

CS1 Lecture 3

Jan. 23, 2017

- Office hours for me and for TAs have been posted
- First homework available, due Sun., 11:59pm.
- Discussion sections tomorrow and Wednesday.
 - Discussion section assignments must be submitted at the end of the discussion section. You need to attend to be able to get the points for the discussion section assignment.
 - Be on time; in some sections you will be paired with a partner at the beginning.
- Make sure to complete the research-study survey that was posted on ICON

Last time

- Course overview
- Chapter 1 of text
- IC Arrest blotter demo program
- Discussion of graph problems posed at end of first lecture

(from last time) Ch 1: Programs

Key components of programming, independent of particular programming language.

- Expressions
- Variables and assignment
- if-then-else (decision making/branching)
- Iteration
- Functions

Essentially all programming languages are built around these components. Once you understand how to describe computations using them, you can program. **Learn these basics well!** Changing programming languages is usually just a matter of looking up details of how to “say” a particular standard thing in the different language.

Chapter 1 of text says it differently: input, output, math, conditional execution, repetition. I/O is important but, to me, not key or interesting at the beginning. “math” is too vague. But assignment and variables, in the programming sense, don’t fit well under everyday use of word ‘math.’ And while functions can be thought of as math, in programming functions (often called procedures when used slightly differently) play a very important role beyond just “being part of math.” They help organize programs into understandable components, etc.

(from last time) Ch 1: expressions, arithmetic, types

- You can use the Python interpreter like a calculator, typing in many different mathematical (and other) expressions and seeing immediate results
 - `print("hello")`
 - `3 + 2`
 - `2**128`
 - `5/2+1`
- Expressions yield **values**. Every value has a **type**.
 - 3's type is **int** (for integer)
 - 3.5's type is **float** (for floating point number)
 - (What about 3.0? 3.0's type is float. It is not the exact same thing as 3 in Python but (fortunately) it *is* "equal" to 3. More on this later.)
 - "hello"'s type is **string**
 - You can ask Python for a value's type
 - `>>> type(3.5)`

Today

Chapter 2

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

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

Expressions

Generally, an **expression** is a combination of literals (things like numbers, strings, Booleans) and operators (+, -, ...) that, when evaluated by Python, yield a **value**

- mathematical expressions
 - yield numbers, objects of type **int** or **float**)
 $3 * (200 / 1.5)$
 $\text{abs}(-4) + 2$
- logical expressions
 - yield **True** or **False**, objects of type **bool**

Logical expressions

Some relational/logical operators (see Ch 5):

>, <, ==, and, or, not

Example expressions:

```
>>> (1 < 3) and (0 > 2)
```

```
>>> False
```

```
>>> abs(-1) == 2
```

```
>>> False
```

```
>>> (5 == (2 + 3)) or True
```

```
>>> True
```

Note: == is not a statement of equality! It's a question
– are the two sides equal? True or False question!

Order of operations

The textbook has a section on order of operations, so that you can figure out how to calculate the value of something like:

```
>>> 3 + 1 or 3 + 2 ** 2 + -14 / 2.0 == 0
```

```
>>> ???
```

This has a legal value but I wouldn't be able to tell you without looking things up. I **always** parenthesize fully to make expression clear.

E.g. $((3 * x) + (4.2 * (z ** 1.2))) - y$

Code should be written so that you and others can readily read and understand it!

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 have/produce a value. But it associates x with the value 10 and subsequently x can be used in expressions!

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

```
>>> 13
```

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
```

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.)

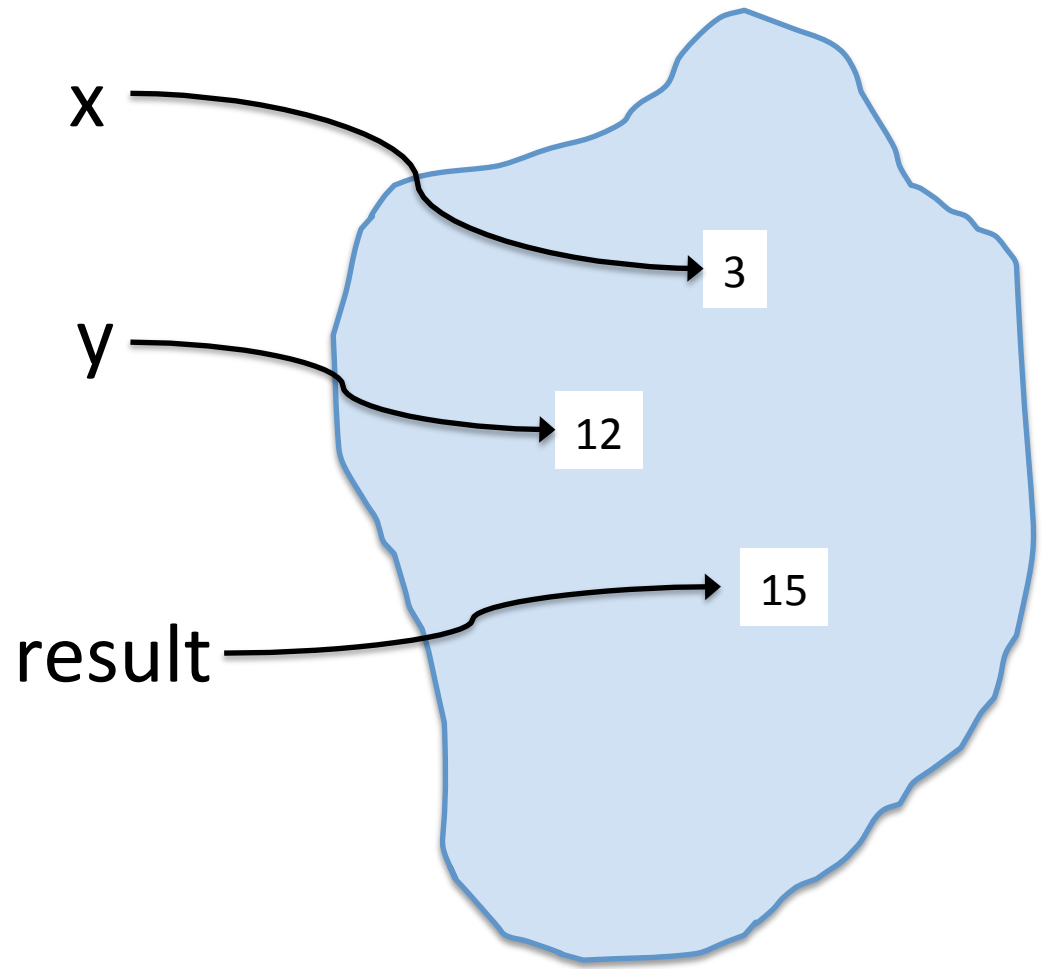
>>> x + 3 == 4 OK (will return True or False, or give error if x has not been assigned a value)

Variables and Assignment Statements

```
>>> x = 3
```

```
>>> y = 4 * x
```

```
>>> result = x + y
```



One of the most important things to know all semester

To process an assignment statement

`x = y * 32.1 ** foo(...) – math.sin(bar(...))) / ...`

- 1) **Evaluate right hand side** (ignore left for a moment!)
yielding a value (**no variable** involved in result – it's a number or a string or a boolean value or ...)
- 2) **Associate variable name on left hand side with resulting value**

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

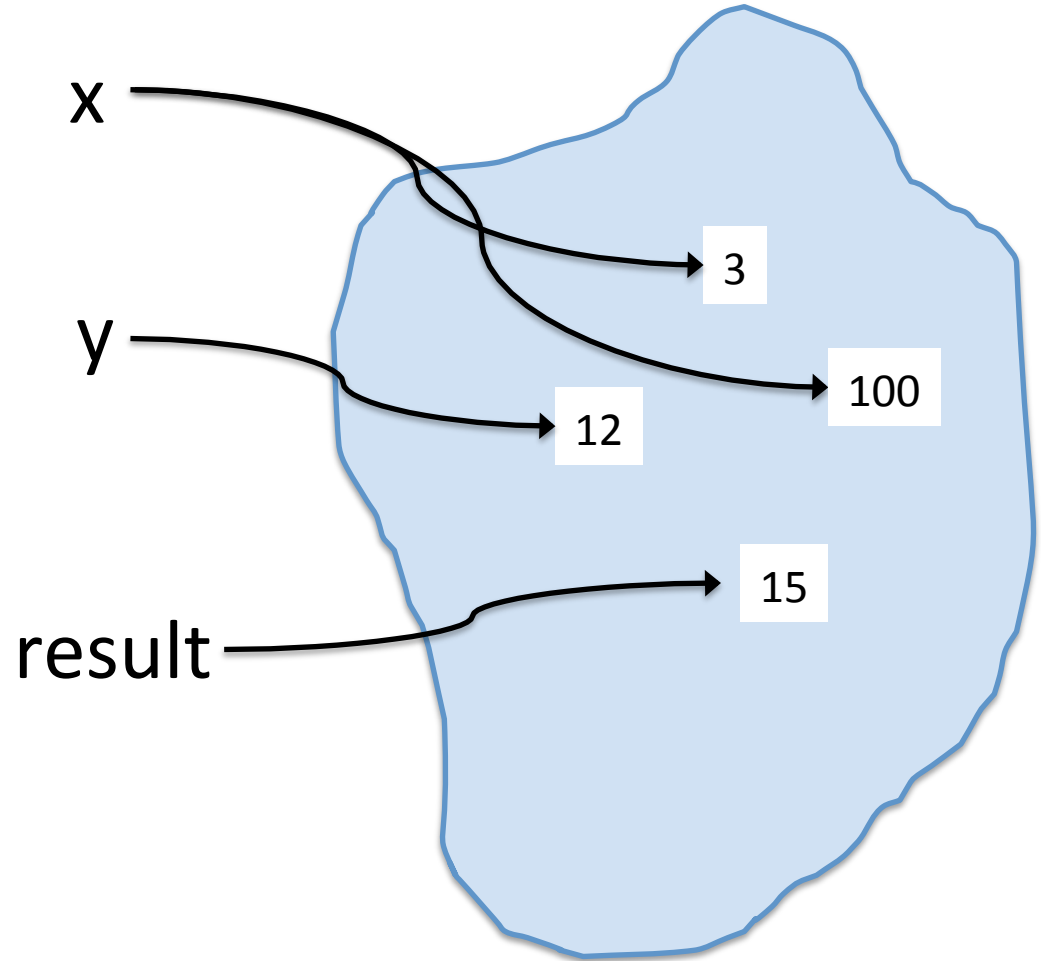
```
>>> x = 100
```

```
>>> 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.

Python scripts

We've been using Python interpreter in interactive mode.

Files of Python code are called **scripts**.

You can execute scripts directly from shell/command window or using IDLE or other IDE.

Demo.

When running scripts, expression (generally) don't produce output. Need print or other operations to see results.

Ch2's section on strings

There's a whole chapter (8) on strings but we want to use them much sooner (for one thing, we need them to print nice output!)

- Use quotes (single, double, even triple!) to delimit:
 `"abc" == 'abc'`
- But `"abc'` is not a legal string.
- The quotes at the ends are *not* part of the string. `"abc123"` has 6 characters in it, not 8.
- Strings *can* contain quote characters E.g. `"abc'def"` is a 7 character string containing a, b, c, d, e, f, and a single-quote character
- Strings in Python are powerful and can get complicated. Can represent full Unicode, chars from many languages, ...

Ch2's section on strings

- Again, we'll talk more about strings later but for now you should at least know you can use + operator on them

```
>>> name = "Jim"
```

```
>>> sentence = "Hi " + name + "!"
```

```
>>> sentence
```

```
>>> "Hi Jim!"
```

```
>>> print(sentence)
```

```
Hi Jim!
```

```
>>>
```

Very quick function (**def**) overview

- From Ch 3, more Wed. and Fri
- A function *call*, $f(a,b,c)$ is an expression with a value, just like other Python expressions. Like in math, a function receives some “input” *arguments*, computes something, and (usually) *returns* an answer.
 - `min(3,4)` returns 3
 - you can use the value a function returns:

```
>>> result = min(3,4)  
>>> z = result + 1
```
- **def** enables you to define your own functions

Functions

def functionName (list of formal parameters):

....

.... (body of function, can be many lines)

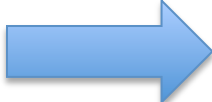
....

Note: indentation matters in Python. Is not optional – has meaning. Python is unusual in that way – indentation is not meaningful in most programming languages.

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

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
```

Next time

- More on variables and assignment
- Ch 3: Functions
- For next time, read Ch3