

CS1 Lecture 5

Jan. 27, 2017

- HW1 due Sunday, 11:59pm.

Notes:

- read the note I sent yesterday (and added to homework spec): it is *not required* to include days/nights info in your output. To do so either requires some duplicate code (not good style) or returning multiple values from tripCost
- **Do not** write all the code at once (for Q1 and 2) before starting to test. Take tiny steps. Write a few lines ... test ... add a line or two ... test ... add another line ... test ... !!
- the math.ceil function can be helpful calculating the number of hotel nights
- HW2 available Monday morning
- First survey (ICON) due tomorrow.

Last time

Much of chapter 3

- Function calls
- Math functions (math module)
- Function composition
- Defining functions <- super important!
- Flow of execution
- Parameters and arguments

Today

Ch 3: The rest

Variables and parameters are local

Stack diagrams

Fruitful functions

- incl return vs. print

Why Functions?

Skip to first part of Ch 5: Conditionals

Floor division and modulus (useful for HW1)

Logical/Boolean expressions

Conditional execution – if/elif/else

Ch 3: 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 **defs**) are called **global**

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

```
>>> foo(3, 4)
```

```
14
```

```
>>> a
```

```
Error
```

```
>>> c
```

```
Error
```

IMPORTANT!

Ch3: 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

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

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

Two completely different variables!

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

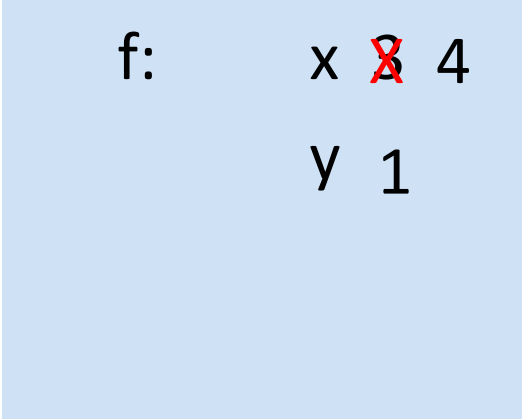
Ch3: Stack frames

- 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 ...*

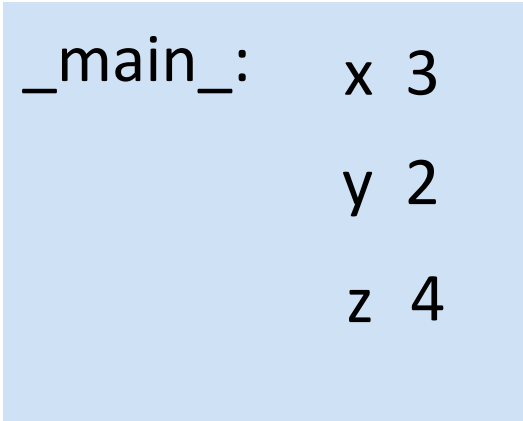
Ch3: Stack frames

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

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



f:	x	3	4
	y	1	



main:	x	3
	y	2
	z	4

Ch 3: “Fruitful” functions

Functions generally produce and **return** values:

```
def add1(number):  
    result = number + 1  
    return result
```

Often use a variable to “grab”/“catch” the returned value so you can use it in further computations:

```
>>> a = 3  
>>> b = add1(5)  
>>> c = (a + b) ** 2  
>>> c  
81
```


“Fruitful” functions

- Though not covered in chapter 3, functions can also return more than one value via, e.g.
 return val1, val2, val3

```
def minAndMax(a, b):  
    if (a<b):  
        return a, b  
    else:  
        return b, a
```

```
>>> small, big = minAndMax(23, 4)  
>>> small  
4  
>>> big  
23  
>>>
```

Ch 3: Fruitful functions. return vs print

print and **return** are very different (and some students never quite get this!)

```
def add1a(number):  
    result = number + 1  
    return result
```

```
def add1b(number):  
    result = number + 1  
    print(result)
```

Explore difference by using invoking them in shell in a couple of ways.

Ch 3: Why functions?

- Functions let you name a group of statements that make up a meaningful computation.
- Make programs easier to read and debug
 - E.g. debug functions one by one, assemble into whole program
- Make programs shorter (in lines of code) by eliminating duplication. If you start seeing same (or nearly the same) segments of code in various places in your program, consider defining a function and replacing the common segment with function calls.
- Well designed functions often useful in other programs you write.

Ch 3: A few words on debugging

- For beginning programmers, I'm not a big advocate of using "real" debuggers (the tools that come with IDEs like Wing and even IDLE).
- I'm a fan of people reading their code carefully and critically and
 - Using a lot of carefully placed print statements – print early, print often - see where things go wrong!
 - in HW1, if printed final results don't seem right, print out all your intermediate variables. Most people have variables for, e.g. veh 1 driving time, veh 1 gallons needed, veh 1 gas cost, veh 1 hotel nights needed, etc. – print them all!
- I am also a fan of "baby steps": as I said earlier, write a little bit of code, then test, then a little more, and test again, etc. Small changes to a so-far-working program makes errors much easier to find.
- More to say on this later ...

Skip to Ch 5: Conditional execution

- Recall list of key programming components: Expressions, variables and assignment, functions, if-then-else (conditional execution), iteration
- We've covered three of them! Can't easily compute a lot yet. Need the last two. Conditional execution today, iteration next week.

Ch5: Floor Division and Modulus

- But first, Ch5 has oddly placed section of two math ops (not really related to rest of Chapter!)
- Convenient right now, though, since useful for HW1
- `//`: “floor division operator. Divides two numbers and truncates (rounds down) to whole number

`14 // 3` yields 4

`8.1 // 2` yields 4.0

- `%`: modulus operator. Yields remainder

`11 % 3` yields 2

$((14 // 3) * 3) + (11 \% 3) == 14$ and in general (except for possible floating point issues), $((n // d) * d) + (n \% d) == n$

- `math.ceil(x)` yield nearest integer equal or greater than x

% can be
very useful
for, e.g.,
odd/even
tests

Ch5: Boolean expressions and Logical Operators

- Demonstrated in Lec 3.
- Boolean expressions: have True/False as value
E.g. $3 == (5 - x)$
- Boolean/relational operators:
<, >, >=, <=, !=
E.g. $(x + y) < 4$
- Logical operators
and, or, not
E.g. $(x < 5) \text{ or } (z == \text{“foo”})$

Understand these well. Critical for conditional execution, decision making in programs!

Ch5: Condition execution - if

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

...

Ch5: Condition execution – 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

...

Ch5: if-else

```
print('before if')
if (a < b):
    print('a is smaller')
else:
    print('a is not smaller')
print('after if')
print('goodbye')
```

NOTE: indentation semantically important. Has execution effect.

Next time

- Finish conditional execution part of Chapter 5
 - nested conditionals, if-elif-else
 - *Note*: the recursion section of Chapter 5 is very important but we will return to it later)
 - start iteration (the last of the five key programming components). Read Ch 7
-

新年快乐

Chúc mừng năm mới
2017

