Announcements

- Class / Recitation slides available on the syllabus page
  - Check morning of for latest version

- We will start grading clickers next lecture
Python Boot Camp!

- Expressions
- Output
- Variables
- Assignment Statements
  - Multiple Assignment
- Input
- Basic Python Types Revisited
Python Programs are Interactive

- Input
- Program
- Output

Possibly repeat
Python Programs are Interactive

Possibly repeat
Using Python as a Calculator

- In the interactive mode you can
  - type *expressions* for python to evaluate (expressions are a particular type of *statement*)
  - python prints the result

```python
>>> 4+5
9
>>> 10/2
5
>>> 10%2
0
```
Using the Math Library

- Besides (+, -, *, /, //, **, %, abs), we have lots of other math functions available in a *math library*.

- A *library* is a module with some useful definitions/functions.
Using the Math Library

Let’s write a program to compute the roots of a quadratic equation

\[
    x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}
\]

\[
    x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}
\]

The only part of this we don’t know how to do is find a square root… but it’s in the math library!
Using the Math Library

- To use a library, we need to make sure this line is in our program, before the first use:

  ```python
  import math
  ```

- Importing a library makes whatever functions are defined within it available to the program.

  ```python
  >>> import math
  >>> math.sqrt(2.0)
  1.4142135623730951
  ```
Using the Math Library

- To access the sqrt library routine, we need to access it as `math.sqrt(x)`.

- Using this dot notation tells Python to use the sqrt function found in the math library module.

- To calculate the root of quadratic equations, you can do `discRoot = math.sqrt(b*b - 4*a*c)`
Output

Output Statements
• A print statement can print any number of expressions.
• Successive print statements will display on separate lines.
• printing a variable prints its value
• A bare print will print a blank line.
  • print()
Clicker Question: Do these programs print the same thing?

1

print(Hello)

2

print(“Hello”)

A: yes
B: no
C: maybe
What does this do in Python?

- Hello = “Hello”

- Now the two programs would be “equivalent”
  - Hello is a variable that stores the value: “Hello”
Assignment Statements

- Simple Assignment

- `<variable> = <expr>`
  variable is an identifier, expr is an expression

- The expression on the RHS is evaluated to produce a value which is then associated with the variable named on the LHS.
Variables

- Just as in algebra, variables are *names* used to represent values (numbers, strings, etc).
- Variables are assigned values using the `=` operator in Python.
- Variable names can be comprised of numbers and alphabetic *characters* but must start with an alphabetic character or with the underscore character “_”

```
a = 4
myPicture = "beach.jpg"
```
CQ: What is printed?

A: beach.jpg

B: beach.jpg

```python
mypicture = "beach.jpg"
print(mypicture)
```
What cannot be a variable

• Some identifiers are part of Python itself. These identifiers are known as *reserved words*. This means they are not available for you to use as a name for a variable, etc. in your program.

• Partial list:
  • and, def, for, is, raise, assert, elif, in, print, if, else, while
Clicker Question

• Which variable name is not valid?

A. a
B. seven
C. 4a
D. _4
Programming Tip

- Use meaningful names!
  - This makes your program much easier to read
  - Example:
    - Foo vs. X vs. R vs. Radius
Differences from Mathematics

\[ y = 3 \]
\[ x = 9 \]
\[ x = y + 6 \]

- Viewed as a system of equations
...but in Python

y = 3
x = 9
x = y + 6

- Separate statements executed one after another ("control flow")
- Each statement assigns a value to a variable
Differences from Mathematics

- But what about …

\[ x = 5 \]

\[ x = x + 6 \]
Assignment

- Expression on the right hand side of the $=$ is evaluated first
  
  $x + 6$

- The result is “stored” in the variable on the left hand side of the $=$

  $x = 11$
Variable Storage

Variables are stored in memory.

Initially memory is blank

(sort of)
Variable Storage

Assigning a variable associates a new ‘name’ with a new ‘value’

\[ a = 4 \]
Variable Storage

Reassigning a variable updates its value

<table>
<thead>
<tr>
<th>a</th>
<th>10</th>
</tr>
</thead>
</table>

a = 4
a = 10
Variable Storage

Some special variable types occupy more than one space in memory

\[
\begin{array}{c|c}
  a & 10 \\
  b & S \\
  \hline
  a & l \\
  \hline
  l & y \\
\end{array}
\]

\[a = 10\]
\[b = \text{“Sally”}\]
Assignment (writes)

- We can view an assignment as a “write” to memory.
- The variable represents a “container” in memory.
- Assignment places, or “writes” the value into that container.
Uses (Reads)

- A use of a variable retrieves the value from the container

<table>
<thead>
<tr>
<th></th>
<th>10 + 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>10</td>
</tr>
</tbody>
</table>
Problem Revisited

- $x = 5$
- $x = x + 6$

$x = 5 + 6$

The result is that the value 11 is stored into the variable $x$.
Assigning Input

- The purpose of an input statement is to get input from the user and store it into a variable.

- `<variable> = eval(input(<prompt>))`

- `input(…)` returns the input as a string! `eval` “evaluates” it, to a Python value

- Try it out, say with input `4+3*5`:
  - `a = input(“type an expression”)`
  - `a`
  - `a = eval(a)`
Assigning Input

- First the prompt is printed
- The input part waits for the user to enter a value and press <enter>
- The expression that was entered is a string
- The string is evaluated to turn it from a string of characters into a Python value (a number).
- The value is assigned to the variable.
Simultaneous Assignment

- $\text{sum, diff} = x+y, x-y$

- How could you use this to swap the values for $x$ and $y$?
  - Why does this not work?
    - $x = y$
    - $y = x$

- We could use a temporary variable…
Temporary variable

- Swapping x and y with a temporary

```python
>>> z = x
>>> x = y
>>> y = z
```

- This is cumbersome (but a must in most languages)
  - We need an extra “box” of memory to do this
Simultaneous Assignment

- We can swap the values of two variables quite easily in Python!
  - `x, y = y, x`
  ```python
  >>> x = 3
  >>> y = 4
  >>> print x, y
  3 4
  >>> x, y = y, x
  >>> print x, y
  4 3
  ```
Simultaneous Assignment

- We can use this same idea to input multiple variables from a single input statement!
- Use commas to separate the inputs

```python
def spamneggs():
    spam, eggs = eval(input("Enter # spam then # of eggs:"))
    print ("You ordered", eggs, "eggs and", spam, "slices of spam. Yum!")

>>> spamneggs()
Enter # spam then # of eggs: 3, 2
You ordered 2 eggs and 3 slices of spam. Yum!
```
Python Types

- Python values are of various “types”
  - Int, Float, String, Character, and more

- Two representations of numbers
  - 1 vs 1.0
Numeric Data Types

• Inside the computer, whole numbers and decimal fractions are represented (encoded) quite differently!

• We say that decimal fractions and whole numbers are two different *data types*.

• The data type of an object determines what values it can have and what operations can be performed on it.
Numeric Data Types

- Whole numbers are represented using the `integer (int for short)` data type.
- These values can be positive or negative whole numbers.
- Internally encoded in binary (last week lecture slides)
Numeric Data Types

- Numbers that can have fractional parts are represented as *floating point* (or *float*) values.
  - How might we encode these?
  - What about two numbers?

- How can we tell which is which?
  - A numeric literal without a decimal point produces an int value
  - A literal that has a decimal point is represented by a float (even if the fractional part is 0)
Numeric Data Types

- Python has a special function to tell us the data type of any value.

```python
>>> type(3)
<class 'int'>
>>> type(3.1)
<class 'float'>
>>> type(3.0)
<class 'float'>
>>> myInt = 32
>>> type(myInt)
<class 'int'>
```
Numeric Data Types

Why do we need two number types?

- Values that represent counts can’t be fractional (you can’t have 3½ iPods)
- Mathematical algorithms can be very efficient with integers
- The float type stores only an approximation to the real number being represented!
- Since floats aren’t exact, use an int whenever possible!
Numeric Data Types

- Operations on ints produce ints, operations on floats produce floats (except for /).

```python
>>> 10.0/3.0
3.3333333333333335
>>> 10/3
3.3333333333333335
>>> 10 // 3
3
>>> 10.0 // 3.0
3.0
>>> 3.0+4.0
7.0
>>> 3+4
7
>>> 3.0*4.0
12.0
>>> 3*4
12
```
Numeric Data Types

• Integer division produces a whole number.

• That’s why 10//3 is 3!

• 10//3 is 3 since 3 goes into 10 just 3 times (with a remainder of 1)

• 10%3 is 1 is the remainder of the integer division of 10 by 3.

• a = (a//b)*(b) + (a%b)
Homework

- Review Chapter 3, Section 1
- Read Chapter 3, Sections 2 and 3
- Work through the python tutorial
Announcements

- Pre Lab 3 is now available
- In Lab 3 you will be using the code you wrote in Lab 2 (it will be provided to you)
- Starting next week your use of iClickers will be graded
Python Boot Camp!

- A few brief string operations
  - You will need these for lab
- Introduction to control flow
- Intuition behind functions
- Arguments / Returns
String Arithmetic?

- In the first week we saw
  - `print("Hello" + "World")`

- The `+` operator “adds” two strings together
  - `print("HelloWorld")`

- What if we wanted a space?
  - `print("Hello  + "World")`
  - `print("Hello" + " World")`
  - `print("Hello" + "  + "World")`
String Arithmetic?

- We have an intuition of what + does, can we reason about * (multiplication)?
  
  ```python
  print("Hello" * "World")
  ```

- But what about?
  
  ```python
  print(3 * "Hello")
  ```
  
  We know that 3 * 5 is short hand for: 5 + 5 + 5 …
  Logically 3 * “Hello” would be short hand for:
  “Hello” + “Hello” + “Hello”
String Arithmetic?

- We know what “Hello” + “Hello” + “Hello” does!
- It gives us: “HelloHelloHello”
Functions

- input
  - program
  - output

- input
  - function
  - output

Arguments

Return
Control Flow

- The term control flow is typically used to describe the order of execution in code.
- This can be represented as a flow chart.
  - The *End* you will see in the flowcharts is not an actual programming construct. It merely represents where the chart ends.
- In simple programs code flows from the first line to the last line, one line at a time.
Intuition behind functions

- We want a mechanism to allow for code reuse
  Consider if we have a program A and we want to write a new program that does what A does only twice

```python
a = 15
b = -7
print(a+b)
```
Program A'

```
a = 15
b = -7
print(a+b)
a = 15
b = -7
print(a+b)
```
We can use a function!

```python
def A():
    a = 15
    b = -7
    print(a+b)
A()
A()
```

Abstractly a function provides a name to a piece of code.
Week 1 Algorithm Example

1. Remove book from bag
2. Place book on desk
3. Open book to first unread page
4. Until end of book, Read()

Let us now look at step 4 as a function call
Functions

Read():

1. Place eyes on top left corner of page

2. For each word on page
   3. ReadWord()

4. If end of page is reached, change current page to next page
Defining A Function

- Function definitions start with `def`
- This is followed by the function name, and any arguments the function should have
- The `:` identifies the start of a block of code
- The body of the function is anything indented beneath this line

```python
def myFunction(a,b):
    print(a)
    print(b)
```
Arguments

- Functions have an important feature
  - They can receive and return values
  - Values are ‘passed to’ a function inside of ( ) when called and returned using a special instruction

```python
def myFunction( input_values ):
    return output_value
```
Functions

- Functions come up in two ways in programs

  * **Function Definition**
    - Where the function is created
    - Starts with `def` in python

  * **Function Call**
    - Where the function is used
    - `result = myFunction(argument1,argument2)`
Function Return

- If a function needs to return a value back to the place in the code where it is called it uses a return statement

```python
def myFunction(a,b):
    print(a)
    print(b)
    return a+b
```
Clicker Question:
Are these programs equivalent?

1

def myFun(a):
    print(a)
    return a

print(myFun(4))

2

def myFun(a):
    print(a)
    print(myFun(4))

A: yes
B: no
Function Call

- Once the function is defined it can be called as many times as one likes

```
myFunction(6,7)
myFunction(4,10)
```

- If the function has a return it can be stored into a variable at the call

```
a = myFunction(6,7)
```
Example Revisited (but slightly modified)

def myFun(a):
    print(a)
    return a+1
myFun(myFun(myFun(4)))
Example Revisited  
(but slightly modified…again)

def myFun(a):
    print(a)
    return a+1
b = myFun(4)
myFun(b)
Clicker Question:
Are these programs equivalent?

A: yes
B: no
Clicker Question:
Are these programs equivalent?

A: yes  B: no
Indentation

- Indentation is very important in python
- In functions it defines where the function ends
- Python will give errors if the indentation is incorrect
Clicker Question:
Are these programs equivalent?

1

a = 3
def myFun(a):
    print (a)
myFun(4)

2

a = 3
print (a)

A: yes
B: no
Naming and Binding

- Arguments are bound at the “call site”
  - Call site: a point in the program where the function is called

- The binding is active up until the return of the function
Naming and Binding

```python
a = 3
def myFun(a):
    print(a)
myFun(4)
print(a)
```
Local Variables

• We saw that arguments can be rebound for the duration of the call
  • What about variables that we define in the function body?
Local Variables

- Abstractly there are two types of local variables
  - 1) those which introduce a new variable
  - 2) those which redefine an existing variable which was initially defined *outside* the function definition

- Local variables, just like arguments, have a lifetime associated with the function call
Local Variables

a = 3
y = 10
def myFun(a):
    print (a)
y = 1
myFun(4)
print(a)
print(y)
Clicker Question: does this program print 3 or 4?

```
x = 3
def myFun():
    print(x)
x = 4
myFun()
```

A: 3
B: 4
Variables and Functions

- Variables used in functions but defined outside of the function can be changed.

- Multiple calls to a function may yield different results if the program “rebinds” such variables.
Variables and Functions

```python
x = 3
def myFun():
    print (x)
x = 4
myFun()
x =5
myFun()
```
Let's Review

- Functions take input and produce output
- Output is provided by the “return” statement
  - Otherwise the function does not provide output
- At the call site of the function the arguments get bound
  - The arguments can rebind variables that have already been defined for the duration of the call
- You can use variables defined outside the function but you must be careful!
Function Arguments

- Functions typically assume something important about the arguments

```python
def sumOfTwo(a, b):
    return a + b
```

- Will this work no matter what we provide as arguments?
Consider the following three cases:

res = sumOfTwo(1,2)
res = sumOfTwo("Hello ", "World")
res = sumOfTwo("Hello", 1)

One of these cases will throw an error. This behavior is defined by the code inside the function.
Function Arguments

- There are two ways to handle this difficulty

1. Tell everyone what the function expects
2. Include checks inside the function to ensure the arguments are what is expected

- A combination of both techniques should be used
This solution uses comments and if-statements.

We will revisit this in later slides

```python
# This function expects two integers
# and returns -1 otherwise

def sumOfTwo(a,b):
    if type(a) == int and type(b) == int :
        return a+b
    return -1
```
What all can I do within the body of a function?

- Define local variables
- Call other functions
- Define new functions
Homework

- Review Chapter 3
- Start reading Chapter 4 (functions), sections 1-7