From HMMM to Python

I learned it last night! Everything is so simple!
Hello World is just print "Hello, world!"

I dunno...
Dynamic typing?
Whitespace?
Come join us!
Programming is fun again!
It's a whole new world up here!
But how are you flying?

I just typed
import antiquity
That's it?
... I also sampled everything in the medicine cabinet for comparison.
But I think this is the Python.
Factorial via Recursion...

```python
x = input()
y = fac(x)
print(y)

def fac(x):
    """ recursive factorial! ""
    if x == 0:
        return 1
    else:
        REC = fac(x-1)
        return x*REC

This is same as return x*f(x-1)
but done in 2 steps...
```

```assembly
00 read r1
01 loadn r15 42
02 call r14 5
03 jump 21
04 nop
05 jnez r1 8
06 loadn r13 1
07 jumpi r14
08 storei r1 r15
09 addn r15 1
10 storei r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 -1
15 loadi r14 r15
16 addn r15 -1
17 loadi r1 r15
18 mul r13 r13 r1
19 jumpi r14
20 nop
21 write r13
22 halt
```
Quiz

Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question): What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Quiz

Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question): What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?
   A. 50  B. 42  C. 43  D. 51  E. 23

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question):
   What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? **What is the lowest memory address that we could start the stack?**
   A. 50   B. 42   C. 43   D. 51   E. 23

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question): **What is the highest value that the stack pointer (r15) gets if the user's original input is 3?**
Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)
   A. 5-19    B. 0-22    C. 0-4    D. 0-3    E. 5-22

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question):
   What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?
   A. 18          B. 6 and 18          C. 6, 18 and 21          D. 2          E. 9, 11, 14 and 16

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question):
   What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Quiz

Refer to the factorial code on the previous slide, and answer the following questions…

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?
   A. 13      B. 5       C. 42       D. 14       E. 12

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question): What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
Refer to the factorial code on the previous slide, and answer the following questions...

1. At what memory address does the stack begin? What is the lowest memory address that we could start the stack?

2. Which lines of code make up the whole factorial function? (If you're bored, match every python line with the corresponding assembly code…)

3. Which line(s) of code place(s) factorial's return value in the right place?

4. On line 13, what value gets stored in the register r14?

5. (Bonus—i.e., it'll help to do the bonus quiz to answer this question):
   What is the highest value that the stack pointer (r15) gets if the user's original input is 3?
   A. 44   B. 46   C. 47   D. 48   E. Other
Bonus Quiz!

Write down what happens in the registers and memory (the stack) as this program runs...

Program (low RAM)  The input is 3.

00 read r1
01 loadn r15 42
02 call r14 5
03 jump 21
04 nop
05 jnez r1 8
06 loadn r13 1
07 jumpi r14
08 storei r1 r15
09 addn r15 1
10 storei r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 -1
15 loadi r14 r15
16 addn r15 -1
17 loadi r1 r15
18 mul r13 r13 r1
19 jumpi r14
20 nop
21 write r13
22 halt

CPU Registers with labels

always-0 register

Input: x

result, return value

return address (line #)

Stack Pointer
Common Pitfalls

• **Problem**: You forgot to store something on the stack and it gets wiped out by accident

• **Solution**: Store EVERYTHING on the stack, even if you think you don't need it!

• **Problem**: Things are coming back from the stack different from when they went on

• **Solution(s)**:
  – Make sure that you take things off in the REVERSE order from how you put them on
  – Make sure your stack pointer ends up the same way it started (for every addn r15 1, you have an addn r15 -1)
Recursive Factorial!

```python
print y
if x == 0: return 1
else:
    REC = fac(x-1)
    return x*REC

input x
```

```assembly
00 read r1
01 loadn r15 42
02 call r14 5
03 jump 21
04 nop
05 jnez r1 8
06 loadn r13 1
07 jumpi r14
08 storei r1 r15
09 addn r15 1
10 storei r14 r15
11 addn r15 1
12 addn r1 -1
13 call r14 5
14 addn r15 -1
15 loadi r14 r15
16 addn r15 -1
17 loadi r1 r15
18 mul r13 r13 r1
19 jumpi r14
20 nop
21 write r13
22 halt
```

setup for call to fac(x)
setup for call to fac(x-1)
unpack after call to fac(x-1)
REC is now in r3
Common Pitfalls

- **Problem**: You forgot to store something on the stack and it gets wiped out by accident
- **Solution**: Store EVERYTHING on the stack, even if you think you don't need it!

- **Problem**: Things are coming back from the stack different from when they went on
- **Solution(s)**:
  - Make sure that you take things off in the REVERSE order from how you put them on
  - Make sure your stack pointer ends up the same way it started (for every `addn r15 1`, you have an `addn r15 -1`
Hmmm Detectives

Hmmm says:
Invalid load target at pc 1: 2
Halting program execution.

00 read r1  # get an integer from the user
            # and hold it in register r1
01 load r2 2  # put 2 in r2
02 mul r1 r2 r1  # r1 = r1 * r2
03 loadn r3 42  # put 42 in r3
04 storei r1 r3  # store the value of r1 at memory
                # location 42
05 load r4 42  # load the value from memory
                # location 42 into r4
06 write r4  # write out (print) the contents of r4
07 halt  # stop here.
Hmmm Detectives

00 read r1  # get an integer from the user
            # and hold it in register r1
01 loadn r2 2  # put 2 in r2
02 mul r1 r2 r1  # r1 = r1 * r2
03 load r3 42  # put 42 in r3
04 storei r1 r3  # store the value of r1 at
                   # memory location 42
05 load r4 42  # load the value from memory
                   # location 42 into r4
06 write r4  # write out (print) the contents of r4
07 halt  # stop here.

Hmmm says:
Invalid store target at pc 4: 0
Halting program execution.
Hmmm Detectives

00 read r1  # get an integer from the user
# and hold it in register r1
01 loadn r2 2  # put 2 in r2
02 mul r1 r2 r1  # r1 = r1 * r2
03 load r3 42  # put 42 in r3
04 store r1 r3  # store the value of r1 at
# memory location 42
05 load r4 42  # load the value from memory
# location 42 into r4
06 write r4  # write out (print) the contents of r4
07 halt  # stop here.

ARGUMENT ERROR:
'r3' IS NOT A VALID NUMBER.

***** ASSEMBLY TERMINATED UNSUCCESSFULLY *****

ASSEMBLY RESULTS:

0 : 0000 0001 0000 0001 00 read r1  # get an integer from
1 : 0001 0010 0000 0010 01 loadn r2 2  # put 2 in r2
2 : 1000 0001 0010 0001 02 mul r1 r2 r1  # r1 = r1 * r2
3 : 0001 0011 0010 1010 03 loadn r3 42  # put 42 in r3
4 : ***ARGUMENT ERROR HERE*** 04 store r1 r3  # store the value of
5 : 0010 0100 0010 1010 05 load r4 42  # load the value from
6 : 0000 0100 0000 0010 06 write r4  # write out (print)
7 : 0000 0000 0000 0000 07 halt  # stop here.
Taxonomy of Programming Models

- Declarative Programming
  - Logic Programming
  - Functional Programming
- Imperative Programming
  - Object-Oriented Programming
  - Low-level Programming

What mood am I in today?

more abstraction away from the machine

Prolog  Racket  Python  Hmmm

closer to the machine
More Powerful Languages…

Functional Languages

Simple, elegant syntax
“Bootstrapping”
No “side effects”
No: for, while, =

Imperative Languages

“Large” syntax
“What’s bootstrapping?”
Major “side effects”
Yes: for, while, =
More Powerful Languages…

**Functional Languages**

Simple, elegant syntax
Few or no “side effects”
No: for, while, =

ML, Lisp, Scheme, Haskell, **Python**

**Imperative Languages**

“Large” syntax
“Side effects” are how you get things done
Yes: for, while, =

Java, C, C++, Pascal, Fortran, **Python**

Legal disclaimer:
Python can be viewed as “functional” if we restrict our attention to a subset of its syntax. May not be considered functional in some states.
Beyond Integers…

```python
>>> 3/4
0
>>> 3.0/4
0.75
>>> 3+2j
(3+2j)
>>> (3+2j)*(1-j)
(5-1j)
>>> S = "spam"
>>> S + "mer"
'spammer'
>>> 3*S
'spamspamspam'
>>> S*3
'spamspamspam'
>>> len(S)
4
```

single quotes and double quotes are the same!
>>> 3 == 1+2
True
>>> 42 == "spam"
False
>>> True == 1
True
>>> False == 0
True
>>> True + 5
6
**string functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>str()</code></td>
<td>converts input to a string</td>
</tr>
<tr>
<td><code>len()</code></td>
<td>returns the string’s length</td>
</tr>
<tr>
<td><code>+</code></td>
<td>concatenates strings</td>
</tr>
<tr>
<td><code>*</code></td>
<td>repeats strings</td>
</tr>
</tbody>
</table>

- `str(42)` returns `'42'`
- `len('42')` returns `2`
- `'XL' + 'II'` returns `'XLII'`
- `'VI'*7` returns `'VIVIVIVIVIVIVI'`

Given these strings

\[
\begin{align*}
\{ & \quad s_1 = "ha" \\
& \quad s_2 = "t"
\end{align*}
\]

What are

\[
\begin{align*}
    s_1 + s_2 \\
    2*s_1 + s_2 + 2*(s_1+s_2)
\end{align*}
\]
String surgery

\[ s = 'harvey mudd college' \]

\[ s[0] \] indexes into the string, returning a one-character string

\[ s[0] \] returns \('h'\)

\[ s[6] \] returns

\[ s[] \] returns \('e'\)

Which index returns \('e'\)?

\[ s[len(s)] \] returns
Negative indices…

```s = 'harvey mudd college'
```

Negative indices count *backwards* from the end!

- `s[-1]` returns `'e'`
- `s[-11]` returns
- `s[-0]` returns

Python can suit any mood…
Slicing what to do if you want a bigger piece of the pie!

$s = 'harvey mudd college'$

$s[]$ slices the string, returning a substring

$s[0:6]$ returns 'harvey'
$s[12:18]$ returns 'colleg'
$s[17:]$ returns 'ge'
$s[::]$ returns 'harvey mudd college'
Slicing

what to do if you want a bigger piece of the pie!

`s = 'harvey mudd college'

`s[: ]` slices the string, returning a substring

- `s[0:6]` returns `'harvey'
- `s[12:18]` returns `'colleg'
- `s[17:]` returns `'ge`
- `s[:]` returns `'harvey mudd college'`

The first index is the first character of the slice.
The second index is ONE AFTER the last character.

A missing index means the end of the string.
Slicing

what to do if you want a bigger piece of the pie!

```
s = 'harvey mudd college'

s[:15]  # slices the string, returning a substring

What are these slices?

s[15:-1]

s[:]

How would you get

'mud'

'e'
```
Lists ~ Strings of anything

\[ L = [ 3.14, [2,40], 'third', 42 ] \]

Square brackets tell Python you want a list.

len(L)

L[0]

L[0:1]

How could you extract from \( L \) \('hi'\)

Indexing: could return a different type
Slicing: always returns the same type

Disclaimer: lists are also arrays…
Skip-Slicing

if you don't want your neighbor to get any...

`s = 'harvey mudd college'

Skip-Slicing

`s[::]` *skip-slices*, returning a subsequence

the third index is the "stride" length

It defaults to 1

`s[0:8:2]` returns `'hre '`

What skip-slice returns `'doe'`

What does this return? `s[1::6]`
pi = [3,1,4,1,5,9]       Q = [ 'pi', 'isn't', [4,2] ]

message = 'You need parentheses for chemistry !'

Part 1

What are 
\[
\begin{aligned}
\text{len}(\text{pi}) \\
\text{len}(Q) \\
\text{len}(Q[1])
\end{aligned}
\]

What slice of \text{pi} is [3,1,4]

What slice of \text{pi} is [3,4,5]

What are 
\[
\begin{aligned}
\text{pi}[0] \ast (\text{pi}[1] + \text{pi}[2])
\end{aligned}
\]

and  
\[
\begin{aligned}
\text{pi}[0] \ast (\text{pi}[1:2] + \text{pi}[2:3])
\end{aligned}
\]

Part 2

What are 
\[
\begin{aligned}
Q[0] \\
Q[0:1] \\
Q[0][1] \\
Q[1][0]
\end{aligned}
\]

What is \text{message}[9:15]

What is \text{message}[::5]

Extra! Mind Muddlers

What is \text{pi}[\text{pi}[2]]?

How many nested \text{pi}'s before \text{pi} [...] \text{pi}[0] [...] produces an error?
Raising and razing lists

**Quiz**

\[ \text{pi} = [3,1,4,1,5,9] \quad \text{Q} = [ '\text{pi}', \text{"isn't"}, [4,2] ] \]

\[ \text{message} = \text{'You need parentheses for chemistry !'} \]

<table>
<thead>
<tr>
<th>Part 1</th>
<th>\text{len(pi)}</th>
<th>\text{len(Q)}</th>
<th>\text{len(Q[1])}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A. 6, 3, 1</td>
<td>B. 5, 2, 2</td>
<td>C. 6, 4, 2</td>
</tr>
<tr>
<td></td>
<td>D. 6, 3, 5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What slice of \text{pi} is [3,1,4]

What slice of \text{pi} is [3,4,5]

What are

\[ \text{pi}[0] \ast (\text{pi}[1] + \text{pi}[2]) \]

and

\[ \text{pi}[0] \ast (\text{pi}[1:2] + \text{pi}[2:3]) \]

**Extra! Mind Muddlers**

What is \text{pi}[\text{pi}[2]]?

How many nested \text{pi}'s before \text{pi} [...\text{pi}[0] ...] produces an error?
Raising and razing lists

"Quiz"

\[ \text{pi} = [3,1,4,1,5,9] \quad \text{Q} = [ \text{'pi'}, \text{'isn't'}, [4,2] ] \]

\[ \text{message} = \text{'You need parentheses for chemistry !'} \]

**Part 1**

\[
\begin{align*}
\text{len(pi)} & \\
\text{len(Q)} & \\
\text{len(Q[1])} &
\end{align*}
\]

What are 

A. \[ \text{pi}[0:2], \text{pi}[0:3:2] \]
B. \[ \text{pi}[1:3], \text{pi}[1:5:2] \]
C. \[ \text{pi}[0:3], \text{pi}[0:5:2] \]
D. \[ \text{pi}[0:3:2], \text{pi}[1:6:2] \]
E. \[ \text{pi}[1:5:2], \text{pi}[0:5:1] \]

What slice of \text{pi} is [3,1,4]?

What slice of \text{pi} is [3,4,5]?

What are

\[ \text{pi}[0] \ast (\text{pi}[1] + \text{pi}[2]) \]
and

\[ \text{pi}[0] \ast (\text{pi}[1:2] + \text{pi}[2:3]) \]

**Extra! Mind Muddlers**

What is \[ \text{pi}[\text{pi}[2]] \]?

How many nested \text{pi}'s before \[ \text{pi}[\ldots\text{pi}[0]\ldots] \] produces an error?
Raising and razing lists  

"Quiz"

\[
\pi = [3,1,4,1,5,9] \quad Q = [\ 'pi',\ 'isn't',\ [4,2]\ ]
\]

message = 'You need parentheses for chemistry !'

**Part 1**

\[
\text{len}(\pi) \\
\text{len}(Q) \\
\text{len}(Q[1])
\]

A. 15, "141414"

What slice of \(\pi\) is [3,1,4]

B. "141414", 15

C. 15, 15

What slice of \(\pi\) is [3,4,5]

D. 15, [1, 4, 1, 4, 1, 4]

E. 12, "141414"

What are

\[
\pi[0] \ast (\pi[1] + \pi[2])
\]

and

\[
\pi[0] \ast (\pi[1:2] + \pi[2:3])
\]

**Extra! Mind Muddlers**

What is \(\pi[\pi[2]]\)?

How many nested \(\pi\)'s before \(\pi[\ldots\pi[0]\ldots]\) produces an error?
Name(s): ____________________________

Raising and razing lists

"Quiz"

\[ \text{pi} = [3,1,4,1,5,9] \]
\[ \text{Q} = [\ 'pi', \ 'isn't', \ [4,2] \ ] \]

message = 'You need parentheses for chemistry !'

<table>
<thead>
<tr>
<th>Part 1</th>
<th>Part 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ \text{len(pi)} ]</td>
<td>[ \text{Q[0]} ]</td>
</tr>
<tr>
<td>[ \text{len(Q)} ]</td>
<td>[ \text{Q[0:1]} ]</td>
</tr>
<tr>
<td>[ \text{len(Q[1])} ]</td>
<td>[ \text{Q[0][1]} ]</td>
</tr>
</tbody>
</table>

What slice of \( \text{pi} \) is \([3,1,4]\)

What slice of \( \text{pi} \) is \([3,4,5]\)

What are
\[ \text{pi}[0] \ast (\text{pi}[1] + \text{pi}[2]) \]
and
\[ \text{pi}[0] \ast (\text{pi}[1:2] + \text{pi}[2:3]) \]

What is \( \text{message}[9:15] \)

What is \( \text{message}[:5] \)

Extra! Mind Muddlers

What is \( \text{pi}[\text{pi}[2]] \)?

How many nested \( \text{pi} \)'s before \( \text{pi}[...\text{pi}[0]...] \) produces an error?

I ♡ ( )
Static vs. Dynamic Typing

Python is dynamically typed

Java is statically typed

```python
>>> x = 5
>>> x = "hello"
>>> x = 3.5

>>> x = 4
>>> y = "2"
>>> x + y
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    x + y
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> str(x) + y
"42"
```
The **in** thing

```python
>>> 3*'i' in 'alien'
False

>>> 'i' in 'team'
False

>>> 'cs' in 'physics'
True

>>> 'sleep' not in 'CS 42'
True

>>> 42 in [41,42,43]
True

>>> 42 in [[42], '42']
False
```

Python is badly confused here...

but otherwise it seems pretty perceptive!

A little bit different for lists...
Functioning in Python

Some basic, built-in functions:

- `abs`: absolute value
- `max`: maximum of lists
- `min`: minimum of lists
- `sum`: sum of elements in lists
- `range`: creates lists
- `round`: only as accurately as it can!

These change data from one type to another:

- `bool`
- `float`
- `int`
- `long`
- `list`
- `str`

These are the most important: `help` `dir`
Functioning in Python

Far more are available in separate files, or *modules*:

```python
import math
math.sqrt( 1764 )  # accesses *math*.py's functions

dir(math)          # lists all of *math*.py's functions

from math import *  # same, but without typing *math.* all of the time...
pi
sin( pi/2 )
```
# my own function!

def dbl( x ):
    """ returns double its input, x """
    return 2*x
Functioning in Python

# my own function!
def dbl( x ):
    """ returns double its input, x """
    return 2*x

Some of Python's baggage...

<table>
<thead>
<tr>
<th>keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>def</strong> starts the function</td>
</tr>
<tr>
<td><strong>return</strong> stops it immediately and sends back the return value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>They begin with #</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Docstrings</th>
</tr>
</thead>
<tbody>
<tr>
<td>They become part of python's built-in help system!</td>
</tr>
<tr>
<td>With each function be sure to include one that</td>
</tr>
<tr>
<td>(1) describes overall what the function does, and</td>
</tr>
<tr>
<td>(2) explains what the inputs mean/are</td>
</tr>
</tbody>
</table>
Functioning in Python

def undo(s):
    """ this "undoes" its string input, s """
    return 'de' + s

>>> undo('caf')

>>> undo(undo('caf'))
Notice how lines with the same level of indentation are in the same code block!

```python
def foo(x):
    """This function demonstrates the use of if, elif, and else""
    if x > 0 and x < 42:
        return "Small"
    elif x >= 42 and x < 100:
        return "Nice!"
    elif 100 <= x < 200:  # <- Funky!
        return "Big"
    else:
        print "That was one nasty number!"
        return "Yuck!"
```
<table>
<thead>
<tr>
<th><strong>Mutable vs. Immutable data</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Changeable</strong> types:</td>
</tr>
<tr>
<td>dictionary</td>
</tr>
<tr>
<td>list</td>
</tr>
<tr>
<td><strong>Unchangeable</strong> types:</td>
</tr>
<tr>
<td>tuple</td>
</tr>
<tr>
<td>string</td>
</tr>
<tr>
<td>int</td>
</tr>
<tr>
<td>float</td>
</tr>
<tr>
<td>bool</td>
</tr>
</tbody>
</table>

What's a dictionary?

I guess I'll have to look it up!
Functions and (immutable) Variables

def fact(a):
    result = 1
    while a > 0:
        result *= a
        a -= 1
    return result

>>> x = 5
>>> y = fact(x)
>>> x
??
Functions and (immutable) Variables

The fine print: This is not the "truth"... but it's close enough

def swap(a, b):
    temp = a
    a = b
    b = temp

>>> x = 5
>>> y = 10
>>> swap(x, y)
>>> print x, y
??

data structure
def swap(L, i1, i2):
    temp = L[i1]
    L[i1] = L[i2]
    L[i2] = temp

>>> MyL = [2, 3, 4, 1]
>>> swap(myL, 0, 3)
>>> print myL
??
**Reference vs. Value**

**Mutable** types:
- dictionary
- list

**Unmutable** types:
- tuple
- string
- float
- int
- bool

L = [5, 42, 'hi']

L = 42

Reference, Pointer, id

Whee!
"Pass By Value"

def main()
    """ calls conform """
    print " Welcome to Conformity, Inc. "
    fav = 7
    conform(fav)
    print " My favorite number is", fav

def conform(fav)
    """ sets input to 42 """
    fav = 42
    return fav
"Pass By Value"

```python
def main()
    """ calls conform """
    print " Welcome to Conformity, Inc. "
    fav = 7
    conform(fav)
    print " My favorite number is", fav

def conform(fav)
    """ sets input to 42 """
    fav = 42
    return fav
```

"Pass by value" means that data is **copied** when sent to a method.
Passing *lists* by value...

```python
def main()
    """ calls conform2 """
    print " Welcome to Conformity, Inc. "
fav = [ 7, 11 ]
conform2(fav)
print " My favorite numbers are", fav

def conform2(fav)
    """ sets all of fav to 42 """
    fav[0] = 42
    fav[1] = 42
```

What gets passed by value here?
Passing **lists** by value...

```python
def main():
    """ calls conform2 """
    print " Welcome to Conformity, Inc. "
    fav = [ 7, 11 ]
    conform2(fav)
    print " My favorite numbers are", fav

def conform2(fav):
    """ sets all of fav to 42 """
    fav[0] = 42
    fav[1] = 42
```

The reference is copied! Can change data elsewhere!
You can change the contents of lists in functions that take those lists as input.

(Actually, lists or any mutable objects)

Those changes will be visible everywhere.

(Immutable objects are safe, however)