Looking forward to Week 1...

Homework 1

Problem 0:  Reading + response...  
Problems 1+2:  Slicing and indexing: These are lab...
Problem 3:  Functions! In lab or beyond...

[A] What other work *might* *adventure()* have encouraged you to procrastinate...? 

[B] What if cs5 were now *finished* with Picobot?

---

due Monday
Reminder of your Challenge: 
Learn names!

Stand up – and introduce yourself to TWO people to your left!

Around the room: “This is X and Y and I am Z.”
The *challenge* of programming...

**Syntax**
- How it looks

**Semantics**
- What it does

**Intent**
- What it should do

- Human-typed input
- Machine-produced output
- Human-desired output
learning a language \sim syntax
unavoidable, but not the point

... but learning CS \sim semantics
guiding how machines think!
Inside the machine...

What's behind the scenes: Processing + Memory:

Computation

Data Storage

variables ~ boxes

name: x
type: int
LOC: 312

memory location 312

name: y
type: int
LOC: 324

memory location 324
Join me, in the machine!

Memory!

Random Access Memory

a big line of boxes, each with a name, type, location, and value
Memory!

Random Access Memory

A big line of boxes, each with a name, type, location, and value.

512 MB of memory
Memory!

Random Access Memory

- **41**
  - name: x
  - type: int
  - LOC: 312

- **42**
  - name: y
  - type: int
  - LOC: 324

- **83**
  - name: z
  - type: int
  - LOC: 336

- **83**
  - name: int
  - type: int
  - LOC: 348

A big line of boxes, each with a name, type, location, and value.

512 MB of memory

- **On or off**
- **Bit** = smallest amt. of info.: 0 or 1
- **Byte** = 8 bits
- **Word** = 64 bits

Now, that's a bit, unboxed!
All languages use **datatypes**

<table>
<thead>
<tr>
<th>Type</th>
<th>Example</th>
<th>What is it?</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>3.14 or 3.0</td>
<td>numeric values with a fractional part, <em>even if the fractional part is .0</em></td>
</tr>
<tr>
<td>int</td>
<td>42 or 10**100</td>
<td>integers – Python has <em>infinite precision ints!</em></td>
</tr>
<tr>
<td>bool</td>
<td>True or False</td>
<td>the T/F results from a test or comparison: <code>==, !=, &lt;, &gt;, &lt;=, &gt;=</code></td>
</tr>
</tbody>
</table>

Hey! Someone can't spell! 

"Boolean values"

"Boolean operators"

George Boole

(type(x))
Python operators

- parens: ( )
- power: **
- negate: -
- times, mod, divide: *, /, %, //
- add, subtract: +, -
- compare: >, ==, <
- assign: =

It's not worth remembering all these *+/* things! I’d recommend parentheses over precedence.
// integer division

7 // 3
8 // 3
9 // 3
30 // 7

\( x \div y \) is \( x/y \), rounded-down to an integer
the "equals" operators

SET equals isn't equal to TEST equals

Difference between == and === in JavaScript
Are numbers enough for *everything*?

*Yes and no...*

You need *lists* of numbers, as well!

and *strings* - lists of characters - too.

Both of these are Python *sequences*...
**Strings: Textual Data**

```
s = 'scrippps'
c = 'college'
```

- **Type...** `type(s)`
  - Returns a type `(str)`

- **Length** `len(s)`

- **Add!** `s + c` 'scripppscollege'

- **Multiply!!** `2*s + 3*c`
  - 'scripppsscripppscollegecollegecollegecollege'
**strings**: textual data

Given

\[
\begin{align*}
    s_1 &= 'ha' \\
    s_2 &= 't'
\end{align*}
\]

What are

\[
\begin{align*}
    s_1 + s_2 &= 'hat' \\
    2s_1 + s_2 + 2(s_1+s_2) &= 'hahahat hat hat hat'
\end{align*}
\]
strings: textual data

Given \[
\begin{align*}
    s_1 &= \text{'ha'} \\
    s_2 &= \text{'t'}
\end{align*}
\]

What are \( s_1 + s_2 \) \( \hat{\text{hat}} \)

\[2s_1 + s_2 + 2(s_1 + s_2)\] 

\( \hat{\text{hahathathathathat}} \)
Lists ~ collections of any data

\[ M = [ 4, 7, 100, 42, 5, 47 ] \]
Lists ~ collections of any data

Square brackets tell Python you want a list.

Commas separate elements.

\[ M = [4, 7, 100, 42, 5, 47] \]

- \( \text{len}(M) \):
  - top-level length
  - 6

- \( M[0] \):
  - indexing
  - 4

- \( M[0:3] \):
  - slicing
  - [4, 7, 100]
Lists ~ collections of any data

Square brackets tell python you want a list. Commas separate elements.

\[ M = [ 4, 7, 100, 42, 5, 47 ] \]

- **len(M)**: 6 (top-level length)
- **M[0]**: 4 (indexing)
- **M[0:3]**: [4, 7, 100] (slicing)
Lists ~ collections of any data

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

- **len(L)**: 4
- **L[0]**: 3.14
- **L[1]**: [2, 40]
- **L[2]**: third
- **L[3]**: 42
- **L[0:1]**: [3.14]

**top-level length**
- only counts top-level elements

**indexing**
- always returns an element

**slicing**
- always returns a substructure!
Lists ~ collections of *any* data

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

- `len(L)`
- `L[2]`
- `L[ ] [ : ]`

- **top-level length**
  - *only counts top-level elements*
- **indexing**
  - *always returns an element*
- **indexing + slicing**

*Composition Science!*
Indexing and Slicing!

\[ s = \text{'harvey mudd college'} \]

Feeling positive? Python!
Feeling negative? Python!
Indexing and Slicing!

```python
s = 'harvey mudd college'

s[0] == 'h'
s[17] == 'g'
s[8] == 'u'
s[1] == 'a'
s[19] error!
s[6] == 

s[-1] == 'e'
s[-2] == 'g'
s[-11] == 'u'
s[-6] == 'o'
s[-20] error!
s[-0] == "h"

s[0:2] == 'ha'
s[15:18] == 'leg'

s[-2:] == 'ge'
s[:3] == 'har'
s[5:3] == ''
s[5:3:-1] == 'ye'
s[10:17:3] == 'doc'
```

Feeling positive? Python!
Feeling negative? Python!
Indexing and Slicing!

`s = 'harvey mudd college'

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
-19 -17 -15 -13 -11 -9 -7 -5 -3 -1
-18 -16 -14 -12 -10 -8 -6 -4 -2

`s[0] == 'h'
s[17] == 'g'
s[8] == 'u'
s[1] == 'a'
s[19] error!
s[6] == ' '

`s[-1] == 'e'
s[-2] == 'g'
s[-11] == 'u'
s[-6] == 'o'
s[-20] error!
s[-0] == 'h'

Indexing

single-location in a sequence

Can go out of bounds!
Let’s see that...

Feeling positive? Python!
Feeling negative? Python!
Indexing and Slicing!

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

**Slicing**

two-index-subsequence
Optional third value is the "stride"
Omit an index to say "the end"

```
s[0:2] == 'ha'
s[15:18] == 'leg'
s[-2:] == 'ge'
s[:3] == 'har'
s[5:3] == ''
s[5:3:-1] == 'ye'
s[10:17:3] == 'doe'
```
\[ L = [5, 4, 2] \]

\[ s = 'harvey mudd college' \]

\[ L[0] == 5 \]
\[ s[0] == 'h' \]
\[ L[1:] == [4, 2] \]
\[ s[1:] == 'arvey mudd college' \]
\[ s[1::6] == 'amo' \]
\[ L = [5, 4, 2] \]

\[ s = 'harvey mudd college' \]

\[ L[0] == 5 \]
\[ s[0] == 'h' \]

\[ L[1:] == [4, 2] \]
\[ s[1:] == 'arvey mudd college' \]

\[ s[1::6] == 'amo' \]
pi = [3, 1, 4, 1, 5, 9]

L = ['pi', 'isn\'t', [4, 2]]

M = 'You need parentheses for chemistry !'

---

**Part 1**

What is \( \text{pi}[0] \)

What is \( \text{pi}[1:] \)

What are \( \text{len(pi)}, \text{len(L)}, \text{len(L[1])} \)?

What is \( \text{pi}[2:4] \)

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

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

**Part 2**

What is \( \text{L[0]} \)

What is \( \text{L[0][1]} \)

What is \( \text{L[0:1]} \)

What slice of \( \text{M} \) is \'try' \n
What slice of \( \text{M} \) is \'shoe' \n
What is \( \text{M[9:15]} \)

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

---

**Extra! Mind Muddlers**

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

These two are different!
The number 42 is, in *The Hitchhiker's Guide to the Galaxy* by Douglas Adams, the "Answer to the Ultimate Question of Life, the Universe, and Everything", calculated by an enormous supercomputer named Deep Thought over a period of 7.5 million years. Unfortunately, no one knows what the question is. Thus, to calculate the Ultimate Question, a special computer the size of a small planet was built from organic components and named "Earth". The Ultimate Question "What do you get when you multiply six by nine"\(^{[17]}\) was found by Arthur Dent and Ford Prefect in the second book of the series, *The Restaurant at the End of the Universe*. This appeared first in the radio play and later in the novelization of *The Hitchhiker's Guide to the Galaxy*. The fact that Adams named the episodes of the radio play "fits", the same archaic title for a chapter or section used by Lewis Carroll in "The Hunting of the Snark", suggests that Adams was influenced by Carroll's fascination with and frequent use of the number. The fourth book in the series, the novel *So Long, and Thanks for All the Fish*, contains 42 chapters. According to the novel *Mostly Harmless*, 42 is the street address of Stavromula Beta. In 1994 Adams created the *42 Puzzle*, a game based on the number 42.

among many 42 references...
# my own function!

def dbl( x ):
    ''' returns double its input, x '''
    return 2*x

default

comment for other coders
documentation string for all users

Python's keywords
Function *Fun*!

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

```
In[1]undo('caf')
'decaf'
```

*strings, lists, numbers ... all data are fair game*
Function *Fun!

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

In[1] undo('caf')

'decaf'

In[2] undo(undo('caf'))

Strings, lists, numbers ... all data are fair game
def demo(x):
y = x/3
z = g(y)
return z + y + x

def g(x):
result = 4*x + 2
return result

demo(15)
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

Call: demo(15)

Local variables:
- x = 15
- y = 5
- z = ????

"the stack"

they stack.
```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

How functions work...

1. **Call**: `demo(15)`
   - **Local variables**: `x = 15`, `y = 5`, `z = ????`
   - **Stack frame**: `result = 22`
   - **Returns**: 22

2. **Call**: `g(5)`
   - **Local variables**: `x = 5`, `result = 22`
   - **Returns**: 22

They stack.
```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

How functions work...

1. call: demo(15)
   local variables:
   - x = 15
   - y = 5
   - z = ????

2. call: g(5)
   local variables:
   - x = 5
   - result = 22
   returns 22

They stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

stack frame

call: demo(15)
local variables:
    x = 15
    y = 5
    z = 22

they stack.
```python
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result
```

How functions work...

15

call: demo(15)
local variables:
  x = 15
  y = 5
  z = 22
return 42

"the stack"

they stack.
def demo(x):
    y = x/3
    z = g(y)
    return z + y + x

def g(x):
    result = 4*x + 2
    return result

How functions work...

"the stack"

afterwards, the stack is empty..., but ready if another function is called

they stack.
How functions work...

```python
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

what's $f(2)$?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How functions work...

stack frame
local variables:
    x = 2
    need f(1)

stack frame
local variables:
    x = 1
    need f(0)

stack frame
local variables:
    x = 0
    returns 12
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

How do we compute the result?
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

Where does that result go?
How functions work...

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

1

1.

```
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

```
call: f(2)  
local variables: x = 2
need f(1)
```

```
call: f(1)  
local variables: x = 1 
f(0) = 12
result = 22
```

"the stack"
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

# call: f(2)
# local variables:
    x = 2
    f(1) = 22
# result =

What's this return value?
How functions work...

```python
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x
```

**call:** \( f(2) \)

**local variables:**
- \( x = 2 \)
- \( f(1) = 22 \)
- \( \text{result} = 42 \)

which then gets returned...
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

result = f(2)
the result then gets returned...

2

How functions work...
def f(x):
    if x == 0:
        return 12
    else:
        return f(x-1) + 10*x

output
gain, the stack is empty, but ready if another function is called...
\[ L = [5, 4, 2] \]

\[ s = 'harvey mudd college' \]

\[ L[0] == 5 \]

\[ s[0] == 'h' \]

\[ L[1:] == [4, 2] \]

\[ s[1:] == 'arvey mudd college' \]

\[ s[1::6] == 'amo' \]
First + Rest = A recipe for recursion!

```
#  
# vwl example
#
def vwl(S):
    """vwl returns the number of vowels in S
    input: S, which will be a string
    """
    if S == '':
        return 0  # if S is the empty string
    elif S[0] in 'aeiou':
        return 1 + vwl(S[1:])  # if first-of-S is a vowel
    else:
        return 0 + vwl(S[1:])  # otherwise, don't add 1
                            # the 0 + is nice, but not needed
```
def keepvwl(S):
    if len(S) == 0:
        return ''
    elif S[0] in 'aeiou':
        return S[0] + keepvwl(S[1:])
    else:
        return '' + keepvwl(S[1:])

Writing keepvwl, to return 'aie' instead of 3
Want more practice?

- We’ll have some recursion on homework
- PythonBAT (linked from homework) has even more practice
- We’ll try some “weird” recursion in class on Thursday!