Data Mutation and Related Topics

CS 8: Introduction to Computer Science, Spring 2019
Lecture #6

Ziad Matni, Ph.D.
Dept. of Computer Science, UCSB

Administrative

- Hw03 – due next week
- Lab01 – due on Sunday by midnight (11:59 pm) on **Gradescope**!
- You can check old homework on GradeScope

Lecture Outline

- Print vs. Return
- The **range()** Function
- Mutability of Variables in Python
  – Caution: may cause temporary headaches! :{

Reassignment

- *Def*: change the value of a variable by assigning (using the = op.) again

Example:

```
>>> x = 9
>>> print(x + 4)
>>> x = 23                    # x is reassigned
```

etc…

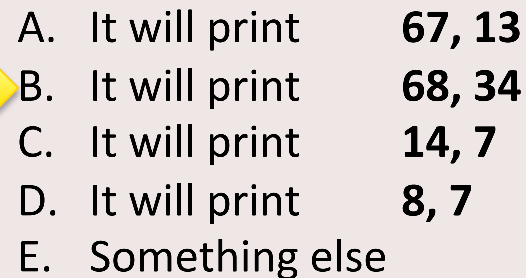
Mutability of Variables

- Consider this function:

```
def DoIt( a, b ):
    a = b + 1
    b = a/2
    print(a, ",", b)
```

What happens if I do this ***in IDLE***?

```
>>> x = 67
>>> y = 13
>>> DoIt( y, x)
```



Answers:

A. It will print **67, 13**
B. It will print **68, 34**
C. It will print **14, 7**
D. It will print **8, 7**
E. Something else

Mutability of Variables

- Consider this function:

```
def DoIt( a, b ):
    a = b + 1
    b = a/2
    print(a, ",", b)
```

What happens if I do this ***in IDLE***?

```
>>> a = 67
>>> b = 13
>>> DoIt( b, a )
>>> print(a, "," , b)
```

Answers:

A. Prints **68 , 34** then **68 , 34** on another line
B. Prints **68 , 34** then **67 , 13** on another line
C. Prints **14 , 7** then **14 , 7** on another line
D. Prints **14 , 7** then **67 , 13** on another line
E. Something else

*Why didn't the **DoIt()** function NOT change the value of the Python shell variables a, b ?*

Mutability of Variables

- Consider this function:

```
def DoIt( a, b ):
    a = b + 1
    b = a/2
    print(a, ",", b)
```

What happens if I do this ***in IDLE***?

```
>>> a = 67
>>> b = 13
>>> DoIt( b, a)
>>> print(a, "," , b)
```

*Why didn't the **DoIt()** function NOT change the value of the Python shell variables a, b ?*

These are treated as different a's and b's!
***Reassignment** within the function has NO EFFECT on the variables in the Python shell / rest of the Python program.*

Mutability of Variables

- Let's try another one:

```
def mutate( a ):
    a[0] = a[1] + 1
    a[1] = a[0]/2
    print(a[0], "," , a[1])
```

What happens if I do this ***in IDLE***?

```
>>> x = [ 67, 13 ]
>>> mutate(x)
>>> print(x)
```

Answer:

It prints:

```
[ 14 , 7 ]
[ 14 , 7 ]
```

Mutability of Variables

- Let's try another one:

```
def mutate( a ):
    a[0] = a[1] + 1
    a[1] = a[0]/2
    print(a[0], "," , a[1])
```

What happens if I do this ***in IDLE***?

```
>>> x = [ 67, 13 ]
>>> mutate(x)
>>> print(x)
```

Answer:

It prints:

```
[ 14 , 7 ]
[ 14 , 7 ]
```

The list WAS changed by the function!!

Mutable vs. Immutable data

Changeable types *vs.*	Unchangeable types	
`list`	`float`	`int`
`Turtle` (more on this later)	`str`	`bool`
`dictionary` (more on this later)	`tuple`	
Any user-defined object		

Lists are Mutable Data

For example, if the list **myL** is defined as follows:

```
myL = [ 1, 2, 3, 4]
```

and then I do this: `myL[3] = 42`

`myL` now becomes: `[1, 2, 3, 42]`

The **range()** Function

- Built-in function in Python provides a handy list
- Simplest use: `range(n)`
  - Creates a *something that looks like a list* with `n` items: `[0, 1, 2, …, n-1]`
- Example:

```
>>> print( list(range(5)) )
```

Will print out:

```
[0 , 1, 2, 3, 4]
```

The **range()** Function

- You can also do a **range()** with **start** & **stop** parameters.
- Example:

```
>>> print (list( range(5, 8) ) )
```

This will print out the list **[5, 6, 7]** (note it excludes 8)

- **Or** you can have **start**, **stop** *and* **step** parameters.
- Example:

```
>>> print (list( range(1, 11, 4) ) )
```

This will print out the list **[1, 5, 9]**

Will come in ***very*** handy when we learn about **loops**!

Reassignment vs. Data Mutation

If I do this:

```
myL = list(range(1, 5))    myL = [1, 2, 3, 4]
```

Then I do this:

```
myL = list(range(10, 13))  myL = [10, 11, 12]
```

This is a **REASSIGNMENT** of the variable **myL**

(I *completely changed* variable **myL**)

Reassignment vs. Data Mutation

But, if I do this (again):

```
myL = list(range(1, 5))
```

myL = [1, 2, 3, 4]

Then I do this:

```
myL[1] = 10
myL[2] = 11
```

mL = [1, 10, 11, 4]

This is ***changing the object*** that **myL** references!

It's NOT a reassignment of **myL**!

So What…?

- It matters because variables are really a *reference* to some value
- Note that if I do the following:

```
>>> myL = list(range(1,5))
>>> yourL = myL
>>> print (yourL[1])          # this prints 2
```

But Wait!...

- And now note
  that if I do **this**:

Explanation
- **myL** references [1,2,3,4]
- **yourL** references what **myL** references
- If something in **yourL** changes, then it is reflected in **myL** also!

```
>>> myL = list(range(1,5))
>>> yourL = myL
>>> yourL[1] = 100
>>> print (myL[1])        # prints 100, not 2!!!
```

One More Thing…

- Now note that if I do **this**:

```
>>> myL = list(range(1,5))
>>> yourL = myL
>>> myL = list(range(7, 10))
>>> myL[1] = 42
>>> print (yourL[1])        # prints 2, not 42!!!
```

Explanation

- **myL** references [1,2,3,4]
- **yourL** references what **myL** references
- I reassigned **myL** completely: this "detaches" **yourL** from **myL's** reference
- If I change something in **myL**, it's not reflected anymore on **yourL**

Summary of Findings…

- **Mutable** is a type of variable that can be changed (Lists are mutable)
- **Immutables** are the objects whose state *cannot* be changed once the object is created (Strings and numbers are immutable)

Example:

```
msg = “Hello”
msg = msg + “ World”
print(msg)    # Will print out “Hello World”
```

- On appending the variable `msg` with a string value, the following events occur:
  - The existing value of string `msg` is retrieved
- "World" is appended to the existing value of string `msg`
- The resultant value is then allocated to a new block of memory
- The `msg` object now points to the *newly created memory space (reassignment)*

Functions and Immutable Variables

- Let's say I have **x = 7** and **y = 9** and I want to swap their values, so that **x = 9** and **y = 7**
  - There's a classic algorithm for that…

```
tmp = x
x = y
y = tmp
```

- But, what if I want to do this through a function **swap(a,b)**

- **Can I do that?**
  - Let's see…

Swap Function: Will it Work or Not?

```
>>> def swap(a,b):
    temp = a
    a = b
    b = temp

>>> x = 5
>>> y = 10
>>> swap(x,y)
>>> print(x, y)
5 10
```

Explanation

- That's because I was dealing with **immutable objects** (ints)!!!!

Functions and ***Mutable*** Variables

- Let's say I have a list **myL = [2, 4, 6]** and I want to swap the values in position 1 and position 2
  - That is, I want **myL** to become **[2, *6*, *4*]**

- I want to do this through a function **swap(L, p1, p2)**

- **Can I do that?**
  - Let's see…

Swap Function: Will it Work or Not?

```
>>> def swap(L, p1, p2):
        temp = L[p1]
        L[p1] = L[p2]
        L[p2] = temp
>>> myL = [2, 4, 6]
>>> swap(myL, 1, 2)
>>> print(myL)
[2, 6, 4]
```

☺ *Yay!*

Explanation

- That's because I was dealing with **mutable objects** (a list)!!!!

Big Conclusion!

- You can change **the contents of lists** inside **functions** that take those lists as input.
  - Actually, lists or any *mutable object…*

- Those changes will be visible **everywhere**.
  - Immutable objects (like ints) are safe from these shinanigans, however…

Loops

- Sometimes we want to be able to **repeat** a part of the program a certain number of times *without* being repetitive
  - Called a "loop"

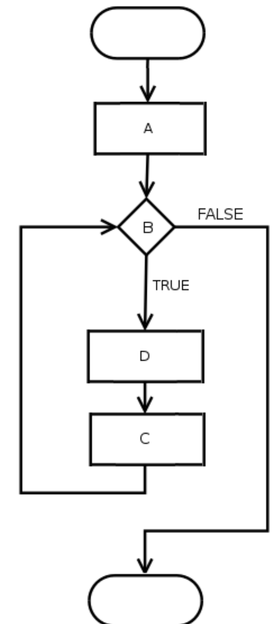
- So instead of saying:

```
print("hello")
print("hello")
print("hello")
```

I can say:

```
do the following 3 times:
    print("hello")
```

- A popular way to do this is with the `for` and the `while` commands.



Repetition with a `for` loop

- **`for`** *ref* **`in`** *some list*`:`
  `#` *block of instructions – ref refers to current object in list*
  `#` *note that the block is all indented*
  - **`for`**, **`in`**, **`:`** – mandatory parts
  - *ref* – a name for referring to objects in the list

- Example:

```
for numbers in (0, 1, 2, 3, 4, 5):
    print (numbers)
```

This will print out the numbers 1 thru 5 in sequence

Other Examples

```
for x in (9, 22, -77, 1):
    y = x + 10
    print (y)
```

WHAT DO YOU THINK THESE LOOPS PRINT OUT?

```
for y in ("Hello", "Mother", "Hello", "Father"):
    print (x, "!!")
```

```
n = 0
for item in ["UCSB Location", (34.4140, -119.8489)]:
    n = n + 1
    print(n, item)
```

Using `range` with `for` loops

- The `range()` built-in function provides a handy list
- Simplest use: `range(n)`
  - Creates a list with `n` items `[0, 1, 2, …n-1]`

- Example:

```
for numbers in range(6):
    print (numbers)
```

This will print out the numbers 1 thru 5 in sequence
(just like the last example)

YOUR TO-DOs

- ❑ Finish reading **Chapter 5**
- ❑ Finish **HW3** (due **TUESDAY**)
- ❑ Finish **Lab2** (turn it in by **Sunday**)

- ❑ Don't bike angry!

</LECTURE>