CS 1124 Media Computation

A Mind Bending Lecture on our Way to Java

Lecture 9.2 October 22, 2008

Steve Harrison

TODAY

- Strings
  - dot.notation
  - and a little bit more about lists...
- A jump into hyperspace

TODAY

- Strings
  - dot.notation
  - and a little bit more about lists...
- A jump into hyperspace

Dot notation

MORE ABOUT THIS LATER !

- All data in Python are actually *objects*
- Objects not only store data, but they respond to special functions that only objects of the same type understand.
- We call these special functions *methods*
  - **Methods are functions known only to certain objects**
- To execute a method, you use *dot notation*
  - **objectName.method()**

Capitalize is a method known only to strings

```
>>> test="this is a test."
>>> print test.capitalize  # without the ()s a method
   will not execute
<built-in method 'capitalize'>
>>> print test.capitalize()
This is a test.
>>> print capitalize(test)
A local or global name could not be found.
NameError: capitalize
>>> print 'this is another test'.capitalize()
This is another test
>>> print 12.capitalize()
A syntax error is contained in the code -- I can't
   read it as Python.
Why?
```

Strings are sequences

```
>>> for i in "Hello":
...             print i
...
H
e
l
l
o
```

Useful string methods (1)

- **strObject.startswith(prefix)** returns true if the string starts with the given prefix
- **strObject.endswith(suffix)** returns true if the string ends with the given suffix

Want to test how a string starts?

```
>>> letter = "Mr. Steve Harrison requests
  the pleasure of your company..."
>>> print letter.startswith("Mr.")
1
>>> print letter.startswith("Mrs.")
0
```

Remember that Python sees "0" as false
and
anything else (including "1") as true

Or how it ends?

```
>>> filename="barbara.jpg"
>>> if filename.endswith(".jpg"):
...           print "It's a picture"
...
It's a picture
>>> file = "srh@cs.vt.edu"
>>> if file.endswith(".edu"):
...               print "looks like a school
  email address."
...
looks like a school email address.
```

Useful string methods (2)

- **strObject.find(pattern)** and **strObject.find(pattern, start)** and **strObject.find(pattern, start, end)** finds the pattern in the string object and returns the index where the pattern starts. You can tell it what index number to start from, and even where to stop looking. **It returns -1 if it fails.**
- **strObject.rfind(pattern)** (and variations) searches from the *end* of the string.

Remember that Python sees "0" as false and anything else (including "1") as true

But strObject.find() returns -1 when false -- that is, can't find string in strObject

Demonstrating find

```
>>> print letter
Mr. Steve Harrison requests the pleasure of your
   company...
>>> print letter.find("Steve")
4
>>> print letter.find("Harrison")
10
>>> print len("Harrison")
8
>>> print letter[4:(4+6)+8]
Steve Harrison
>>> print letter.find("fred")
-1
```

Replace method

```
>>> print letter
Mr. Steve Harrison requests the pleasure of your
   company...
>>> letter.replace("a", "!")
'Mr. Steve H!rrison requests the ple!sure of your comp!
   ny...'
>>> print letter
Mr. Steve Harrison requests the pleasure of your
   company...
```

N.B. The string that is stored in letter did not change. The replace method creates a new string with the replacement having happened. Store that if you want the string in letter to change.

```
letter = letter.replace("S", "s")
```

TODAY

- Strings
  - dot.notation
  - and a little bit more about lists...
- A jump into hyperspace

Converting from strings to lists

```
>>> print letter.split(" ")
['Mr.', 'Steve', 'Harrison', 'requests',
  'the', 'pleasure', 'of', 'your',
  'company...']
```

N.B. this split is splitting on a space. You can split on other characters too!

Talk Like a Computer Scientist

```
>>> print letter.split(" ")
['Mr.', 'Steve', 'Harrison', 'requests',
  'the', 'pleasure', 'of', 'your',
  'company...']
```

Cutting up a string into parts like this is called "parsing" or "tokenizing". This is useful since it gives structure where there was none.

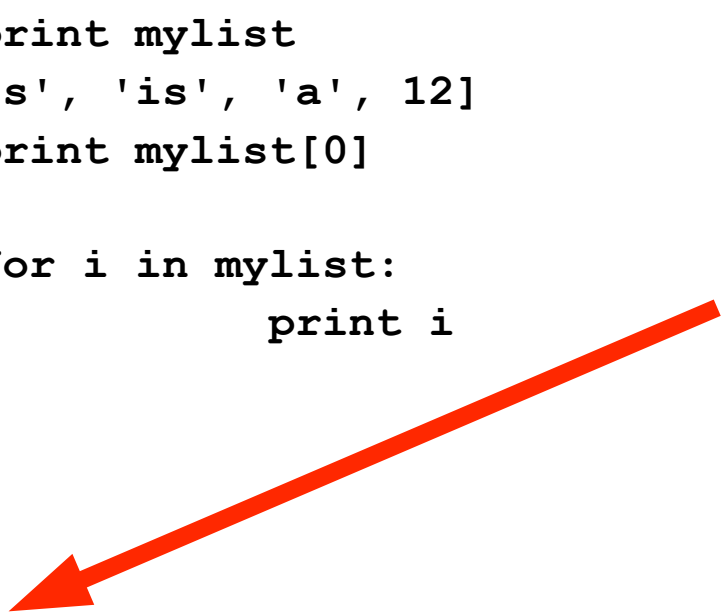
Even thought they mean slightly different things, computer scientists often use "parse" and tokenize in casual conversation as a synonym for "understand". For example, they will say "I can't parse what you are saying."

Lists

- We've seen lists before—that's what **range()** returns.
- Lists are very powerful structures.
  - **Lists can contain strings, numbers, even other lists.**
  - **They work very much like strings**
    - You get pieces out with []
    - You can "add" lists together
    - You can use **for** loops on them
  - **We can use them to process a variety of kinds of data.**

Demonstrating lists

```
>>> mylist = ["This", "is", "a", 12]
>>> print mylist
['This', 'is', 'a', 12]
>>> print mylist[0]
This
>>> for i in mylist:
...             print i
...
This
is
a
12
>>> print mylist + ["Really!"]
['This', 'is', 'a', 12, 'Really!']
```



Whoa! Lists can have different kinds of objects in them.

N.B. Again assign that back into mylist to update mylist's value!

```
>>> mylist = mylist + ["Really!"]
```

Useful methods to use with lists: But these don't work with strings

- **append(something)** puts something in the list at the end.
- **remove(something)** removes something from the list, if it's there.
- **sort()** puts the list in alphabetical order
- **reverse()** reverses the list
- **count(something)** tells you the number of times that something is in the list.
- **max()** and **min()** are **functions** that take a list as input and give you the maximum and minimum value in the list.

TODAY

- Strings
  - dot.notation
  - and a little bit more about lists...
- A jump into hyperspace

Question:

- Give a definition of “time”
- Can a function call itself?

A very powerful idea: Recursion

- Recursion is writing functions that call *themselves*.
- When you write a recursive function, you write (at least) two pieces:
  - **What to do if the input is the smallest possible datum,**
  - **What to do if the input is larger so that you:**
    - (a) process one piece of the data
    - (b) call the function to deal with the rest.

- SEE CHAPTER 14 FOR MORE ON RECURSION

First, a reminder of lists

```
>>> fred=[1,2,3,4,5]
>>> fred[0]
1
>>> fred[1:]
[2, 3, 4, 5]
```

- In functional programming languages, there are usually functions called **head** and **rest** for these two operations.
  - **They're very common in recursion.**

```
>>> print fred[:-1]
[1, 2, 3, 4]
```

A recursive decreaseRed

```
def decreaseRed(alist):
  if alist == []:   #Empty
    return
  setRed(alist[0],
    getRed(alist[0])*0.8)
  decreaseRed(alist[1:])
```

This actually won't work for reasonable-sized pictures—takes up too much memory in Java.

- If the list (of pixels) is empty, don't do anything.
  - **Just return**
- Otherwise,
  - **Decrease the red in the first pixel.**
  - **Call decreaseRed on the rest of the pixels.**
- Call it like: decreaseRed(getPixels(pic))

Recursion can be hard to get your head around

- It really relies on you *trusting* your functions.
  - **They'll do what you tell them to do.**
  - **So if a function decreases red on a list of pixels, just let it do that!**
- Let's try some different ways to think about recursion.
- But first, let's take a smaller problem.

DownUp

- Let's define a function called **downUp**

```
>>> downUp("Hello")
Hello
ello
llo
lo
o
lo
llo
ello
Hello
```

3 ways to understand recursion

1. Procedural abstraction
2. Trace it out (use a small problem like **downUp** to do this)
3. Little people method

1. Procedural abstraction

- Break the problem down into the smallest pieces that you can write down easily as a function.
- Re-use as much as possible.

downUp for one character words

```
def downUp1(word):
  print word
```

- Obviously, this works:

```
>>> downUp1("I")
I
```

downUp for 2 character words

- We'll reuse downUp1 since we have it already.

```
def downUp2(word):
  print word
  downUp1(word[1:])
  print word
>>> downUp2("it")
it
t
it
>>> downUp2("me")
me
e
me
```

downUp3 for 3 character words

```
def downUp3(word):
  print word
  downUp2(word[1:])
  print word
>>> downUp3("pop")
pop
op
p
op
pop
>>> downUp3("top")
top
op
p
op
top
```

Are we seeing a pattern yet?

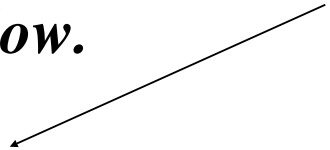
Let's try our pattern

```
def downUpTest(word):
  print word
  downUpTest(word[1:])
  print word
```

It starts right!

```
>>> downUpTest("hello")
hello
ello
llo
lo
o
I wasn't able to do what you wanted.
The error java.lang.StackOverflowError has occured
Please check line 58 of C:\Documents and Settings\Mark Guzdial\My
    Documents\funcplay.py
```

A function can get called so much that the memory set aside for tracking the functions (called the *stack*) runs out, called a *stack overflow.*



How do we stop?

- If we have only one character in the word, print it and STOP!

```
def downUp(word):
  if len(word)==1:
    print word
    return
  print word
  downUp(word[1:])
  print word
```

or

```
def downUp(word):
  print word
  if len(word)==1:
      return
  downUp(word[1:])
  print word
```

or

```
def downUp(word):
  print word
  if len(word)>1:
      downUp(word[1:])
      print word
```

That works

```
>>> downUp("Hello")
Hello
ello
llo
lo
o
lo
llo
ello
Hello
```

2. Let’s trace what happens

- >>> downUp("Hello")
  - **The len(word) is not 1, so we print the word**
- Hello
  - **Now we call downUp(“ello”)**
  - **Still not one character, so print it**
- ello
  - **Now we call downUp(“llo”)**
  - **Still not one character, so print it**
- llo

Still tracing

- downUp("lo")
- Still not one character, so print it

- lo
  - Now call downUp("o")
  - THAT'S ONE CHARACTER! PRINT IT AND RETURN!
- o

On the way back out

- **downUp(“lo”) now continues from its call to downUp(“o”), so it prints again and ends.**

- lo
  - **downUp(“llo”) now continues (back from downUp(“lo”))**
  - **It prints and ends.**
- llo
  - **downUp(“ello”) now continues.**
  - **It prints and ends.**
- ello
  - **Finally, the last line of the original downUp(“Hello”) can run.**
- Hello

3. Little elves

- Some of the concepts that are hard to understand:
  - **A function can be running multiple times and places in memory, with different input.**
  - **When one of these functions end, the rest still keep running.**
- A great way of understanding this is to use the metaphor of a function call (a *function invocation*) as an elf.
  - **(We'll use students in the class as elves.)**

Elf instructions:

1. Accept a word as input.
2. If your word has only one character in it, write it on the screen and you're done! Stop and sit down.
3. Write your word down on the "screen"
4. Hire another elf to do these same instructions and give the new elf your word *minus* the first character.
    1. **Wait until the elf you hired is done.**
5. Write your word down on the "screen" again.
6. You're done!

Exercises

- Try writing **upDown**

```
>>> upDown("Hello")
Hello
Hell
Hel
He
H
He
Hel
Hell
Hello
```

Why use functional programming and recursion?

- Can do a lot in very few lines.
- Very useful techniques for dealing with hard problems.
- *ANY* kind of loop (FOR, WHILE, and many others) can be implemented with recursion.
  - **It's the most flexible and powerful form of looping.**

Factorial -- the classic recursive function

```
def factorial( number ) :
  if number == 1 :
    return number
  else :
    return number * factorial( number - 1.0 )
```

Coming Attractions

- Friday:
  - Group project 2 - Sound Abstraction due 2:00 PM
  - bring to Lab to demo!
- Trick or Treat !
  - Early Warning - next midterm Friday 10/31