

# An Introduction to: Programming with Python

Variables, types, statements, functions



# Last Time

- Brief History of Computing
- Due:
  - PA1 (Tuesday Night)
  - PS1 (now!)

# Reminders

- ▣ Lab Today
- ▣ My OH are today
- ▣ OLI Iteration Module Tonight
- ▣ PS2 due Friday, July 5<sup>th</sup> at 9:00AM**

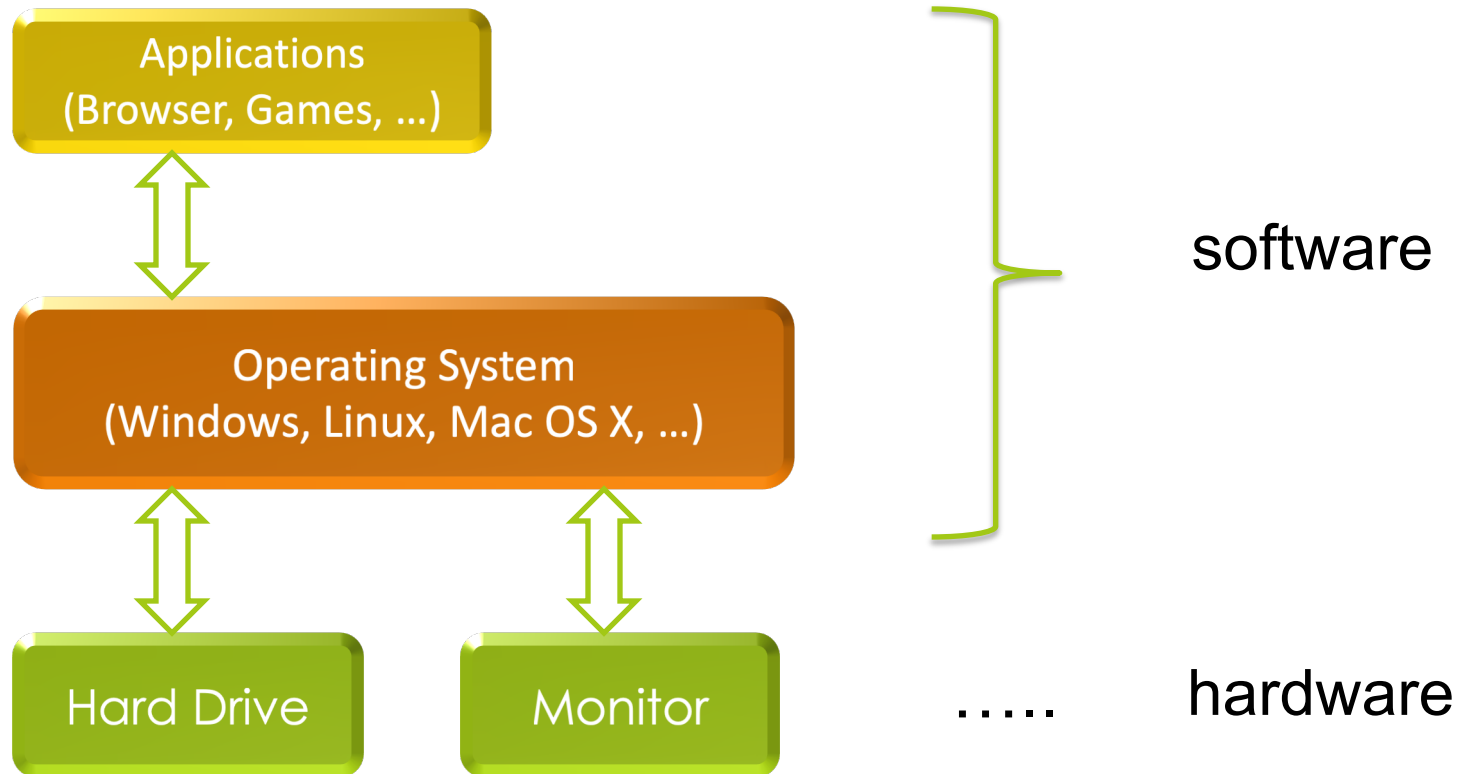
# Questions

- How was submission process:
  - Gradescope – PA/PS/Lab
- How was the OLI module?

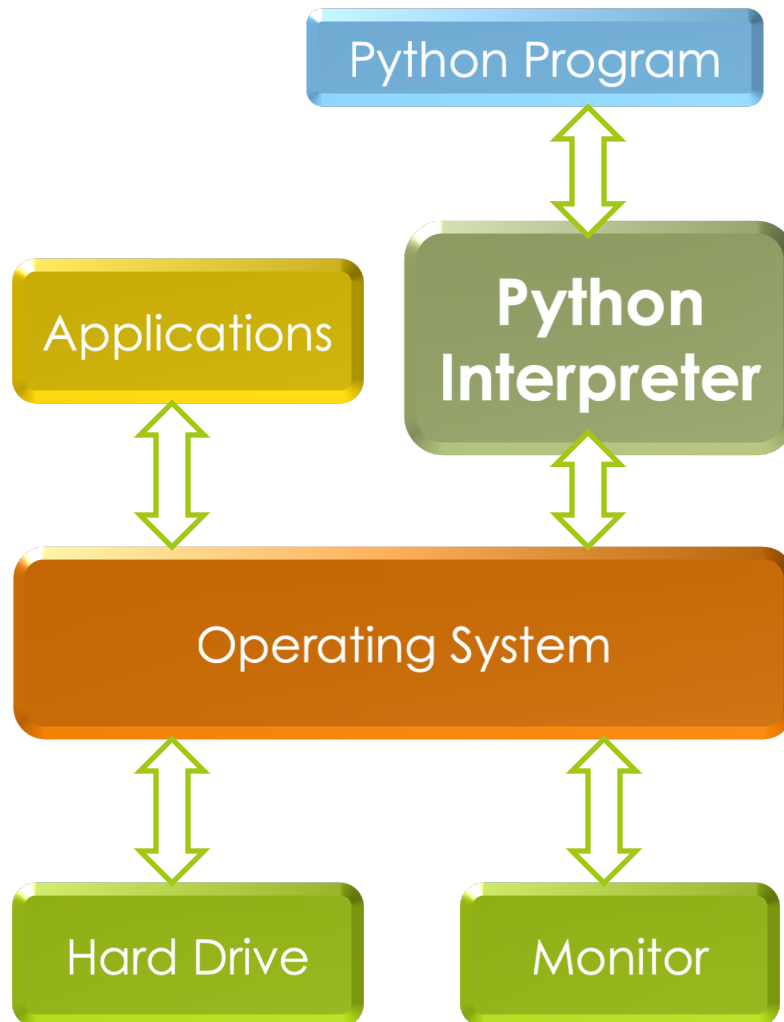
# Today's Lecture

- ▣ Introduction to Python
- ▣ Mechanics
- ▣ Some Specifics:
  - ▣ Programming Languages
  - ▣ Basic datatypes
  - ▣ Variables
  - ▣ Functions

# Hardware versus Software



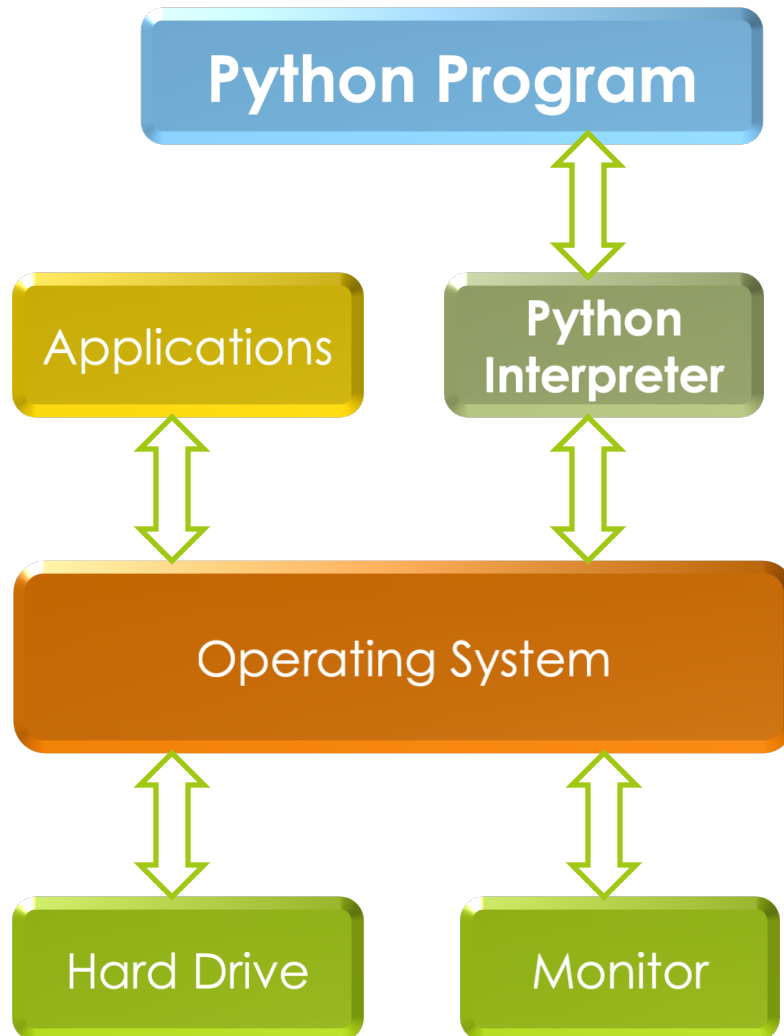
# Execution of Python Programs



When you write a program in Python, Java etc. It does not run directly on the OS.

Another program called an interpreter or virtual machine takes it and runs it for you translating your commands into the language of the OS.

# Execution of Python Programs



We will write Python programs that are executed by the Python Interpreter.

A Python Interpreter for your OS already exists. You can use the one on lab machines, install one for your laptop, or use one remotely  
...



# Using a Python Interpreter

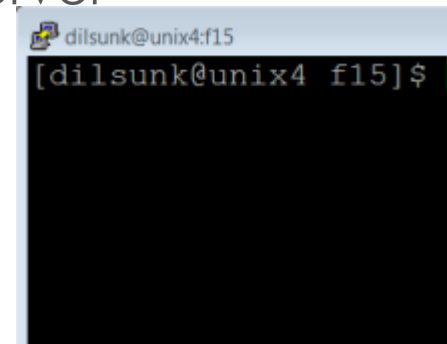
There are two ways to interact with a Python interpreter:

1. Tell it to execute a program that is saved in a file with a `.py` extension
2. Interact with it in a program called a shell

You will interact with Python in both ways.

# A Short Introduction to Python

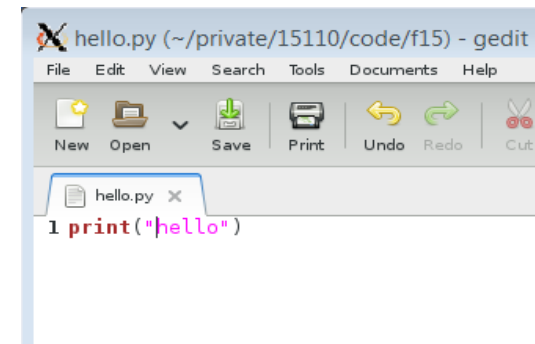
- Starting the Python interpreter either using a Unix Server at CMU or on your own computer
  - See the Resources page for specific instructions



```
dilsunk@unix4:f15  
[dilsunk@unix4 f15]$
```

- Creating .py files with a text editor

- Files with the .py extension can be created by any editor but needs a Python interpreter to be read.
- We have chosen editor editor for the course but you may use an editor of your own choice if you feel comfortable.
- Why IDE? Why Not?



```
hello.py (~/.private/15110/code/f15) - gedit  
File Edit View Search Tools Documents Help  
New Open Save Print Undo Redo Cut  
hello.py x  
1 print("hello")
```

# Programming Languages

A programming “language” is a  
**formal** notation

# Not a *natural* language

## Recipe

Toast and cereal

Toast the bread. Butter it. Put some cereal in a bowl. Add miki.

---

- ❑ Interpreted by a person
- ❑ Unclear? Can be figured out (What kind of bread? Butter? What kind of milk?)
- ❑ Typos? Can be figured out (“miki” means “milk”)

## Computer program

```
for i in range(5):  
    pritrn(whatever I want)
```

---

- ❑ Interpreted by a machine
- ❑ ...for a human (“somebody wants to print something”)
- ❑ Unclear? Not a program (“whatever I want”???)
- ❑ Typos? Program errors (“pritrn”???)

A programming “language” is a  
**formal notation**  
for **generalized** problem solving

# Programs should be *general*

## SWISS CHEESE & WHITE WINE SAUCE

1/4 c. butter  
4 tbsp. flour  
2 c. milk  
1 c. Swiss cheese  
1/2 c. white wine  
Salt & pepper

Make a roux, heat the milk, and when the roux is cooked, add some of the warm milk. Break or grate the cheese and stir it into the sauce until it is melted. Now add the rest of the milk and wine. Season with salt and pepper. Makes 2 cups.

Specific: “output” is two cups of sauce.

## Program

```
def force(mass, accel) :  
    return mass*accel
```

General: output is force for **any** combination of mass and acceleration.

# Python

- ▣ Python is one of *many* programming languages.
- ▣ 2 widely used versions. We will use Python 3.
- ▣ Running Python on the command line:

```
$ python3
```

or

```
$ python3 -i filename.py
```



# Command Line Interfaces

Be aware of the difference between  
“talking to the shell” and “talking to Python”

Talking to  
the shell

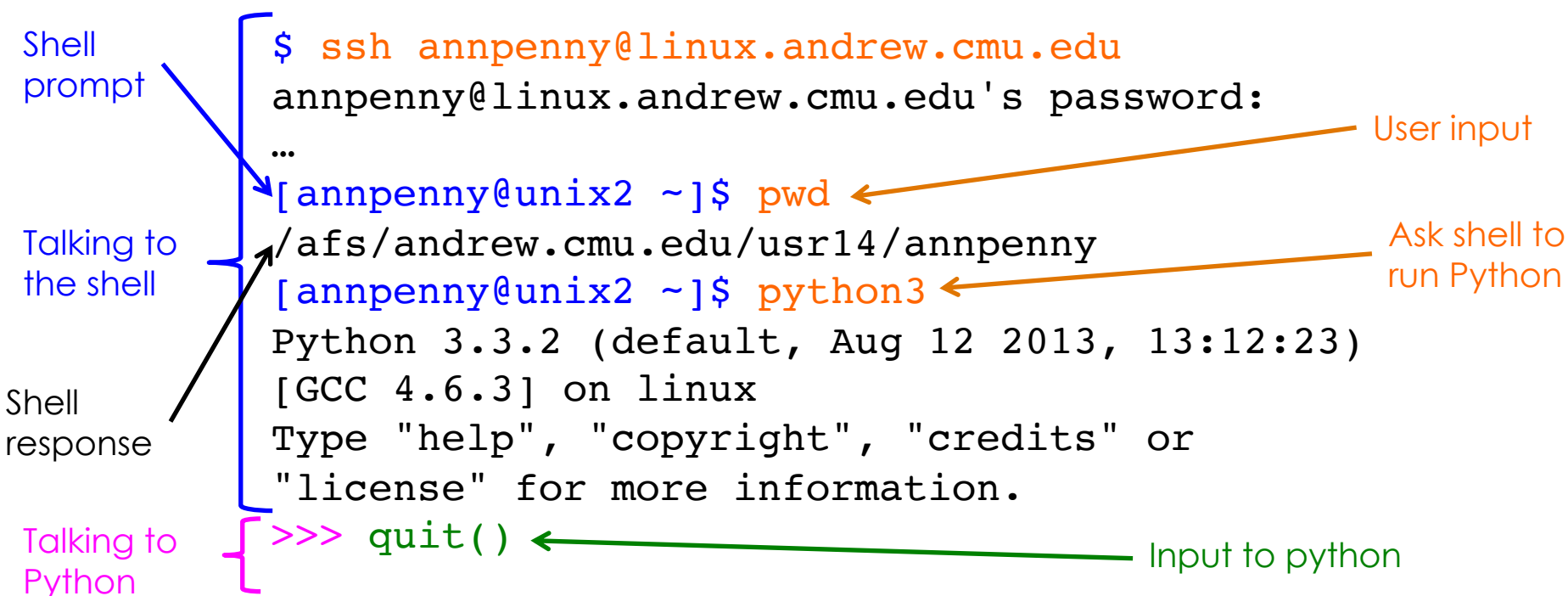
```
$ ssh annpenny@linux.andrew.cmu.edu
annpenny@linux.andrew.cmu.edu's password:
...
[annpenny@unix2 ~]$ pwd
/afs/andrew.cmu.edu/usr14/annpenny
[annpenny@unix2 ~]$ python3
Python 3.3.2 (default, Aug 12 2013, 13:12:23)
[GCC 4.6.3] on linux
Type "help", "copyright", "credits" or
"license" for more information.
```

Talking to  
Python

```
>>> quit()
```

# Command Line Interfaces

Be aware of the difference between  
“talking to the shell” and “talking to Python”



# Data Types

# Data Types

## Integers

4    15110    -53    0

## Floating Point Numbers

4.0    0.803333333333  
7.34e+014

## Strings

"hello"    "A"    " "    ""  
'there'    ' "'    '15110'

## Booleans

True    False

## Literal None

None

# Arithmetic Expressions

## ■ Mathematical Operators

+ Addition

- Subtraction

\* Multiplication

/ Division

//

Integer division

\*\*

Exponentiation

%

Modulo (remainder)

- Python is like a calculator: type an expression and it tells you the value.

```
>> 2 + 3 * 5  
⇒ 17
```

# Order of Evaluation

Order of operator precedence:



Use parentheses to force alternate precedence

$$5 * 6 + 7 \neq 5 * (6 + 7)$$

Left associativity *except* for  $**$

$$\begin{aligned} 2 + 3 + 4 &= (2 + 3) + 4 \\ 2 ** 3 ** 4 &= 2 ** (3 ** 4) \end{aligned}$$

# Integer Division

In Python3:

- ▣ `7 / 2` equals **3.5**

- ▣ `7 // 2` equals **3**

- ▣ `7 // 2.0` equals **3.0**

- ▣ `7.0 // 2` equals **3.0**

- ▣ `-7 // 2` equals **-4**

- ▣ beware! `//` rounds **down to smaller number, not towards zero**

# Expressions and Statements

*Know the difference!*

- Python evaluates an **expression** to get a *result* (number or other value)
- Python executes a **statement** to perform an action that has an *effect* (printing something, for example)



# Variables

# Variables

- ▣ A variable is *not* an “unknown” as in algebra.
- ▣ In computer programming, a variable is a *place* where you can *store a value*.
- ▣ In Python we store a value using an *assignment statement*

Assignment  
statement

Expression

Python's  
response

```
>> a = 5
```

```
>> a
```

```
=> 5
```

Computer  
memory

a:

5

# Variables

- ❌ A variable is a “known” as in algebra.
- In computer programming, a variable is a *place* where you can *store a value*.
- In Python we store a value using an *assignment statement*

Assignment  
statement

```
>> a = 5
```

Expression

```
>> a
```

Python's  
response

```
=> 5
```

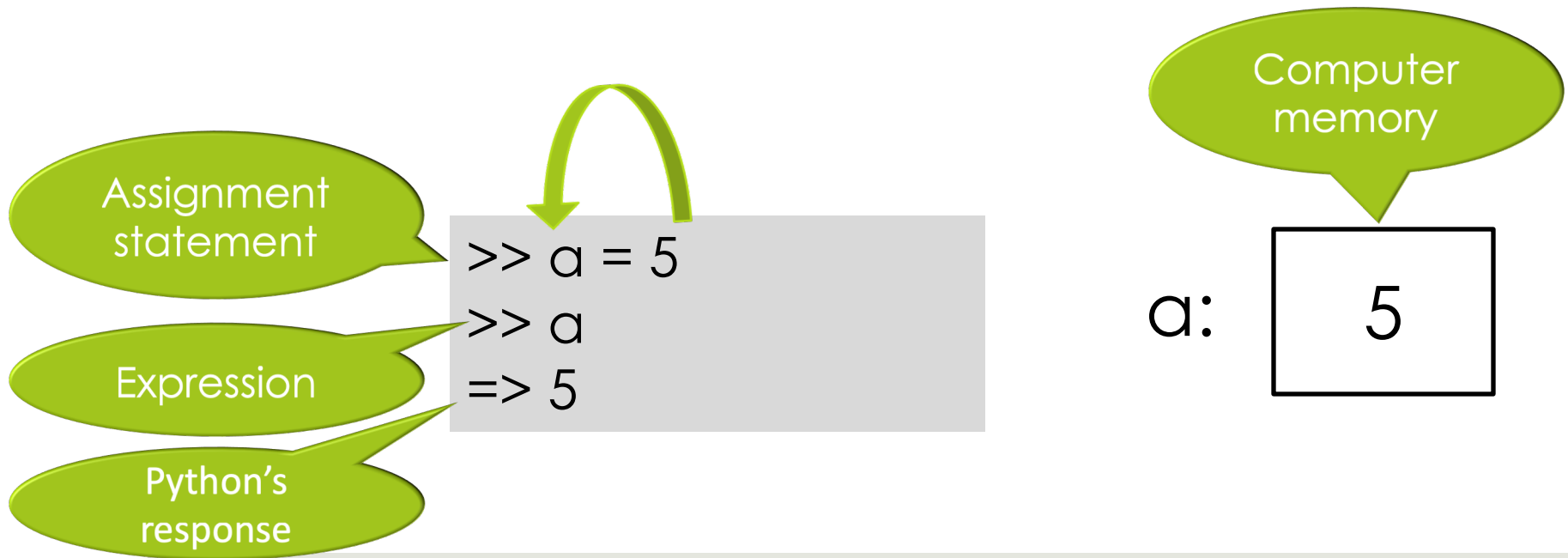
Computer  
memory

a:

5

# Variables

- ❌ A variable is a “known” as in algebra.
- In computer programming, a variable is a *place* where you can *store a value*.
- In Python we store a value using an *assignment statement*



# Variables

Expression

```
>> a  
⇒ 5
```

Assignment  
statement

```
>> b = 2 * a
```

Expression

```
>> b  
⇒ 10
```

Computer  
memory

a:

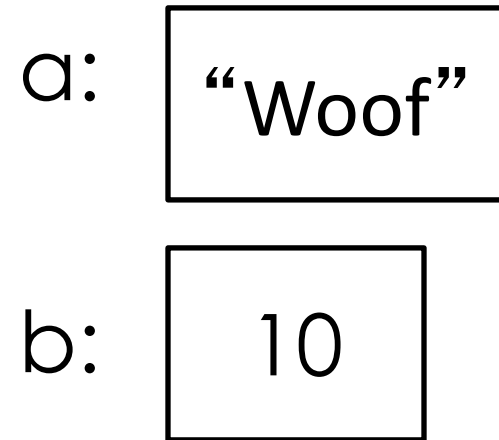
5

b:

10

# Variables

```
>> a  
⇒5  
>> b  
⇒10  
>> a = "Woof"  
>> a  
⇒"Woof"  
>> b  
⇒10
```



Variable b does not “remember” that its value came from variable a.

# Variable Names

- All variable names must start with a letter (lowercase recommended).
- The remainder of the variable name (if any) can consist of any combination of:
  - uppercase letters
  - lowercase letters
  - digits and underscores (\_).
- Identifiers in Python are case sensitive.
  - Example: `Value` is not the same as `value`.

# Syntax and Semantics



# Syntax vs. Semantics

## Syntax

---

- Rules, structure
- Errors result when code is not well formed

## Semantic

---

- Meaning
- Error results when expression/statement can't be evaluated or executed due to meaning

Colorless green ideas sleep furiously

# Colorless green ideas sleep furiously

It can only be the thought of verdure to come, which prompts us in the autumn to buy these dormant white lumps of vegetable matter covered by a brown papery skin, and lovingly to plant them and care for them. It is a marvel to me that under this cover they are laboring unseen at such a rate within to give us the sudden awesome beauty of spring flowering bulbs. While winter reigns the earth reposes but these colorless green ideas sleep furiously.

# Functions

# Function Syntax

```
def functionname (parameterlist) :  
    □ □ □ □ instructions
```

# Function Syntax

`def` is a reserved word and  
cannot be used as a variable name



```
def functionname (parameterlist) :  
    □ □ □ □ instructions
```

# Function Syntax

`def` is a reserved word and  
cannot be used as a variable name

The name you give  
to your function



```
def functionname (parameterlist) :  
□ □ □ □ instructions
```

# Function Syntax

`def` is a reserved word and cannot be used as a variable name

The parameter list can contain 1 or more variables that represent data to be used in the function's computation

The name you give to your function



`def functionname (parameterlist) :`  
`□ □ □ □ instructions`

The diagram illustrates the function syntax with three green arrows pointing to specific parts of the code. One arrow points from the text 'def is a reserved word...' to the 'def' keyword. Another arrow points from 'The name you give to your function' to the 'functionname' placeholder. A third arrow points from 'The parameter list can contain 1 or more variables...' to the 'parameterlist' placeholder. The code is shown with 'def' in a monospace font, 'functionname' and 'parameterlist' in italics, and 'instructions' in italics. There are four square boxes before 'instructions'.

# Function Syntax

`def` is a reserved word and cannot be used as a variable name

The parameter list can contain 1 or more variables that represent data to be used in the function's computation

The name you give to your function

```
def functionname (parameterlist) :  
    □ □ □ □ instructions
```

Declares the start of an indented block



# Function Syntax

`def` is a reserved word and cannot be used as a variable name

The parameter list can contain 1 or more variables that represent data to be used in the function's computation

The name you give to your function

```
def functionname (parameterlist) :  
    □ □ □ □ instructions
```

Declares the start of an indented block

*Indentation is critical.*  
Use spaces *only*, **not tabs!**

# Function Syntax

`def` is a reserved word and cannot be used as a variable name

The parameter list can contain 1 or more variables that represent data to be used in the function's computation

The name you give to your function

```
def functionname (parameterlist) :  
    □ □ □ □ instructions
```

Declares the start of an indented block

Block of code

*Indentation is critical.*  
Use spaces *only*, **not tabs!**

# Functions are general

- Function that takes 1 parameter:

```
def multiply_5(val):  
    return val*5
```

- Function that takes 3 parameters

```
def multiply(v1,v2,v3) :  
    return v1*v2*v3
```

- A function can also have no parameters – but now it can only do one thing!

```
def hello_world():  
    print("Hello World!\n")
```

# Functions are general

- Function that takes 1 parameter:

```
def multiply_5(val):  
    return val*5
```

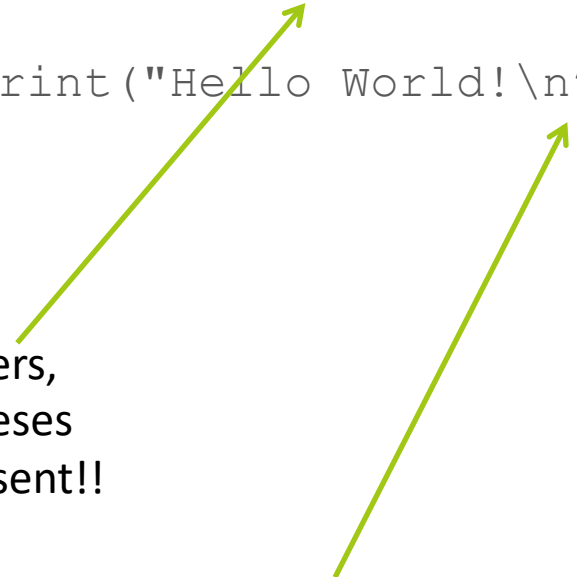
- Function that takes 3 parameters

```
def multiply(v1,v2,v3) :  
    return v1*v2*v3
```

- A function can also have no parameters – but now it can only do one thing!

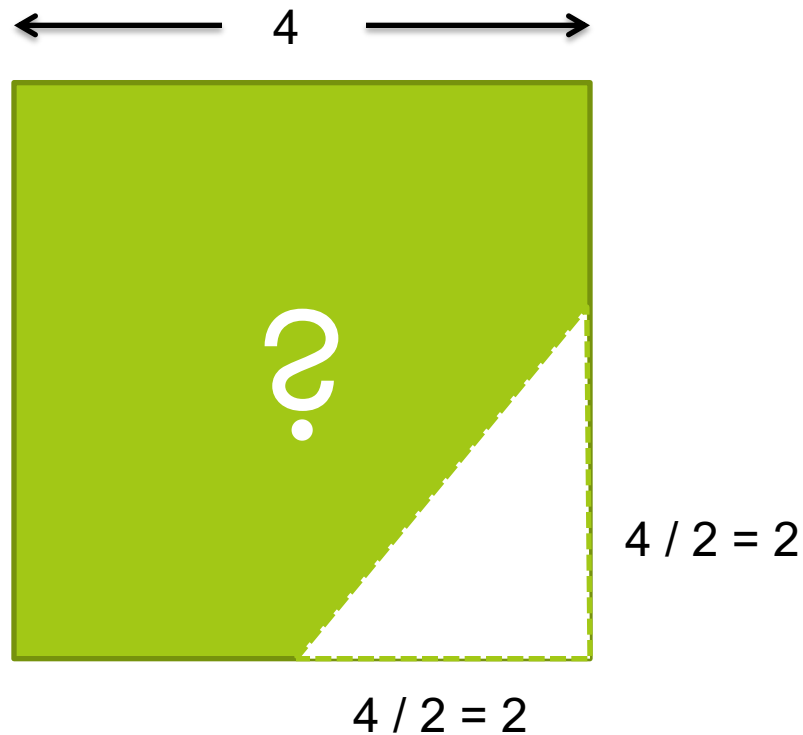
```
def hello_world():  
    print("Hello World!\n")
```

No parameters,  
but parentheses  
must be present!!



(\n is a newline character)

## Example: area of a countertop



# countertop.py

```
def compute_area():  
    square_area = 4 * 4  
    triangle_area = 0.5 * (4 / 2) * (4 / 2)  
    total_area = square_area - triangle_area  
    return total_area
```

← empty parameter list

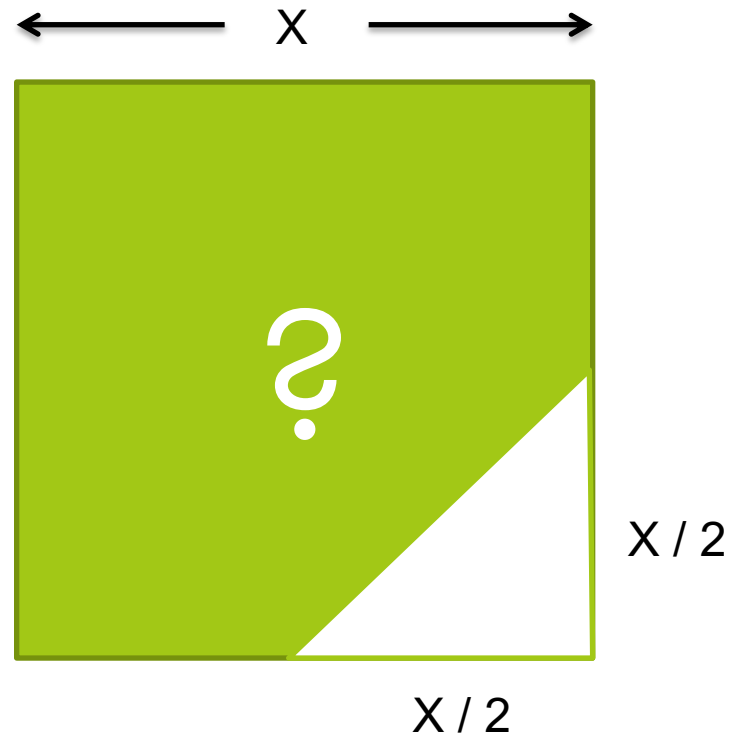
To *call* (use) the function in python3:

```
python3 -i countertop.py
```

```
>>> compute_area()  
14.0
```

← empty argument list

# Generalizing the problem



# countertop.py

```
def compute_area(side):  
    square_area = side * side  
    triangle_area = 0.5 * (side / 2 * side / 2)  
    total_area = square_area - triangle_area  
    return total_area
```

← parameter

To *call* (use) the function in python3:

```
python3 -i countertop.py
```

```
>>> compute_area(109)
```

← argument

(run function with side = 109)



# Function Outputs

# Function Outputs

## Return value



- Function does some action and returns a value:

```
def three_x(x):  
    return x * 3
```

## Return None



- Function does some action and (by default) returns None:

```
def hello_world():  
    print("Hello World!\n")
```

# Function Outputs

- `>>> three_x(12)`  
36     ←..... value returned  
`>>> print(three_x(12))`  
36     ←..... value returned and printed
- `>>> hello_world()`  
Hello World!     ←..... value printed by method  
`>>> print(hello_world())`  
Hello World!     ←..... value printed by method  
None  
                 ←..... value returned and printed

# Function Outputs

- To use a function, we “call” the function.
- A function can return either one answer or no answer (`None`) to its “caller”.
- The `hello_world` function does not return anything to its caller. It simply prints something on the screen.
- The `three_x` function does return its result to its caller so it can use the value in another computation:  
`three_x(12) + three_x(16)`

# Function Outputs

- Suppose we write `compute_area` this way:

```
def compute_area(side):  
    square_area = side * side  
    triangle_area = 0.5 * side/2 * side/2  
    total_area = square_area - triangle_area  
    print(total_area)
```

- Now the following computation does not work. Why?

```
compute_area(109) + compute_area(78)
```

# Built-in Functions (Methods)

# Built-In Functions (Methods)

- Lots of math stuff, e.g., sqrt, log, sin, cos

```
import math  
r = 5 + math.sqrt(2)  
alpha = math.sin(math.pi/3)
```

# Using predefined modules

- `math` is a predefined module of **functions** (also called **methods**) that we can use without writing their **implementations**.

```
math.sqrt(16)
```

```
math.pi
```

```
math.sin(math.pi / 2)
```



# What Could Possibly Go Wrong?

```
alpha = 5  
2 + alhpa
```

```
3 / 0
```

```
import math
```

```
math.sqrt(-1)  
math.sqrt(2, 3)
```

# Try

- Create a function that calculates 18% tip
  - `input("Enter your check's total: ")` would return a user-entered variable. Write a short python script that would advise users of an appropriate tip based on their input.
- Create a function that takes two parameters (mass and radius) and calculates escape velocity. Note:
  - $G = 6.67e-011$
  - Our fine planet has mass of  $5.9742e+024$ , and a radius of 6378.1

$$v_{\text{esc}} = \sqrt{\frac{2 G M}{R}}$$

# Remember

- ▣ Next Lecture: Algorithms
  - ▣ Note resources link and tutorials have extra info on getting running with python
- ▣ Tonight:
  - ▣ Lab 2
  - ▣ OLI Iteration Module
- ▣ For tomorrow (Friday, 9:00AM)
  - ▣ PS2

NO CLASS  
TOMORROW!!



# Talking to the Shell (not Python!!)

Some useful Unix commands

# Useful Unix Commands (Part 1)

All commands must be typed in lower case.

`pwd` --> print working directory, prints where you currently are

`ls` --> list, lists all the files and folders in the directory

`cd` stands for 'change directory':

`cd lab1` --> change to the lab1 directory/folder

`cd ..` --> going up one directory/folder

`cd ../..` --> going up two directories

# Useful Unix Commands (Part 2)

`mkdir lab1 -->` make directory lab1 aka makes a folder called lab1

`rm -r lab1 -->` removes the directory lab1  
(-r stands for recursive, which deletes any possible folders in lab1 that might contain other files)

`cp lab1/file1.txt lab2 -->` copies a file called file1.txt, which is I  
inside of the folder lab1, to the folder lab2

`mv lab1/file1.txt lab2 -->` moves a file called file1.txt, which is inside  
of the folder lab1, to the folder lab2

`zip zipfile.zip file1.txt file2.txt file3.txt -->`  
zips files 1 to 3 into zipfile.zip

`zip -r zipfile.zip lab1/ -->` zips up all files in the lab1 folder into  
zipfile.zip

# Useful Unix Commands (Part 3)

`^c` --> ctrl + c, interrupts running program

`^d` --> ctrl + d, gets you out of python3

`"tab"` - autocompletes what you're typing based on the files in the current folder

`"up"` - cycles through the commands you've typed. Similarly for the opposite effect, press `"down"`



# Useful Unix Commands (Part 4)

`python3 -i test.py` --> load test.py in python3, and  
you can call the functions in test.py.

`gedit lb1.txt &` --> opens up lb1.txt on gedit and & allows you to  
run your terminal at the same time  
( else your terminal pauses until you close gedit)

And lastly, you can always do `man <command>` to find out more about a  
particular command you're interested about (eg. `man cp`, `man ls`)