CS2110 Lecture 6 Feb. 5, 2021

- HW1 was due yesterday. Will be graded by Sunday night
- DS assignment 2 (due Tues.) and HW2 (due Thurs.) available
- Quiz 1 in class next Friday
- Last time
  - More on functions Ch6 - variables and parameters are local, stack frames, return vs print
  - Introduction to if/else, Ch7
- Today
  - A little more on functions, local variables, stack frames
  - More Ch 7 - if/elif/else, nested conditionals
  - 8.3 iteration with while

(last time)Ch 6: Variables and parameters are local

- A function's parameters are **local** variables. That is, they exist only during the execution of that function.
- Other variables that are assigned-to within a function body are also local (note: we'll see exception to this later)
- Top-level variables (not within **def**s) are called **global**

```
def foo(a, b):
    c = (a + b) * 2
    return(c)
```

```
>>> foo(3, 4)
    14
>>> a
    Error
>>> c
    Error
```

IMPORTANT!

(last time) Ch6: local/global variables

- Another way to say this is that each function has its own **scope** or **namespace**

```
>>> def f(x):
        y = 1
        x = x + y
        print(x)
        return x
```

Function f's variable x. A **local** variable that exists only within scope of definition of f

Two completely different variables!

```
>>> x = 3
>>> y = 2
>>> z = f(x)
4
>>> x
?    3
>>> y
?    2
>>>  z
?    4
```

"Top-level" or **global** variable x

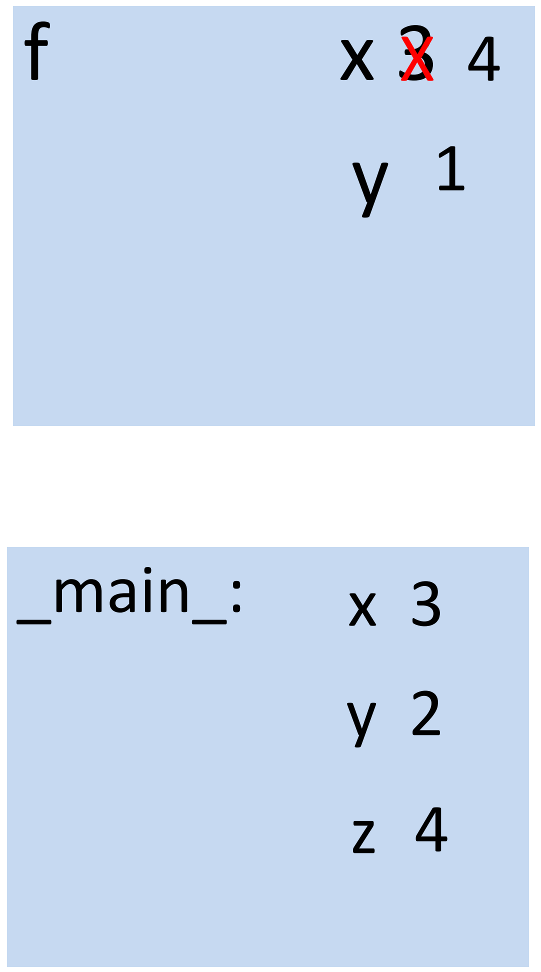
Hint: think of them as, e.g., x_f and x_{global}

(last time) Ch6: Stack frames

- At top level (the interpreter/shell) a “symbol table” keeps track of all variables (and values bound to them) defined at that level
- When a function is called, a new symbol table , or **stack frame**, is created to keep track of variables defined in the function and the values associated with those names. This includes (1) the function’s parameters and (2) any other variables assigned to within the function
  - When the function finishes, the stack frame goes away.
- *Note: this isn’t just one level, of course. As functions call other functions that call other functions, the stack (of stack frames) grows deeper …*

(last time) Ch6: Stack frames

```
>>> def f(x):
        y = 1
        x = x + y
        print(x)
        return x

>>> x = 3
>>> y = 2
>>> z = f(x)

4
>>> x

3
>>> y

2

>>> z

4
```



Demo - more complex example, #2, in lec5and6localvars.py.
Understand it by drawing stack frames!

Ch7: Condition execution – **if** (last time)

The most basic conditional statement is **if**

```
if (Boolean expression):
    …
    …       lines of code that execute when Boolean
    …       expression evaluates to True
…
… more lines of code. These execute whether or
… not Boolean expression evaluated to True
…
```

Ch7: **if-else**

```
if (Boolean expression):
    …
    …        code that executes when Boolean expr
    …        evaluates to True
else:
    …
    …        code that executes when Boolean not true
    …
…
… code that executes after if-else statement, whether
… Boolean expression was True or not
…
```

Ch7: nested conditionals

```
if (a < b):
    print('a is smaller')
else:
    if (a == b):
        print('a and b are equal')
    else:
        print('a is larger')
```

One statement

One statement

But there's an alternative in this particular situation …

Ch7: if-elif-else

(book calls this “chained conditionals)

```
if (a < b):
    print('a is smaller')
elif (a == b):
    print('a and b are equal')
else:
    print('a is larger')
```

One statement

demo: lec6ifs.py

Ch 8: Iteration

- Recall again, five key components of programming, independent of particular programming language.
  - Expressions
  - Variables and assignment
  - Functions
  - Conditional execution (if-else)
  - Iteration

We’ve covered the first four. Chapter 8 introduces the last one – iteration/repetition.

Reminder from Ch2: Reassignment and Updating Variables

Reassignment: Sometimes programs reassign variables to new values. E.g.

```
>>> a = 3
>>> b = a + 2
>>> a = 2
```

As mentioned in earlier lectures, this should not cause confusion. Remember, assignment statements are not algebraic constraints. They have an immediate, perhaps temporary, effect. When evaluating an assignment statement, think in two steps:

1) [Ignore left hand side for the moment] Evaluate expression on right hand side of '=' using current values of variables.

2) associate variable name of left hand side with value obtained in step 1

Thus, in b = a + 2, associates b *with the value* 5, *not with the variable* a

And then a = 2 changes the associate of a from 3 to 2, and does not affect b's association (with 5) at all

Reassignment and Updating Variables

Updating: Often programs *reassign* a variable to a new value in terms of its own current value.

```
>>> x = 3
>>> y = 4
>>> x = x + y
```

If you think via the two-step process, this is not mysterious/confusing.

1) [Ignore left hand side for the moment] Evaluate expression on right hand side of ‘=‘ using current values of variables.

2) associate variable name of left hand side with value obtained in 1

Thus, after the first two lines, x has value 3, y has value 4.

To evaluate the third line,

1) evaluate x + y, yielding 7

2) associate x with 7

It’s only mysterious/confusing when thought of as an algebraic equality/constraint.

Ch 8.3: Iteration – the **while** statement

- Ch 8 actually starts with **for** which is also introduced briefly in Ch4 (which we might not cover). We will start with more general iteration form: **while**
- Many computations involve repetition, doing the same (or nearly the same) things repeatedly (perhaps a few times, perhaps billions of times)
- You can already write a program to, say, print out the first 1000 integers

```
def printFirstThousand():
    print(1)
    print(2)
    …
    print(1000)
```

- But Python (and other languages) provide statements to conveniently describe and control repetitive computations.

Ch 8 – the while statement

The **while** statement provides a very general mechanism for describing repetitive tasks.

```
…
…   (B1: code before while)
…
while boolean expression:
    …
    …       (B2: code in while body)
    …
…
…   (B3: code after while)
…
```

What happens?

1. Execute B1 code
2. Evaluate boolean expr
3. If True, do
   3a. eval B2 code.
   3b. jump to step 2 again

   If False, ignore B2 code and simply continue with step 4
4. Execute B3 code

Ch 8: the **while** statement

Using **while**, how can we write concise printFirstThousand()?

and

sumFirstThousand()

sumFirstN(number)

demo: lec6while.py

Next time

- Read/do Ch 8
- Several more while examples
- Another iteration construct: **for**
- Some parts of Ch 9 on strings