CS1 Lecture 4 Jan. 25, 2017

- First homework due Sun., 11:59pm
  - Meet specs precisely.
    - Functions only.
    - Use a file editor! Don't type functions/long sections of code directly into Python interpreter. Keep the code you're working with in a .py file. Use "run" (or similar, depending on IDE) to "send" that code to interpreter. Test your code in interpreter by calling functions defined by your .py file.(Then modify code/fix errors in editor, send modified code back to interpreter, test again, etc. - editor < - > interpreter until correct.)
- First discussion section assignment (and first HW) can be quite hard for students completely new to programming. Read the book, practice, think … it will make sense if you work at it.
- First survey (on ICON) due soon, Jan. 28

Last time

Chapter 2

- Expressions
- Variables and assignment statements
- Python scripts
- Strings and expressions

Plus a very quick look at defining functions (from Ch 3)

(last time) Variables and Assignment Statements

Expressions yield values and we often want to give names to those values so we can use them in further calculations. A **variable** is a name associated with a value.

The **statement**

```
>>> x = 10
>>>
```

creates the variable x and associates it with value 10.

‘x = 10’ is a statement not an expression. It doesn’t produce a value. Instead, it associates x with the value 10 and subsequently x can be used in expressions!

```
>>> x + 3
13
```

(last time) Variables and Assignment Statements

In general, you write:

```
>>> var_name = expression
```

where var_name is a legal variable name (see book/Python reference) and expression is any expression

```
>>> zxy1213 = 14.3 + (3 * math.sin(math.pi/2))
>>> zxy1213
17.3
```

And since zxy1213 is a variable, thus a legal expression, we can write:

```
>>> sillyVarName = zxy1213 – 1.0
>>> sillyVarName
16.3
```

(last time) Variables and Assignment Statements

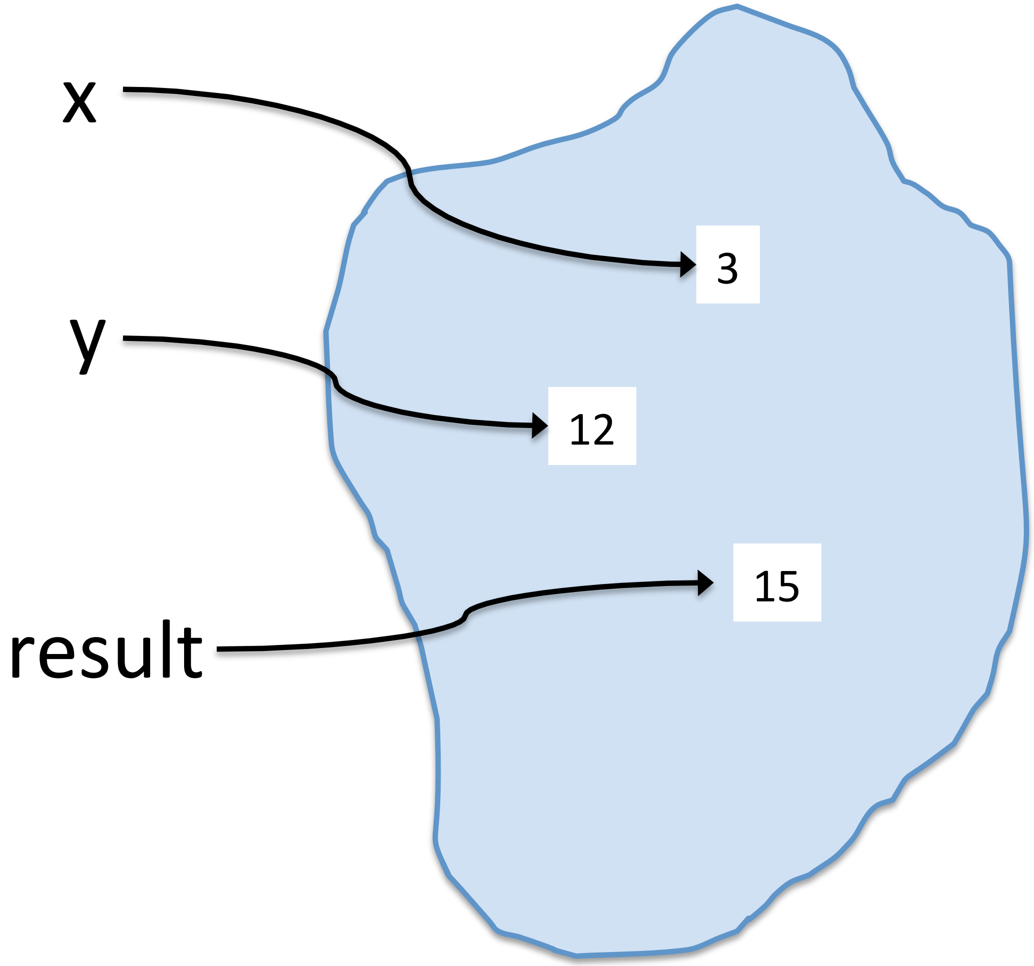
Only a single variable name can appear on to the left of an = sign (unlike for ==, the equality "question")

>>> x + 3 = 4   X (crashes, yields syntax error.)

>>> x + 3 == 4 OK (will return True or False, or give error if x has not been assigned a value)

(last time) Variables and Assignment Statements

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



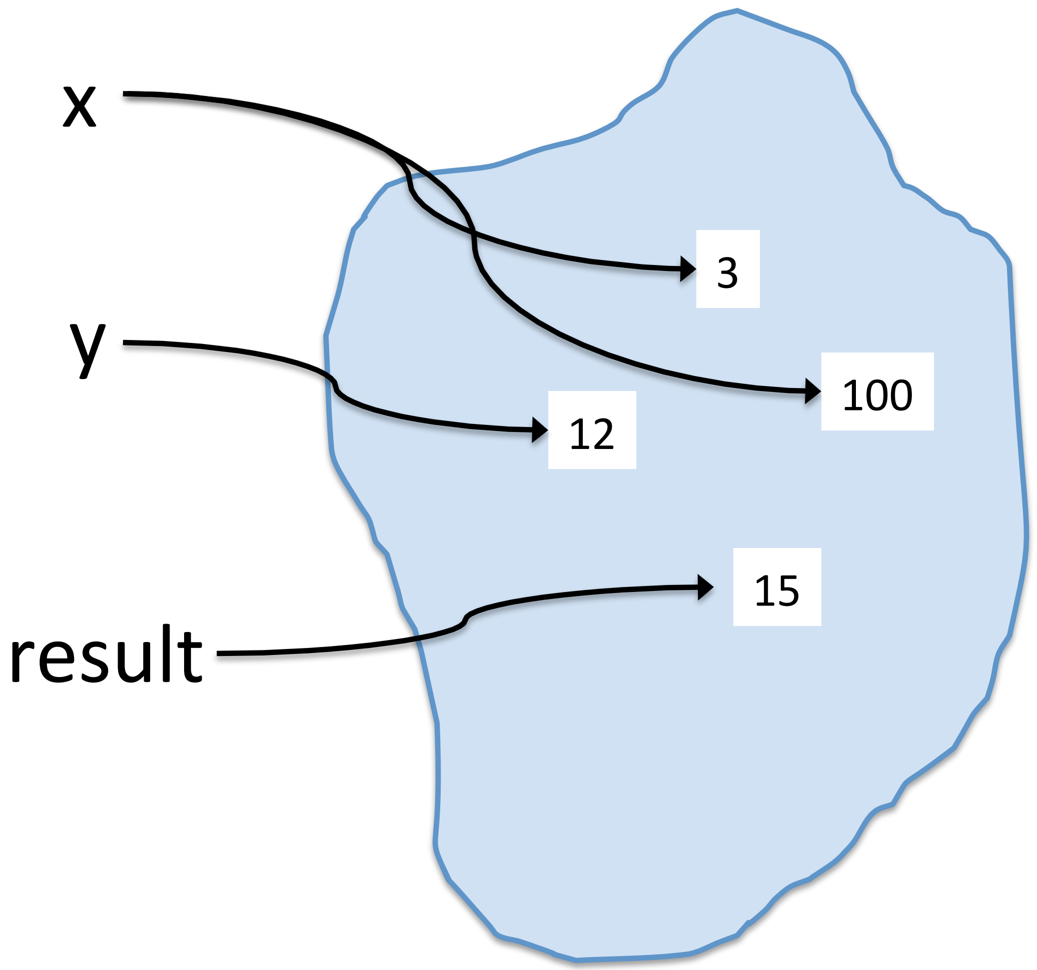
(last time) Variables and Assignment Statements

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

Rule (*very important to remember*):

1) Evaluate right hand side (ignore left for a moment!) yielding a value (no variable involved in result)
2) Associate variable name on left hand side with resulting value

```
>>> x = 100
>>> y
>>> ?
>>>result
>>> ?
```



y and result are not changed!

Don't think of assignments as constraints or lasting algebraic equalities. They make (perhaps temporary) associations between names and values.

Today

Chapter 3

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

(There's a bit more to Ch3 but we'll cover that on Friday.)

Ch 3: Function calls

In general:

```
>>> fn_name(param1, param2, …, paramN)
returned_value
```

We say a function takes N arguments and **returns** a computed value, returned_value

(*note: some functions, notably **print**, don't return anything other than special Python value None! Printing is **not** the same as returning a value. I will say more on this later…*)

```
>>> abs(-3) ← "function call"
3             ← value returned from function call
>>> min(17, 4)      ← function call
4                     ← value returned
```

Ch 3: Function calls

When arguments to function calls are expressions (very common), *evaluate expressions first*:

Presume variable a has value 23, b has value -3

```
>>> max(a, 14, b+12)
```

is evaluated by passing 23, 14, and 9 to the max function, yielding

```
23
```

In no sense are the variables a and b given to the function. Again, each argument expression is evaluated to produce a value, and those values are passed to the function.

Ch 3: Math functions

I've mentioned that Python has many libraries of useful functions (and sometimes special values). They're called **modules**. The functions in these modules are not part of basic Python. Usually, to get access, you use the **import** statement to load functions from a module into Python. We'll cover this in more detail later, but you should know about one key module: **math** (and you probably want to include "import math" in your HW1 Python file)

```
>>> sqrt(4)    error – not defined in basic Python
>>> import math
>>> math.sqrt(4)
2.0
>>> math.pi
3.141592653589793
>>> math.sin(math.pi/2)
1.0
```

Ch 3: Function composition

Do not be afraid to compose/nest function calls!

Just like in math, f(g(h(x),i(y)), j(z)) is legal/sensible.

```
>>> math.log(abs(math.sin(-math.pi / 2.0)))
```

Evaluate from inside out:

```
    math.sin(-math.pi / 2.0)    ->    -1
    abs(-1)                     ->    1
    math.log(1)                 ->    0.0
```

Ch 3: Defining New Functions

Super important to understand this! (You will do a *lot* of this in this course!)

Again, a function *call,* f(a,b,c) is an expression with a value, just like other Python expressions. Like in math, a function takes some "input" *arguments*, computes something, and *returns* an answer (though sometimes that answer is special Python value None)

def enables you to *define your own functions*

Ch 3: Defining New Functions

```
def functionName (param1, param2, …, paramN):
    ….
    …. (body of function, can be many lines,
    …. computes results value in terms of parameter
    …. variables bound to input values)
    ….
    return some_result_value
```

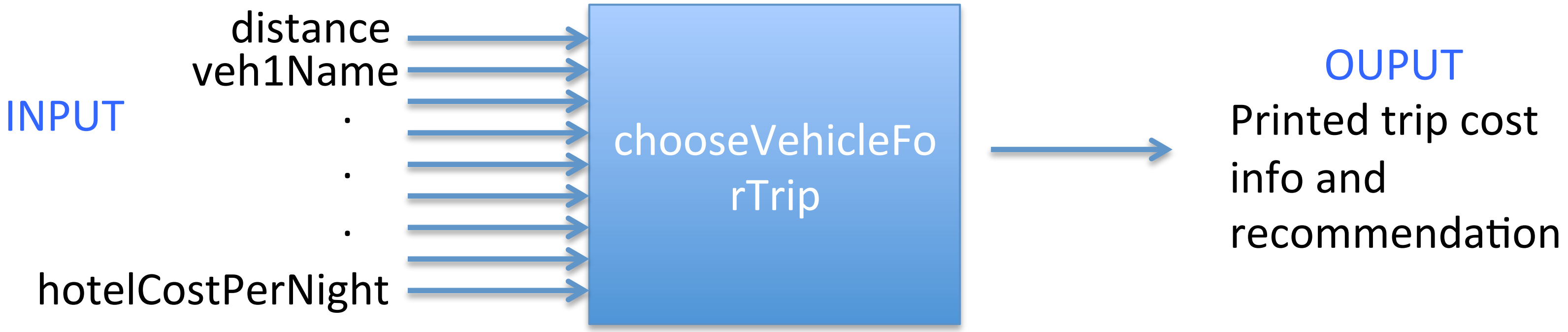
Make sure you understand:

- A primary use of functions is to define a general computation:
  - Compute square root of **any** (non-neg) number
  - Compute min of **any** pair of numbers, etc.
- If you don't include a **return** statement, function returns special value None
- Function body/computation specified in terms of **variables** (param1, …, paramN) rather than specific values. The variables will be bound to specific values when the function is actually called (not at function definition time!)

HW1: function bestVehicleFor

```
def chooseVehicleForTrip(distance,
                         veh1Name, veh1Speed, veh1MPG,
                         veh2Name, veh2Speed, veh2MPG,
                         gasCostPerGallon, hotelCostPerNight) :
     ....
     .... Lines of code that calculate, in terms of parameters, cost of trip
     .... using each car, and decides which is cheapest
     ....
     print( string summarizing veh 1 cost info )
     print( string summarizing veh 2 cost info )
     print( string giving vehicle recommendation )
     return
```

To the *user* of function, it's a "black box." User sends in values, sees printed answer!
(Note: actual *returned* value of function is None. The function doesn't returned anything; instead it prints information)



Ch3: Defining functions

```
def myMin (a,b):
    if (a < b):
        return a
    else:
        return b
-------------------------
>>> myMin(5,7)
>>> 5
```

(think of it like)

```
a, b = 5, 7
if (a < b):
    return a
else:
    return b
```

5

Ch 3: Defining functions

```
def myMin (a,b):
    if (a < b):
        return a
    else:
        return b
------------------------
>> x, y = 12, 10
>> myMin(x,y)
```

```
myMin(12,10)
```

```
a, b = 12, 10
if (a < b):
    return a
else:
    return b
```

```
>> 10
```

Ch 3: Defining New Functions

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

IMPORTANT

When executing a function call:

1) first, the function's parameter variables are bound to the *values* of the function call's arguments

2) then, body of the function is executed

```
>>> x = 3
>>> foo(x * x, x + 1, 3)
```

← foo will be executed with variable a bound to 9, b bound to 4, c bound to 3

foo "knows" nothing about x. x *isn't* passed in. 9, 4, and 3 are passed into foo.

Ch 3: Defining New Functions

Functions can have zero parameters:

```
def giveBack1():
    return(1)


>>> giveBack1()
1
```

You can type the function name at interpreter prompt:

```
>>> giveBack1
<function giveBack1 at 0x10367cb70>
>>> type(giveBack1)
<class 'function'>
```

Note: function name is just like any other variable. It is bound to a value; in this case that value is a function object. Don't try to

use that same name as a simple variable (e.g. print1 = 3). You'll "lose" your function. (And, you can even "break" Python …)

Ch 3: Control Flow

A program consists of a list of instructions, executed (except in certain situations) in top-to-bottom order

```
x = 3                     ←1
y = x + 13                ←2
z = foo(x, y, 3)          ← 3
newVal = math.sqrt(z)     ←6
result = x + newVal       ←7
print result              ←8
```

```
def foo(a, b, c):
 temp = a + (b * c)       ← 4
 temp = temp/3.0          ← 5
 return temp              ← 6
```

As part of 3, foo parameter

a is bound to 3 (not x!)

b is bound to 16 (not y!)

c is bound to 4

Next time

Ch 3: The rest

Variables and parameters are local

Stack diagrams

Fruitful functions

Why Functions?

Skip to first part of Ch 5: Conditionals

Logical/Boolean expressions

Conditional execution – if/elif/else

Read those sections!

side note: be careful with floats!

```
>>> x = 2 ** 64
>>> y = 2.0 ** 64
>>> x == y
True
>>> (x-1) == (y-1)
False
>>> (2.0 **64) == (2.0**64) + 1
True
>>> (2.0**53) + 1 + 1 == (2.0**53) + 2
False
>>> 2.2 * 3 == 6.6
>>> False
```

Floating point representations are often not exact. It probably won't be an issue in this class, but keep in mind for the future. Be careful when comparing floating point numbers for equality with each other or with 0.0!

Instead of

x == 0.0

use comparisons like

x <= abs(epsilon)

for some suitably small value epsilon