

E. 10

F. 11

G. (Python crashes)

H. I have no idea

Esacape characters

Popular escape characters

\n	New line
\t	Tab
\\	\

String length, indexing, and slicing

```
    012345678
s = 'brown cat'
len(s)   9

s[2]     'o'

s[-2]    'a'

s[1:7:3]
```

start end step

$n = \text{len}(s) \longrightarrow \underline{5}$

0	1	2	3	4
b	r	o	w	n
n-5	n-4	n-3	n-2	n-1
-5	-4	-3	-2	-1

Poll 3 (unused)

Which is better?

A)

```
# Given string s
for i in range(len(s)):
    # Do stuff
    c = s[i]
    print(c)
```

B)

```
# Given string s
for c in s:
    # Do stuff
```

Poll 4

What does this code print?

A. abcde

B. edcba

C. bcdea

D. bcda

E. ba

F. ab

G. (Python crashes)

H. I have no idea

```
def ct(s):
    return s[1:-1] + s[0]

print(ct('abcde’))
```

String indexing and slicing

Reference slide

Indexing

```
c = s[index] # c will be character at position index
```

Valid indices

- Positive: 0 to len(s)-1 (but not len(s))
- Negative: -len(s) to -1

Slicing

```
s[start:end:step]
```

Similar to range arguments

- Doesn’t include end
- There are default values if any of these are left blank
- (Gets a bit goofy with a negative step)

Poll 5

What does this function do?

A. Return a copy of s

[✓ B.] Return the reverse of s

C. Return string that is only the last character of s

D. Return string that is only the first character of s

E. Return None

F. (Python crashes)

G. I have no idea

```
def revStr(s)
def ~~mystery~~(s):
    return s[::-1]
```

Pattern: Building up a result

Building up a string

String operations

- `sNew = s1 + s2`
- `sNew += s3`

Sketch:

- Start with empty string: `result = ''`
- Loop
  - adding to string as needed: `result += nextChar`

Example: reverseString(s)

Pattern: Building up a result

Building up a string

String operations

- `sNew = s1 + s2`
- `sNew += s3`

Sketch:

- Start with empty string: `result = ''`
- Loop
  - adding to string as needed: `result += nextChar`

Example:

```python
def reverseString(s):
    newString = ''
    for c in s:
        newString = c + newString

    return newString
```

c

'xyz'

new = 'x' + ''

new = 'y' + 'x'

new = 'z' + 'yx'

Poll 6

What does this print?

→ A. dog

B. DOG

→ C. mog

D. MOG

E. mOG

[✓ F.] (Python crashes)

G. I have no idea

```
s = 'dog'
s.upper()
s[0] = 'm'
print(s)
```

dog

→

X

XDOG

mOG

Functions vs Methods

String functions take in a string (return something useful)

Like a all the functions that we've been working with

```
x = chr(s)
    ord(s)
    len(s)
    repr(s)
```

Methods on the other hand have a different syntax:

```
x = s.upper()
obj.name(arg1, arg2 ...)
```

String methods

Reference slide

Some convenient methods that return Boolean values

s.is___()

s	isalnum	isalpha	isdigit	islower	isspace	isupper
ABCD	True	True	False	False	False	True
ABcd	True	True	False	False	False	False
abcd	True	True	False	True	False	False
ab12	True	False	False	True	False	False
1234	True	False	True	False	False	False
	False	False	False	False	True	False
AB?!	False	False	False	False	False	True

Strings are immutable

Once a string object is created, we can’t change it.

This is what we call “immutable”

Actually, everything we have used so far is immutable: ints, floats, etc. (they just aren’t very interesting objects)

It might see as though you can change strings but we can’t. It always ends up as some new string object.

This will be much more relevant once we get to our first mutable object type, lists!

Poll 7

What does this print?

[✓ A. lil]

B. nasx

C. lilnasx

D. (Python crashes)

E. I have no idea

```
s = 'lil'
t = s
s += 'nasx'

print(t)
```

s = $\underline{\text{s + 'nasx'}}$

print(s) lilnasx ✓

s = $\underline{\text{'lil' + 'nasx'}}$

Strings and aliases

Two variables are “aliases” are when they reference the exact same object.

This happens when you assign a variable to another variable:

```
s = 'abc'
t = s
```

s and t are aliases referencing the same to the same exact string object 'abc’

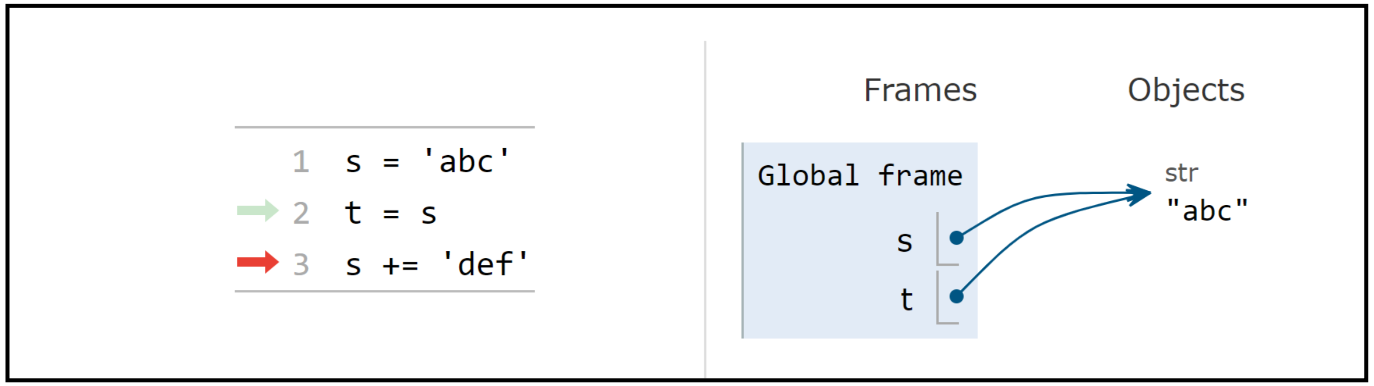
But…strings are immutable. We can’t possibly change s without making a new string.

```
s += 'def' # Assigns s to a new string 'abcdef’
# The string t is referencing remains 'abc'
```

Strings and aliases

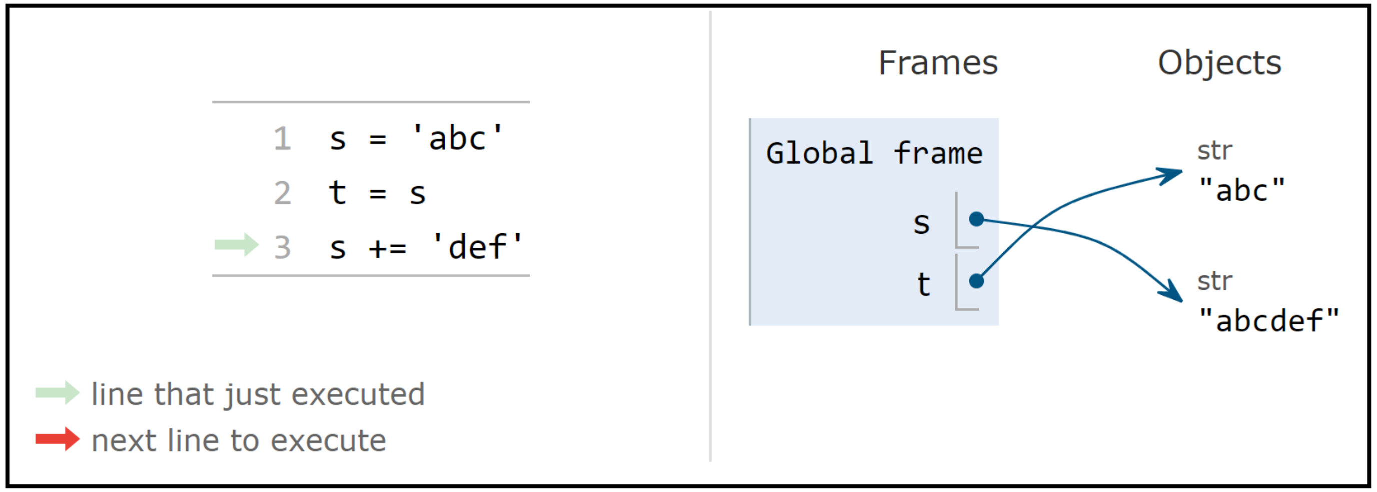
Two variables are “aliases” are when they reference the exact same object.

This happens when you assign a variable to another variable



```
s = 'abc'
t = s
s += 'def'
```

But…strings are immutable. We can’t possibly change s without making a new string.



```
s = 'abc'
t = s
s += 'def'
```

Pattern: Building up a result

Building up a string

String operations

- `sNew = s1 + s2`
- `sNew += s3`

Sketch:

- Start with empty string: `result = ''`
- Loop
  - adding to string as needed: `result += nextChar`

Example:

```python
def reverseString(s):
    newString = ''
    for c in s:
        newString = c + newString

    return newString
```

$$\begin{tikzcd}
c \arrow[d] \\
'xyz'
\end{tikzcd}$$

new = 'x' + ''

new = 'y' + 'x'

new = 'z' + 'yx'

Pattern: Keeping track of state in a loop

Use a variable to keep track of the state you are in during a loop

Sketch:

- Start with initial state:
  `currentState = False`
- Loop
  - Check for changes and adjust state variable

Example:

Collapse consecutive whitespace down to a single space

```
def collapseWhitespace(s):
    result = ''
    isWhite = False
    for c in s:
        if c.isspace():
            if not isWhite:
                result += ' '
            isWhite = True
        else:
            isWhite = False
            result += c

    return result
```

Design Challenge

```
def toCamelCase(s):
    pass
```

Style: variable/function names

- Camel case: `myNewVariable`
- Snake case: `my_new_variable`

```
assert(toCamelCase('goodToGo') == 'goodToGo')


assert(toCamelCase(‘Hi Walter') == 'hiWalter')


assert(toCamelCase('add_all_the_numbers') == 'addAllTheNumbers')
```

Design Challenge

```
def toCamelCase(s):
    pass
```

Style: variable/function names

- Camel case: `myNewVariable`
- Snake case: `my_new_variable`

```
assert(toCamelCase('goodToGo') == 'goodToGo')
assert(toCamelCase('goodtogo') == 'goodtogo') # Oh well    ←

assert(toCamelCase('Hi Walter') == 'hiWalter')


assert(toCamelCase('add_all_the_numbers') == 'addAllTheNumbers')
```

Poll 8 (unused)

Which of the following would you want to use in your `toCamelCase(s)` implementation?

Select ALL that apply

A. For loop over the characters in the string
B. `isalnum`
C. `isalpha`
D. `isdigit`
E. `islower`
F. `isspace`
G. `isupper`
H. None of the above

Design Challenge

```
def printFunctionInfo(code):
    pass
```

params.split(';')

```
code = """\
def f(x):
    return 4*x

def ct(x, y, z):
    return f(x) + f(y+1) + f(x+2)

print(ct(2))
"""


print(code)
#print(repr(code))

printFunctions(code)
```

Output

```
Function f takes parameters:
    x
Function ct takes parameters:
    x
    y
    z
```

Design Challenge

```
def printFunctionInfo(code):
    for line in code.splitlines():
        name, params = parseLine(line)

        if name is not None:
            print(f'Function {name} takes parameters:’)
            for param in params.split(',’):
                param = param.strip()
                print(f"\t{param}")
```

Output

```
Function f takes parameters:
      x
Function ct takes parameters:
      x
      y
      z
```

Top-down design

```
def parseLine(line):
    '''
    Extract the function name and parameters given a line of code
    Returns two strings:
    -- Function name (no def, no parentheses)
    -- parameters (one string that includes any commas; may be empty string)
    Returns None, None if no function defined on the line
    '''
```

Design Challenge

Pattern: Building up a result

```
def parseLine(line):
    ''' Function comment ... '''
    line = line.strip()
    if line[4:] != "def ":
        return None
    remainingLine = line[4:]
    name = ""
    params = ""
    foundOpenParenthesis = False
    for c in remainingLine:
        if c == '(':
            foundOpenParenthesis = True
            continue
        if c == ')':
            break
        if not foundOpenParenthesis:
            name += c
        else:
            params += c
    return name, params
```

Design Challenge

```
def parseLine(line):
    ''' Function comment ... '''
    line = line.strip()
    if line[4:] != "def ":
        return None
    remainingLine = line[4:]
    name = ""
    params = ""
    foundOpenParenthesis = False
    for c in remainingLine:
        if c == '(‘:
            foundOpenParenthesis = True
            continue
        if c == ')’:
            break
        if not foundOpenParenthesis:
            name += c
        else:
            params += c
    return name, params
```

Pattern: State in a loop