Computer Scientists Unveil New High Energy “Bit Accelerator”

(Claremont, AP): A team of CS professors at Harvey Mudd College have announced that they have built a new high energy “bit accelerator” that promises to shed light on the hypothesized existence of sub-bit particles. The accelerator is a Python program that spins a bit (a 0 or a 1) in a loop at very high speeds. The bit is then bombarded by docstrings, causing it break apart into its constituent sub-bit particles. “Although 0’s probably contain nothing, we believe that 1’s may contain fract-ions,” said one excited CS professor. Some researchers feel that the fract-ions may themselves be made of even smaller particles called numerators and denominators, but this is highly speculative and may require many more years of experimental and theoretical research. “This theory of numerators and denominators is not a rational theory,” said one expert. Other experts disagreed. “I think that there is a 1 in 2 chance that there is something to this notion of numerators and denominators,” said one expert, Professor Ray Sheeyo of the Pasadena Institute of Technology.
Learning Goals
• Explain benefits of modular programming
• Practice modular programming

Reminder
• Homework 1 due on Tuesday at 11:59 PM
A word on grading

• How it works: auto-grading coupled with human grading
• Style points
• Your part: check your scores every week, and contact Kevin if there are problems (kherrera@hmc.edu)
Review: the virtues of negative thinking!

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 \\
-5 & -4 & -3 & -2 & -1 \\
\end{array}
\]

my_list = [42, 47, 23, [3.141, 2.718], 5]

```python
>>> my_list[len(my_list)-3]
23

>>> my_list[-3]
23

>>> my_list[-2:]
[[3.141, 2.718], 5]
```
Review: the virtues of negative thinking!

my_string = "I luv spam"

>>> my_string[:-1]
'I luv spa'

>>> my_string[-3:]
'pam'
def spamify(lst):
    '''Add "n spam" to every string in list L.''
    new_lst = []
    for s in lst:
        new_lst.append(s + "n spam")
    return new_lst

>>> spamify(["eggs", "sausage", "oatmeal"])
['eggsn spam', 'sausagen spam', 'oatmealan spam']

Green eggs n spam!
def spam_count(S):
    '''Count occurrences of "spam" in input S.''
    counter = 0
    for i in range(len(S)):
        if S[i:i+4] == "spam":
            counter = counter + 1
    return counter

>>> spam_count("gspamtspammspamn")
3
An alternate use of `in`

```python
>>> for num in range(1, 100): # what we’ve seen before
...  

>>> 42 in [3, 67, 42, 18, 2001] # new use
True

>>> 42 in [13, 33, 300]
False

>>> food = ["carrots", "coffee", "arugula", "spam"]
>>> if "spam" in food: print("Yay!!!")
...  

Yay!!!

>>> "bio" in "symbiont"
True
```
Displaying output for the user: the print function

```python
def verbose():
    print('Prof Wu likes spam.')
    print('How much does she like spam?')
    x = 42
    print('Her favorite number: ', x, '!!!')

>>> verbose()
Prof Wu likes spam.
How much does she like spam?
Her favorite number: 42!!!
```

That is a totally bizarre example!
def print_codons(DNAstring):
    for i in range(0, len(DNAstring), 3):
        print("Next codon: ", DNAstring[i:i+3])
    # no return statement necessary!

>>> print_codons("AAATTTGGGC")
Next codon: AAA
Next codon: TTT
Next codon: GGG
Next codon: C

What colorful codons you have!
def dbl(x):
    return 2 * x

def trbl(x):
    print(2 * x)
    return

def happy(input):
    y = dbl(input)
    return 2 * y

def sad(input):
    y = trbl(input)
    return 2 * y

>>> happy(4)
16

>>> sad(4)
8
TypeError: unsupported operand type(s) for *: 'int' and 'NoneType'
Advantages of modular programming

• Simpler to read understand
• Easier to test and debug
• Easier to modify in future
• Easier to reuse parts of the code

www.modular.org
pi.py
(Modularity and top-down design!)

• “Write a program to compute an approximation of pi using the Monte Carlo method. The program should take the number of darts as input, report the approximation of pi, and draw the board showing the dart throws.”

• What are parts of this problem which can/should each have their own functions?

• Let’s write the function signatures and docstrings!
def main(num_darts):
    """ Takes a number num_darts as input and returns an approximation of pi using a monte carlo simulation with that number of darts. """

def throw(n):
    """ Takes a number darts to be thrown using random uniform coordinates in a 2x2 box with x ranging from -1 to 1 and y from -1 to 1. Returns a list of (x, y) coordinates of darts. """

def num_inside_circle(dart_list):
    """ Takes a list of dart coordinates as input and returns the # that landed on or inside the circle. """

def draw_picture(dart_list):
    """ Draws the dartboard and the darts """
Professor P.I. Pette: calculate median genome size in intracellular pathogens vs. free livers

<table>
<thead>
<tr>
<th>speciesList</th>
<th>habitatList</th>
<th>genomeSizeList</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Agrobacterium tumefaciens C58&quot;</td>
<td>[&quot;free living&quot;]</td>
<td>5100</td>
</tr>
<tr>
<td>&quot;Chlamidia psittaci 1H&quot;</td>
<td>[&quot;intracellular&quot;]</td>
<td>1450</td>
</tr>
<tr>
<td>&quot;Salmonella typhimurium&quot;</td>
<td>[&quot;intracellular&quot;, &quot;free living&quot;]</td>
<td>4780</td>
</tr>
<tr>
<td>&quot;Lactococcus lactis C2&quot;</td>
<td>[&quot;free living&quot;]</td>
<td>2500</td>
</tr>
<tr>
<td>&quot;Escherichia coli K12&quot;</td>
<td>[&quot;free living&quot;, &quot;commensal&quot;]</td>
<td>4670</td>
</tr>
</tbody>
</table>

http://www.genomesize.com/prokaryotes/table1/
The data (in Python)

habitat_list = [['free living'], ['intracellular'], ['intracellular', 'free living'], ['free living'], ['free living', 'commensal'], ['intracellular'], ['free living', 'commensal'], ['free living'], ['free living'], ['free living'], ['free living', 'commensal'], ['free living'], ['intracellular'], ['free living', 'commensal'], ['free living', 'commensal']

genome_size_list = [5100, 1450, 4780, 2500, 4670, 1600, 4670, 1600, 4140, 4330, 6400, 6700, 4592, 3100, 7490, 4200]
**Nick’s code** (calculates median genome size in intracellular pathogens vs. free livers)

```python
from bacGenData import *

def gsize(habitat_list, genome_size_list):
    a = []
    for i in range(len(genome_size_list)):
        if 'intracellular' in habitat_list[i]:
            a.append(genome_size_list[i])
    a.sort()
    if len(a) % 2 != 0:
        mid_index = int((len(a) - 1) / 2)
        print('intracellular', 1.0 * a[mid_index])
    else:
        y = int((len(a) / 2) - 1)
        b = int(len(a) / 2)
        print('intracellular', (a[y] + a[b]) / 2.0)

    z = []
    for i in range(len(genome_size_list)):
        if 'free living' in habitat_list[i]:
            z.append(genome_size_list[i])
    z.sort()
    if len(z) % 2 != 0:
        mid_index = int((len(z) - 1) / 2)
        print('free living', 1.0 * z[mid_index])
    else:
        y = int((len(z) / 2) - 1)
        b = int(len(z) / 2)
        print('free living', (z[y] + z[b]) / 2.0)
```
Prof Pette now wants to look at **exclusively** intracellular or free living species. But Nick is out of town...
def genome_size(habitat_list, genome_size_list):
    '''Calculate and print median genome size in intracellular pathogens and also in free livers.'''

    intra_sizes = find_sizes("intracellular", habitat_list, genome_size_list)
    free_sizes = find_sizes("free living", habitat_list, genome_size_list)

    print("intracellular", median(intra_sizes))
    print("free living", median(free_sizes))
def median(L):
    '''Calculates the median of list L.''
    L.sort()
    if len(L) % 2 != 0:
        # odd number of elements, take middle one
        mid_index = int((len(L) - 1) / 2)
        return 1.0 * L[mid_index]
    else:
        # even number of elements
        mid_index1 = int((len(L) / 2) - 1)
        mid_index2 = int(len(L) / 2)
        return (L[mid_index1] + L[mid_index2]) / 2.0

[2,7,18,23,500]          [2,7,18,23]

>>> median([500,18,2,23,7])
18.0
def find_sizes(habitat, habitat_list, genome_size_list):
    '''Find all species which live in habitat, and return a list of
    their genome sizes.'''

    out_list = []

    return out_list

>>> find_sizes("commensal", habitat_list, genome_size_list)
[4670, 4140, 4592, 7490, 4200]
def find_sizes(habitat, habitat_list, genome_size_list):
    '''Find all species which live in habitat, and return a list of
    their genome sizes.'''

    out_list = []

    for i in range(len(genome_size_list)):
        if habitat in habitat_list[i]:  # Nick's code
            out_list.append(genome_size_list[i])

    return out_list

>>> find_sizes("commensal", habitat_list, genome_size_list)
[4670, 4140, 4592, 7490, 4200]

This does what Nick’s code did...
def find_sizes(habitat, habitat_list, genome_size_list):
    '''Find all species which live in habitat, and return a list of their genome sizes.'''

    out_list = []

    for i in range(len(genome_size_list)):
        if len(habitat_list[i]) == 1 and habitat == habitat_list[i][0]: # what Pette asked for
            out_list.append(genome_size_list[i])

    return out_list

>>> find_sizes("commensal", habitat_list, genome_size_list)
[]

This does the modification Prof Pette asked for...

>>> find_sizes("commensal", habitat_list, genome_size_list)
[]
def z(input):
    '''Count z's in a string'''
    counter = 0
    for symbol in input:
        if symbol == 'z':
            counter = counter + 1
    return counter

dictionary = [
    "abdomen",
    "abdominal",
    "abduct",
    "abduction",
    "aberration",
    "abet",
    "abhor",
    "abhorrence",
    "abhorrent",
    "abide",
    "abiding",
    "ability",
    "abject",
    "ablaze",
    ...
    ...
    etc.
    ...
    ...
]
def extremz(word_list):
    '''Find and return the word with the most z's'''
def extremz(word_list):
    '''Find and return the word with the most z's'''
    best_count = 0
    best_word = ""
    for word in word_list:
        count = z(word)
        if count > best_count:
            best_count = count
            best_word = word
    return best_word

This is a common “programming motif”.