Alien Attack? Picobot programmer Z. Dodds was subject of a bizarre attack yesterday by three-eyed aliens. The trinocular terrors, it seems, were conducting experiments that would help them understand “how humans think.”

It seems the aliens used a shrinking ray, which let them enter the programmer’s head in order to see what was happening. A witness reports deeply disappointed voices emanating from within.

see three-eyed alien attack, p. 42
**Computer** command-line

% cd Desktop
% ls or dir
% python
% python hw1pr1.py

**Python** command-line

```python
>>> 6*7
>>> s = 'harvey mudd college'
>>> len(s)
>>> s.upper()
```

uses a "shell" language

just about all we'll do...

runs the Python language

this feels like CS5's primary topic!
Picobot!?
Jack Ma’s Picobot rules...
Data, data everywhere...
Data, data everywhere...

Data produced each year

1 Zettabyte
1 Exabyte
1 Petabyte

1 Petabyte == 1000 TB
1 TB = 1000 GB


100-years of HD video + audio
Human brain's capacity

References


(life in video) 60 PB: in 4320p resolution, extrapolated from 16MB for 1:21 of 640x480 video (w/sound) – almost certainly a gross overestimate, as sleep can be compressed significantly!

Big Data?

How President Obama’s campaign used big data to rally individual voters.

By Sasha Issenber on December 19, 2012

Big data: The next frontier for innovation, competition, and productivity

Is Big Data an Economic Big Dud?

By JAMES GLANZ
Published: August 17, 2013
Quantifying Trading Behavior in Financial Markets Using *Google Trends*

Tobias Preis, Helen Susannah Moat & H. Eugene Stanley

Profit and loss for an investment strategy based on the volume of the search term *debt*, the best performing keyword in our analysis, with \( \Delta t = 3 \) weeks, plotted as a function of time (blue line). This is compared to the “buy and hold” strategy (red line) and the standard deviation of 10,000 simulations using a purely random investment strategy (dashed lines). The Google Trends strategy using the search volume of the term *debt* would have yielded a profit of 326%.

more-than-average searches for *debt*: sell
fewer-than-average searches for *debt*: buy
Figure S38. Profit and loss for an investment strategy based on the volume of the search term *fun* with $\Delta t = 3$ weeks.
If you torture the data enough, it will confess.

- R. Coates, statistician
Data's elevation?


Google's users

G. Garcia Marquez

Google

Data

Information

Knowledge

Wisdom
# Python's data types

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
<th>What is it?</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>3.14</td>
<td>values with a fractional part</td>
</tr>
<tr>
<td>long</td>
<td>10**100</td>
<td>integers &gt; 2147483647</td>
</tr>
<tr>
<td>int</td>
<td>42</td>
<td>integers &lt;= 2147483647</td>
</tr>
<tr>
<td>bool</td>
<td>True, False</td>
<td>the results from a comparison:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>==, !=, &lt;, &gt;, &lt;=, &gt;=</td>
</tr>
</tbody>
</table>

Hey - someone can't spelle!

George Boole
Datatypes ~ genes...

**Dominant**

- **float**: \( 1.0 / 5 \)
- **long**: \( 10^{100} - 10^{100} \)
- **int**: \( 1 / 5 \)
- **bool**: \( 41 + \text{True} \)

**Recessive**

What will these results be?
Operate!

(  )

**

- 

* % /

+ - 

> == <

= 

higher precedence
O-per-ate!

higher precedence

( **
- *
% / 
+ --
> == < 
==
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.
The `mod` operator

\[
\begin{align*}
7 \mod 3 &= 1 \\
8 \mod 3 &= 2 \\
9 \mod 3 &= 0 \\
16 \mod 7 &= 2
\end{align*}
\]

\texttt{x\%y} returns the \textit{remainder} when \texttt{x} is divided by \texttt{y}

For what values of \texttt{x} are these \texttt{True}?

\[
\begin{align*}
\texttt{x\%2} &= 0 \\
\texttt{x\%2} &= 1 \\
\texttt{x\%4} &= 0 \\
\texttt{x\%4} &= 3
\end{align*}
\]

What happens on these years?
the "equals" operators

This is true – but what is it saying!?
the "equals" operators

SET equals isn't equal to TEST equals

I want ===!
Try it! how = works

Run these lines:

\[ x = 41 \]
\[ y = x + 1 \]
\[ z = x + y \]

What are \( x \), \( y \), and \( z \) at this time?

Then run this:

\[ x = x + y \]

What are \( x \), \( y \), and \( z \) at this time?

Extra!

\[ a = 11/2 \]
\[ b = a \% 3 \]
\[ c = b ** a + b * a \]

What are the values of \( a \), \( b \), and \( c \) after these lines run?
Inside the machine...

What's happening in python:

\[
\begin{align*}
x &= 41 \\
y &= x + 1
\end{align*}
\]

What is happening behind the scenes:

Computation

Data Storage

variables ~ boxes

id, del
Memory!

Random Access Memory

- Name: x, Type: int, LOC: 312
- Name: y, Type: int, LOC: 324
- Name: z, Type: int, LOC: 336
- Name: a, Type: int, LOC: 348

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

512 MB of memory

Wow! Who knew they make memory out of wicker?

<table>
<thead>
<tr>
<th>bit = 1 &quot;bucket&quot; of charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte = 8 bits</td>
</tr>
</tbody>
</table>
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*...
string functions

\texttt{str(42)} \quad \text{returns} \quad '42' \quad \text{converts input to a string}

\texttt{len('42')} \quad \text{returns} \quad 2 \quad \text{returns the string's length}

'XL' + 'II' \quad \text{returns} \quad 'XLII' \quad \text{concatenates strings}

'VI' * 7 \quad \text{returns} \quad 'VIVIVIVIVIVIVI' \quad \text{repeats strings}

composing strings using + and *
**string functions**

- `str(42)` returns `'42'` (converts input to a string)
- `len('42')` returns `2` (returns the string’s length)
- `'XL' + 'II'` returns `'XLII'` (concatenates strings)
- `'VI' * 7` returns `'VIVIVIVIVIVIVI'` (repeats strings)

Given these strings:

\[
\begin{align*}
\text{s1} &= 'ha' \\
\text{s2} &= 't'
\end{align*}
\]

What are:

\[
\begin{align*}
\text{s1} + \text{s2} \\
2*\text{s1} + \text{s2} + 2*(\text{s1}+\text{s2})
\end{align*}
\]
Data ~ *dig in!*

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

\[
s[2] \quad \text{is} \quad \text{'r'}
\]

\[
s[2:4] \quad \text{is} \quad \text{'rv'}
\]

\[
s[:::-1] \quad \text{is} \quad \text{'yevrah'}
\]
Indexing uses [ ]

`s = 'harvey mudd college'`

index

```
s[0]     is
```

```
s[6]     is
```

```
s[]     is 'e'
```

Read as "s-of-zero" or "s-zero"
Negative indices...

```
s = 'harvey mudd college'

s[-1] is
s[-2] is
s[-0] is
```

Negative indices count *backwards* from the end!
Slicing

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

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

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

What's going on here?
Slicing

\[ s = 'harvey mudd college' \]

\[ s[0:6] \text{ is } 'harvey' \]
\[ s[12:18] \text{ is } 'colleg' \]
\[ s[17:] \text{ is } 'ge' \]
\[ s[:] \text{ is } 'harvey mudd college' \]
Slicing

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

What are these slices? \[ s[15:-1] \] is \text{"mud"}

and these? \[ s[:2] \] is \text{"e"}

Don't wor'e' - Be hap'e'!
Skip-Slicing

\[ s[\quad : \\quad : \quad : ] \quad \text{the third index is} \quad \text{the stride length} \quad \text{default is +1} \]

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

\[ s[0:8:2] \quad \text{is} \quad \text{'hre'} \]

\[ s[17:12:-1] \quad \text{is} \quad \text{'doe'} \]

\[ s[1::6] \quad \text{is} \]
Lists ~ collections of any data

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

- Commas separate elements.
- Square brackets tell Python you want a list.
### Lists ~ collections of any data

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

<table>
<thead>
<tr>
<th>len(L)</th>
<th>L[0]</th>
<th>L[0:1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>indexing</td>
<td>slicing</td>
</tr>
</tbody>
</table>

- **length**
- **indexing** *could* return a different type
- **slicing** *always* returns the same type, and *always* returns a substructure!

From L how could you get 'hi'?
\[
\pi = [3, 1, 4, 1, 5, 9]
\]

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

\[
M = 'You need parentheses for chemistry !'
\]

---

**Part 1**

What is `len(pi)`

6

What is `len(L)`

3

What is `len(L[1])`

What is `\pi[2:4]`

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

`\pi[:3]`

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

---

**Part 2**

What is `L[0]`

`'\pi'`

These two are different!

What is `L[0:1]`

What is `L[0][1]`

`'i'`

What slice of `M` is `'try'`?

is `'shoe'`?

What is `M[9:15]`

What is `M[::5]`

---

**Extra! Mind Muddlers**

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

These two are different, too...
\[ \text{pi} = [3,1,4,1,5,9] \]
\[ \text{L} = [ '\text{pi}', "\text{isn't}" , [4,2] ] \]

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

**Part 1**

- What is `len(pi)`? 6
- What is `len(L)`? 3
- What slice of `pi` is `[3,1,4]`?
- What slice of `pi` is `[3,4,5]`?
- What is `pi[:3]`?
- What is `pi[::2]`?
- What is `len(L[1])`?

**Part 2**

- What is `L[0]`? 'pi'
- What is `L[0:1]`? ['pi']
- What slice of `M` is 'try'? M[31:34] is 'shoe'?
- What is `M[::5]`?

**Extra! Mind Muddlers**

\begin{align*}
\text{pi} &= [3, 1, 4, 1, 5, 9] \\
\text{L} &= [ '\text{pi}', \text{"isn't"}, [4, 2] ] \\
\text{M} &= \text{"You need parentheses for chemistry!"}
\end{align*}
Python slices - it dices...

... but wait, there's more!
# my own function!
def dbl( x ):
    """ returns double its input, x """
    return 2x

This doesn't look quite right...
Functioning in Python

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

Python's keywords

comment for other coders
documentation string for all users

Some of Python's baggage...
Functioning in Python

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

>>> undo('caf')
'decaf'

>>> undo(undo('caf'))
'decaf'
```

strings, lists, numbers ...
all data are fair game
Computation's Dual Identity

But what does all this stuff look like?
Functioning across disciplines

**procedure**

```python
def g(x):
    return x**100
```

**structure**

\[ g(x) = x^{100} \]

**CS's googolizer**

defined by *what it does*
+ how efficiently it works

**Math's googolizer**

defined by *what it is*
def flipside(s):
    """ flipside(s): swaps s's sides!
    input s: a string
    """
    x = len(s) / 2
    return s[x:] + s[:x]

This idea is the key to your happiness!
Use variables!

```python
def flipside(s):
    x = len(s)/2
    return s[x:] + s[:x]
```

```python
def flipside(s):
    return s[len(s)/2:] + s[:len(s)/2]
```

Why would computers "prefer" the top version, too?
Challenge

How functions work...

What is \texttt{demo(-4)}? 

```python
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x
```

I might have a guess...
How functions work...

```python
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)  # stack frame
x = -4
return -4 + f(-4)

Hey! This must be a stack-frame frame!
```
def demo(x):
    return x + f(x)

def f(x):
    return 11 * g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)

How functions work...
How functions work...

```python
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)  # ?
```

These are distinct memory locations both holding x's.
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)
How functions work...

def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4) ?

def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4) ?

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)
```

```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)
```
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)  # the "return value"
How functions work...

def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4) 

These are distinct memory locations both holding x's – and now they also have different values!!
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

def demo(x):
    return x + f(x)

def demo(-4):
    x = -4
    return -4 + f(-4)

>>> demo(-4)
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)
46
def demo(x):
    return x + f(x)

def f(x):
    return 11*g(x) + g(x/2)

def g(x):
    return -1 * x

>>> demo(-4)  # 42

How functions work...

stack frame
Douglas Adams's 42

Those zero-eyed aliens are a bit much...

**answer:** 42  **question:** unknown

---

The Ultimate Answer

According to *The Hitchhiker's Guide to the Galaxy*, researchers from a pan-dimensional, hyper-intelligent race of beings constructed the second greatest computer in all of time and space, Deep Thought, to calculate the Ultimate Answer to Life, the Universe, and Everything. After seven and a half million years of pondering the question, Deep Thought provides the answer: "forty-two." The reaction:

"Forty-two!" yelled Loonquawl. "Is that all you've got to show for seven and a half million years' work?"

"I checked it very thoroughly," said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."
Function *stacking*

```python
def demo(x):
    return x + f(x)
def f(x):
    return 11*g(x) + g(x/2)
def g(x):
    return -1 * x
demo(x = -4)
```

"The stack" is a memory area that

1. keeps separate variables for each function call...
2. remembers where to send results back to...

(1) keeps separate variables for each function call...

(2) remembers where to send results back to...
def dbl(x):
    
    return 2*x

>>> ans = dbl(21)

def dblPR(x):
    
    print 2*x

>>> ans = dblPR(21)
def dbl(x):
    """ dbls x? """
    return 2*x

>>> ans = dbl(21)

def dblPR(x):
    """ dbls x? """
    print 2*x

>>> ans = dblPR(21)

return yields the function call's value ...

print just prints stuff to the screen...

... which the shell then prints!
**Challenge!**

```python
def myst(x):
    """ _myst_ery fun' ""
    print "x is", x

    if x <= 1:
        print "Done! Returning 1"
        return 1
    else:
        print "Calling myst("", x-1, ")"
        old_result = myst( x-1 )
        new_result = x * old_result
        print "Returning", new_result
        return new_result
```

What eight lines does **myst(3)** print?
Challenge!

```
def myst(x):
    """_myst_ery fun' ""
    print "x is", x

if x <= 1:
    print "Done! Returning 1"
    return 1
else:
    print "Calling myst("", x-1, ")"
    old_result = myst(x-1)
    new_result = x * old_result
    print "Returning", new_result
    return new_result
```

What eight lines does **myst(3)** print?

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x is 3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calling myst( 2 )</td>
<td>x is 2</td>
<td></td>
</tr>
<tr>
<td>Calling myst( 1 )</td>
<td>x is 1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Done! Returning 1</td>
<td>Returning 2</td>
</tr>
<tr>
<td></td>
<td>Returning 6</td>
<td></td>
</tr>
</tbody>
</table>

... returns 6
Function design
Thinking *sequentially*

**factorial**

\[ 5! = 120 \]

\[ 5! = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1) \times (N-2) \times ... \times 3 \times 2 \times 1 \]
Thinking *sequentially*

factorial

\[ 5! = 120 \]

\[ 5! = 5 \times 4 \times 3 \times 2 \times 1 \]

\[ N! = N \times (N-1) \times \ldots \times 3 \times 2 \times 1 \]

bit later (March and beyond)
Thinking *recursively*

Recursion == *self*-reference!

5! = 120

5! = 5 * 4 * 3 * 2 * 1

N! = N * (N-1) * (N-2) * ... * 3 * 2 * 1
Warning: *this is legal!*

```python
def fac(N):
    return N * fac(N-1)
```

I wonder how this code will STACK up!?
The calls to `fac` will never stop: there's no BASE CASE!

Make sure you have a base case, *then* worry about the recursion...

```
def fac(N):
    return N * fac(N-1)
```
def fac(N):
    if N <= 1:
        return 1
    Ask yourself: "How could I use the factorial of anything smaller than N?" Then do!

Thinking recursively
def fac(N):
    if N <= 1:
        return 1
    else:
        return N*fac(N-1)

**Human:** Base case and 1 step

**Computer:** Everything else
Thinking recursively

```python
def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return rest * N
```

**Human:** Base case and 1 step

**Computer:** Everything else
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

Behind the curtain...

fac(5)

\[ 5 \times fac(4) \]
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)

5 * fac(4)

5 * 4 * fac(3)

Operation waiting...
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

Behind the curtain...

More operations waiting...
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)

5 * fac(4)

5 * 4 * fac(3)

5 * 4 * 3 * fac(2)

5 * 4 * 3 * 2 * fac(1)
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N - 1)

N = 5
N = 4
N = 3
N = 2
N = 1

5 * 4 * 3 * 2 * 1
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)
    5 * fac(4)
        5 * 4 * fac(3)
            5 * 4 * 3 * fac(2)
                5 * 4 * 3 * 2 * 1
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)
    5 * fac(4)
        5 * 4 * fac(3)
            5 * 4 * 3 * 2
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)

5 * fac(4)

5 * 4 * 6
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)

5 * 24
def fac(N):
    if N <= 1:
        return 1
    else:
        return N * fac(N-1)

fac(5)

Result: 120

Look familiar?

0 x*** -> N 0  Recursive step
0 N*** -> X 1  Base case

Behind the curtain...
**Thinking** recursively...

```python
def facWPR( N ):
    if N <= 1:
        print "Hi from the base case!"
        print "    I'm returning", 1
        return 1
    else:
        print "Next call: fac("", N-1, ")"
        rest = facWPR(N-1)  # recursion!
        print "    I got back rest =", rest
        print "    and I'm returning", N*rest
        return N*rest
```

What will print when
\textbf{facWPR(5)} is called?
Let recursion do the work for you.

Exploit self-similarity
Produce short, elegant code

```
def fac(N):
    if N <= 1:
        return 1
    else:
        rest = fac(N-1)
        return rest * N
```
But you *do* need to do one step yourself...

```python
def fac(N):
    if N <= 1:
        return 1
    else:
        return fac(N)
```

(This will not work!)

Recursion's advantage:

It handles arbitrary structural depth – *all at once!*

As a hat, I'm recursive, too! 🐸
The dizzying dangers of having no base case!
Recursion - Wikipedia, the free encyclopedia
A visual form of recursion known as the Droste effect. The woman in this image is holding an object which contains a smaller image of her holding the same ...
en.wikipedia.org/wiki/Recursion - Cached - Similar

Recursion (computer science) - Wikipedia, the free encyclopedia
Recursion in computer science is a way of thinking about and solving problems. In fact, recursion is one of the central ideas of computer science ...
en.wikipedia.org/wiki/Recursion_(computer_science) - Cached - Similar

Recursion -- from Wolfram MathWorld
A recursive process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the entire class ...
mathworld.wolfram.com/Recursion.html - Cached - Similar

recursion
Definition of recursion, possibly with links to more information and implementations.
www.itl.nist.gov/div897/sca/iad/ftml/recursion.html - Cached - Similar

Mastering recursive programming
Recursive design...

(1) Program the base case.  

(2) Find the self-similarity.  

(3) Do one step!  

(4) Delegate the **rest** to recursion...
One step?

...is easy to do with Python

\[ s = 'aliens' \]

How do we get at the *initial character* of \( s \)? \[ s[0] \]

How do we get at ALL THE REST of \( s \)? \[ s[ ] \]

\[ L = [ 42, 21 ] \]

How do we get at the *initial element* of \( L \)? \[ L[0] \]

How do we get at ALL THE REST of \( L \)? \[ L[ ] \]

[ 21 ]
def mylen(s):
    """ returns the number of characters in s
    input: s, a string
    """

s = ''
mylen('') \rightarrow 0

s = 'hi'
mylen('hi') \rightarrow 1 + mylen('')

s = 'recursion'
mylen('recursion') \rightarrow 1 + mylen('')

NOT a space -- this is no characters at all. This is the empty string -- and it has length of 0!

starts with a vowel -- count that vowel and delegate the rest to recursion

wow!
def mylen(s):
    
    # returns the number of characters in s
    # input: s, a string
    
    if p == 0:
        return  # Base case
    else:
        return  # Recursive case

s = ''
mylen('') → 0

s = 'hi'
mylen('hi') → 1 + mylen(   )

s = 'recursion'
mylen('recursion') → 1 + mylen(   )
... complete!

def mylen(s):
    ''' input: any string, s
    output: the number of characters in s
    '''
    if s == '':
        return 0
    else:
        return 1 + mylen(s[1:])

There's not much len left here!
Behind the curtain: how recursion works...

```python
def mylen(s):
    if s == '':
        return 0
    else:
        return 1 + mylen(s[1:])
```

mylen('cs5')

1 + mylen('s5')

1 + 1 + mylen('5')

1 + 1 + 1 + mylen('')

1 + 1 + 1 + 0
Visualizing...

http://www.pythontutor.com/visualize.html

```python
1 def mylen(s):
2     if s == '':
3         return 0
4     else:
5         return 1 + mylen(s[1:])
6
7 tryit = mylen( 'cs5' )
8 print "tryit is", tryit
```

Frames

Objects

Global frame

```
mylen

s "cs5"
```

```
mylen

s "s5"
```

```
mylen

s "5"
```

```
mylen

s "d"
```

Return value

0
```python
def mymax(L):
    """ returns the max of a nonempty list of elements, L """

L = [42]
mymax([42]) → 42

L = [1, 4, 42]
mymax([1, 4, 42]) → mymax( )

L = [4, 1, 3, 42, 7]
mymax([4, 1, 3, 42, 7]) → mymax( )
```
def mymax(L):
    
    if len(L) == 1:
        return L[0]
    
    elif L[0] < L[1]:
        return L[1]
    
    else:
        return mymax([L[1]] + L[2:])

L = [42]
mymax([42]) → mymax([42])

L = [1, 4, 42]
mymax([1, 4, 42]) → mymax([4, 42])

L = [4, 1, 3, 42, 7]
mymax([4, 1, 3, 42, 7]) → mymax([4, 3, 42, 7])
def mymax(L):
    if len(L) == 1:
        return L[0]
    elif L[0] < L[1]:
        return mymax(L[1:])
    else:
        return mymax(L[0:1]+L[2:])

mymax([1, 4, 3, 42, -100, 7])
mymax([4, 3, 42, -100, 7])
mymax([4, 42, -100, 7])
mymax([42, -100, 7])
mymax([42, 7])
mymax([42])
42
def power(b, p):
    """ returns b to the p power
    Use recursion, not **
    Inputs: int b, int p:
        the base and the power
    """

2^0 == 1
power(2, 0) -> 1

2^5 == 2 * 2^4
power(2, 5) -> 2 * 2^4

Do you see the call to power!? 

2^p == 2 * 2
power(2, p) -> 2 * power(..., ...) 

power(b, p) ->
def power(b, p):
    """ returns b to the p power
    Use recursion, not **
    Inputs: int b, int p:
    the base and the power
    ""
    Base case test
    if p == 0 :
        return ____________________________ Base case
    else:
        return ____________________________ Recursive case

Want more power?
Handle negative p values w/elif.
E.g., power(5, -1) == 0.2

Try it!
\[ \text{power}(2, 5) \]
\[
2 \times \text{power}(2, 4)
\]
\[
2 \times 2 \times \text{power}(2, 3)
\]
\[
2 \times 2 \times 2 \times \text{power}(2, 2)
\]
\[
2 \times 2 \times 2 \times 2 \times \text{power}(2, 1)
\]
\[
2 \times 2 \times 2 \times 2 \times 2 \times \text{power}(2, 0)
\]
\[
2 \times 2 \times 2 \times 2 \times 2 \times 1 = 32
\]
Picture it!

def sajak(s):
    """ returns the number of vowels in the input string, s ""

' '

sajak('') → 0

'sokay'

sajak('okay') → 1+sajak(null)

'swhat'

sajak('what') → 0+sajak(null)
```python
def sajak(s):
    
    """ returns the number of vowels in the input string, s 
    """
    Base case test
    if s == ' ':
        return

    elif else:
        return

    # starts with a vowel – count that vowel and delegate the rest to recursion
    'okay'
sajak('okay') \rightarrow 1 + sajak('')

    # starts with a consonant – so skip it and delegate the rest!
    'what'
sajak('what') \rightarrow 0 + sajak('')
```

Want more Pat?
What 7-letter English word \( w \) maximizes \( sajak(w) \)?

Try it!
sajak( 'eerier' )
1+ sajak( 'erier' )
1+ 1+ sajak( 'rier' )
1+ 1+ 0+ sajak( 'ier' )
1+ 1+ 0+ 1+ sajak( 'er' )
1+ 1+ 0+ 1+ 1+ sajak( 'r' )
1+ 1+ 0+ 1+ 1+ 0+ sajak( '' )
1+ 1+ 0+ 1+ 1+ 0+ 0

4
def power(b, p):
    
    """ inputs: base b and power p (an int)
    implements: b**p
    """

    if p == 0:
        return 1.0

    elif p < 0:
        return ____________________

    else:
        return b * power(b, p-1)

Recursion is power!
**sajak(s):** # of vowels in s

**Base case?**  When there are no letters, there are **ZERO** vowels

---

**Rec. step?**  Look at the initial character.

- If \( s[0] \) is **NOT** a vowel, the answer is
  
  \[
  \text{sajak}( \ s[1:] \ )
  \]

- If \( s[0] \) is **is** a vowel, the answer is
  
  \[
  1 + \text{sajak}( \ s[1:] \ )
  \]
def sajak(s):
    if s == '':
        return 0
    elif s[0] == 'a' or s[0] == 'e' or ...

but how to check for vowels?
Python is...  

`'i' in 'team'`  
False

`'cs' in 'physics'`  
True

`'i' in 'alien'`  
True

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

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

`42 in [[42], '42']`  
False
def sajak(s):
    if len(s) == 0:
        return 0
    elif s[0] in 'aeiou':
        return 1 + sajak(s[1:])
    else:
        return 0 + sajak(s[1:])

let's input 'eerier' for s

if s[0] IS a vowel, the answer is 1 + the # of vowels in the rest of s

if s[0] is NOT a vowel, the answer is just the number of vowels in the rest of s
The key to understanding recursion is, first, to understand recursion.

- a former CS 5 student

Good luck with Homework #1

tutors @ LAC all week!