CS2110 Lecture 4 Feb. 1, 2021

- DS Assignment 1 due tomorrow afternoon
- HW 1 due Thursday
  - Meet specifications precisely.
    - Functions only (except “import math” also okay)
    - NO user interaction – do not use ‘input’ function.
    - Function names and parameter names must match specification *exactly*, including case
    - 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 Module” (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.
    - Q3 and Q4 are important. It is critical for you to learn skill of testing your code thoroughly on carefully considered set of cases that attempt to cover all legal input situations.
  - First homework assignment can be difficult for students completely new to programming. Read the book, practice, think … it will make sense if you work at it.

Last time

Chapter 2

- Expressions
- Variables and assignment statements
- Strings and expressions

(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

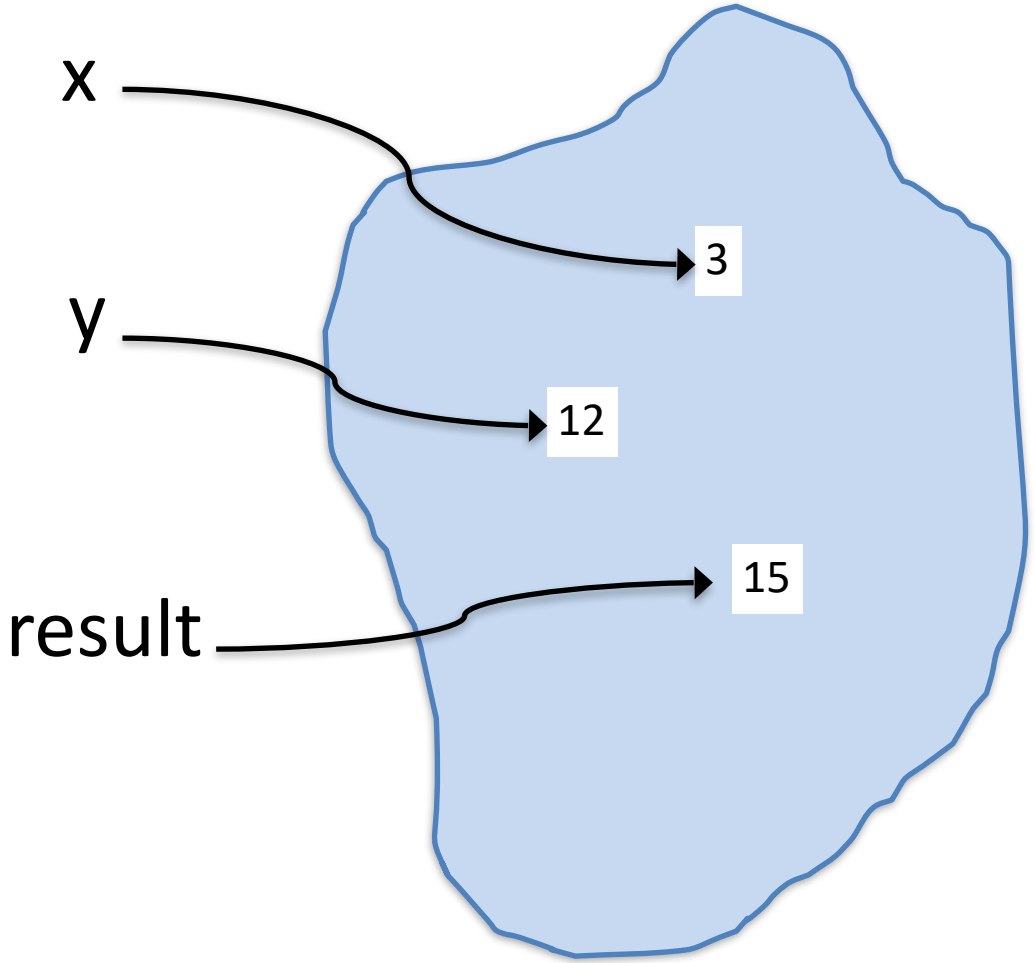
Only a single variable name can appear on to the left of an = sign (unlike for ==, the equality “question”)

>>> 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)

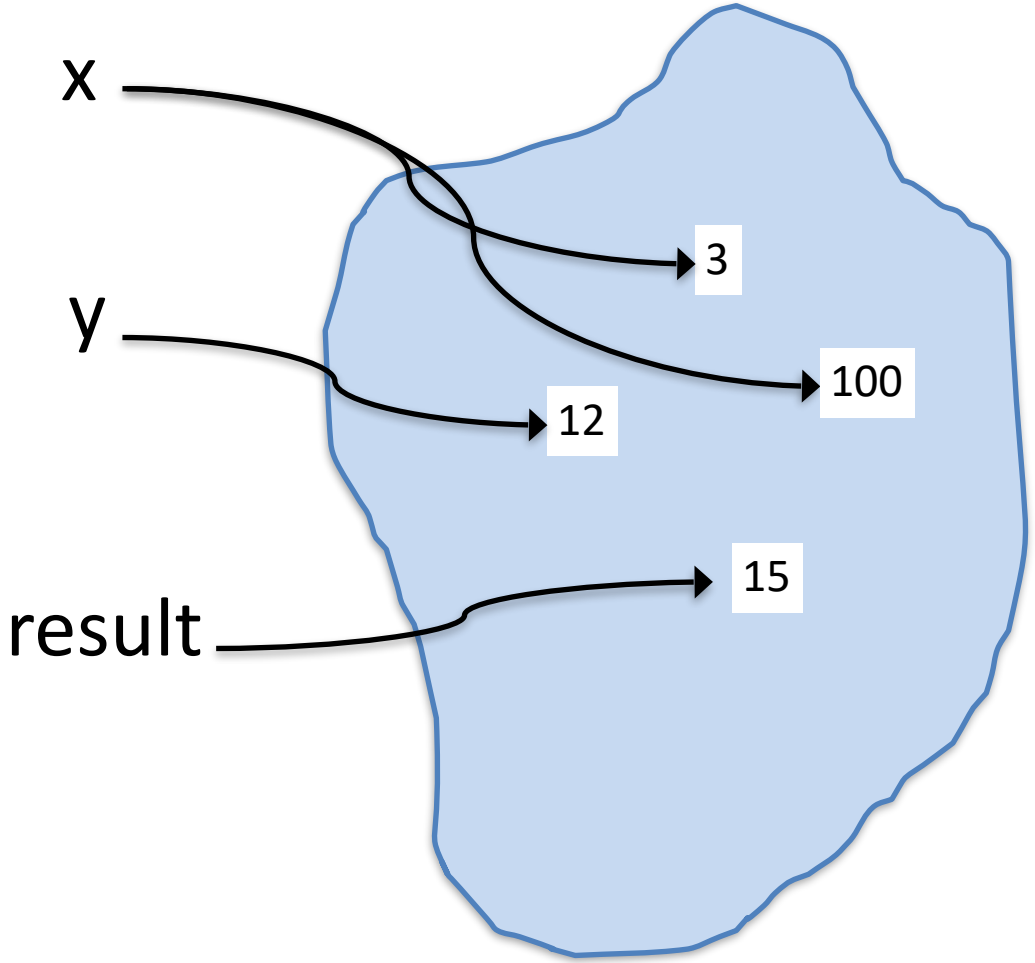
(last time) Variables and Assignment Statements

```
>>> 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

- Ch 5.3: math functions from the math module
- Chapter 6
  - Function calls
  - Composition
  - Defining functions <- super important!
  - Flow of execution
  - Parameters and arguments
- HW1 details/hints

Ch 5.3: 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 6: Function calls

In general:

>>> fn_name(arg1, arg2, …, argN)

returned_value

We say a function takes N argument values 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 6: 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 6: Function composition

Get used to and do not be afraid to compose, or nest, function calls!

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

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

Evaluate from inside out:

```
-math.pi / 2.0          ->   -1.5707963267948966
math.sin(-1.5707963267948966)  ->  -1.0
abs(-1.0)                      ->   1.0
math.log(1.0)                  ->   0.0
```

Ch 6: 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 6: Defining New Functions

```
def functionName (param1, param2, …, paramN):
    ….
    …. (body of function, can be many lines,
    …. computes result value in terms of parameter
    …. variables bound to input values)
    ….
    return 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.
  - Convert any temperature in Celsius to temperature in Fahrenheit
- If you don't include a **return** statement, function returns special value None
- Function body/computation specified in terms of parameter **variables** (param1, …, paramN). The parameter variables will be bound to argument values when the function is called (not at function definition time)

Ch6: Defining functions

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

```
>>> myMin(5,7)
5
```

Super important: Parameter variables a and b are only defined *during the execution of myMin*

```
>>> a
```

ERROR: a not defined

think of calling myMin(5,7) as:

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

5

Ch 6: 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 6: 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) second, the 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.

”Receiving” and saving and using the result of a function call

- You can directly use the value returned by a function

```
>>> print(3 + foo(2,3,4))
```

- Often, though, you want to save it in a variable for use in other parts of your program

```
>>> fooResult = foo(2,3,4)
…..
>>> print(17 + fooResult)
```

It is not usually effective to have a function call by itself as a line of your code

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

```
>>> result = 0
>>> foo(2,3,4)
10
>>> print(result + 1)
1
>>> a
Error
>>> fooResult
Error
```

```
>>> result = 0
>>> fooResult = foo(2,3,4)
>>> print(fooResult + 1)
11
```

Super important: parameter variables a, b, c and foo's "local" variable temp and fooResult have no value except during the computation of foo(2,3,4). They are not accessible outside of foo.

Maybe the prior version is easy to understand but what about this one?

```
>>> def foo(a, b, c):
        temp = a * b
        result = temp + c
        return result
>>> result = 0
>>> foo(2,3,4)
10
>>> print(result + 1)
1
```

```
>>> result = 0
>>> result = foo(2,3,4)
>>> print(result + 1)
11
```

We will talk more about this next time BUT it is *super* important to realize that the **result** variable inside **foo** is a different variable and unconnected to the variable **result** used outside foo at the command prompt. When foo(2,3,4) is executing, a separate result variable is created temporarily. Think of it as $\mathbf{result}_{\mathbf{foo}}$. When foo completes its computation, it returns the $\mathbf{result}_{\mathbf{foo}}$**'s** value. We can think of the other result variable as $\mathbf{result}_{\mathbf{global}}$ **. foo** modifies its local result variable ($\mathbf{result}_{\mathbf{foo}}$) not the global one, so if we don't save the returned value, the call foo(2,3,4) has no effect computation (other than using some computer time)

Functions can return multiple values

```
def minAndMax(a, b, c):
    minVal = min(a,b,c)
    maxVal = max(a,b,c)
    return minVal, maxVal
```

minAndMax will return two values. To “receive” and save them, use parallel assignment:

```
>>> minResult, maxResult = minAndMax(17, 3, 5)
>>> minResult
3
>>> maxResult
5
```

USE THIS STYLE WHEN YOU CALL tripCostData in HW1

HW1: function tripCostData

def tripCostData (distanceKM, vehSpeedMPS, vehKPL, gasCostPerLiter, hotelCostPerNight, breakfastCostPerDay, lunchCostPerDay, dinnerCostPerDay) :

    ….

    …. Lines of code that calculate, in terms of parameters, cost of trip

    ….

    return tripCost, gasCost, foodCost, numLunches, numHotelNights

To the *user* of function, it's a "black box." User sends in values, sees printed answer!

HW1: function tripCostInfo

```
def tripCostInfo(distanceKM, vehSpeedMPS, vehKPL, gasCostPerLiter , hotelCostPerNight,
                 breakfastCostPerDay, lunchCostPerDay, dinnerCostPerDay) :
    totalCost = 0
    # compute length of trip in hours
    hours = …
    # compute liters of gas needed
    litersNeeded = …
    # compute gas cost
    gasCost = …

    # compute number of nights (hotel stays) needed
    nights = …                    This should be an integer!
    # perhaps compute a number (0.0->1) representing fraction of final day
    lastDayFraction = …

    # compute breakfast, lunch, and dinner costs
    breakfastCost = …
    # (lunchCost might involve more than one line and an if statement)
    dinnerCost = …

    # sum costs
    totalCost = …

    # return results
    return totalCost, gasCost, …
```

HW1: compareVehiclesForTrip

```
def compareVehiclesForTrip(…) :
    # assign some new variables values based on converting units in input
    …

    # make two calls to tripCostInfo
    veh1Cost, gasCost1, veh1Nights, veh1Lunches, veh1FoodCost = tripCostData(...)
    veh2Cost, gasCost2, veh2Nights, veh2Lunches, veh2FoodCost = tripCostData(…)

    # compare results and print information and recommendation
    if (…):
        print(…)
        ..
    else:
        print(…)
        ..

    return # nothing needs to be returned
```

HW1 Q3 & Q4 - test functions?

- "Write function testQ1() that makes at least five calls to tripCostData(...) with different arguments."
- Students email me – "what's this, dude????"
- You should get in the habit of writing functions that you use to test your code as you are debugging it.
  - When you wrote tripCostData, you probably (I hope!) test it by trying
    >>> tripCostData(....)
    multiple times with different values in the interpreter. But that can get tedious, especially for larger, more complicated program.

    It can be better to "package up" the tests, so you can easily run them.
  -
    def testQ1():
      tripCostData(....)
      tripCostData(... different arguments...)
      tripCostData(… different arguments...)

    NOTE: Something is not right here! What? tripCostData doesn't print anything so testQ1() will run but show you nothing
    FIX? *Use print(tripCostData(…))*

    Then, every time you change tripCostData, you just run testTripCost and examine the output version what it should be.
- It's important that you choose the arguments of your tests carefully! Think about what sets of argument test the variety of situations that the function should handle
- Similarly, for testP2:
  - def testP2():
      compareVehiclesForTrip(…)
      compareVehiclesForTrip(…)
      compareVehiclesForTrip(…)

    NOTE: unlike for testP1(), no print(…) needed around compareVehiclesForTrip() because compareVehiclesForTrip already prints information (rather than returning it)

HW1 Q3 & Q4 - test functions

- It's important that you choose the arguments of your tests carefully! Think about what sets of argument test the variety of situations that the function should handle
- def testQ1()

  print(tripCostData(100.0, 100.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0))

  Print(tripCostData(200.0, 100.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0))

  …
- from random import random

```
def testP1()
  print(tripCostData(random(),random(),random(),random(),random(),random(), random(), random()))
  print(tripCostData(random(),random(),random(),random(),random(),random(), random(), random()))
  print(tripCostData(random(),random(),random(),random(),random(),random(), random(), random()))
  print(tripCostData(random(),random(),random(),random(),random(),random(), random(), random()))
  print(tripCostData(random(),random(),random(),random(),random(),random(), random(), random()))
```

Not effective for testing!!

- for the random case, would you know the expected results? It's not a good test if you don't know expected results to compare to actual results

Next time

Ch 6: More on functions

Variables and parameters are local

print vs return

- super important to know the difference and to pay attention to when homework assignments and exam specify whether to print or return things. E.g. Do NOT *print* values when we say your function should *return* something.

Ch 7: Conditional execution

Logical/Boolean expressions

Conditional execution - if/elif/else

HW1 Help

Use of math.ceil

Convenient printing using String format