Python

Lez go

In the “Interpreter”

```
>>> print(“My code goes next to the carrots”)
My code goes next to the carrots

>>> print(“The output goes in a new line”)
The output goes in a new line
```

Elementary, my dear Python

```
>>> 5 + 4 - 1
8
>>> 6 / 4
1.5
>>> 5 * 4
20
```

```
>>> 14 % 5
4
>>> 6 // 4
1
>>> 3**3
27
```

*In Python 2.7, both / and // are floor divide.

Assigning a variable

```
>>> my_variable = 14
>>> my_variable
14
>>> foo = 1
>>> my_variable + foo
15
```

Setting the value of my_variable to 14, setting the value of foo to 1.

```
>>> bar = 5
>>> bar = 6
>>> bar / 2
3
>>> bar
6
```

Performing an elementary operation on a variable assigned to a number does not change the value of the variable

A useful operation

```
>>> counter = 3
>>> counter += 1
>>> counter
4
>>> counter -= 1
counter
3
```

+= or -= do two things: They perform an addition/subtraction on the variable, then set the variable to that new value!

How could this be useful?

Strings

```
>>> name = “Steven”
>>> name + “ is not my friend anymore”
‘Steven is not my friend anymore’
```

```
>>> print(name)
Steven
```

Variables can be assigned to words as well! They are called ‘strings’, and are surrounded by “ or ‘

Booleans and Logic

```
>>> 5 == 5
True
>>> 5 != 5
False
>>> 5 < 4
False
```

```
>>> True and False
False
>>> True or False
True
>>> not True
False
```

If / Elif / Else

```
>>> if 4 > 5:
...     print(“if case”)
... elif 1 <= 1:
...     print(“else if case”)
... else:
...     print(“else case”)
...
else if case
```

Your friend, the while loop

```
>>> while <this is true>:
...     <do some stuff>
...     <get closer to stopping loop>
```

The ‘while loop’ will repeat until the <this is true> condition becomes false.

A quickie

```
>>> count = 1
>>> coding = ‘fun’
>>> while count < 4:
...     coding += ‘!’
...     count += 1
...
>>> coding
‘fun!!!’
```

[‘Lists’, ‘are’, ‘fun’]

```
>>> my_list = [1, 2, 3, 4, 5]
>>> print(my_list)
[1, 2, 3, 4, 5]

>>> word_list = [‘Apple’, ‘Orange’, ‘cat’]
```

Lists can hold lots of types of data:
numbers, strings, even other lists!

Manipulating Lists

```
>>> plain = [1, ‘thing’, 2, ‘do’, 3, ‘words’]
>>> white = [4, ‘you’]
>>> theres_only = plain + white
>>> theres_only
[1, ‘thing’, 2, ‘do’, 3, ‘words’, 4, ‘you’]
```

List Indices

The first item of a list is now the zeroth item of the list. CS.

```
>>> len([item0, item1, item2])
3
```

Length function does not start with 0!

List Indices

```
>>> my_list = [‘Ah’, ‘luhv’, ‘kittehz’]
>>> my_list[0]
‘Ah’
>>> my_list[1]
‘luhv’
>>> my_list[2]
‘kittehz’
```

You can get items from a list with this notation <--

The number in the hard brackets indicates the index of the item you want.

Mutability

```
>>> my_list = [17, 45, 100]
>>> my_list[1] = ‘altered’
>>> my_list
[17, ‘altered’, 100]

>>> my_list[3]
IndexError: list index out of range
```

Lists are “mutable” data, which means you can change them after they are created.

A simple list loop

```
>>> my_list = [‘iterate’, ‘over’, ‘me’]
>>> for i in my_list:
...     print(i)
...
‘iterate’
‘over’
‘me’
```

It’s easy to iterate over lists in Python! Use:

```
>>>for (thing) in (list):
...     do stuff here
```

Defining Functions

```
>>> def my_func(x, y):
...     return x * y
...
>>> my_func(5, 6)
30
```

It's easy to write functions in Python! "def" followed by your function's name, followed by (your variables): will get you started!

A note on Indentation

Indents (4 spaces) are important in Python when they are preceding statements.

```
>>> def exclaimer(word):
...     for i in [‘!’, ‘?’, ‘1’]:
...         k = 1
...         while k < 4:
...             word = word + i
...             k = k + 1
...     return word
```

Think of it like nesting in Snap!

If / Else and Indentation

```
>>> chan = “DAE code”
>>> if len(chan) == 8:
...     print(exclaimer(chan))
... else:
...     print(‘not 8 characters’)
...
DAE code!!!???111
```

Lambdas: you hate to love them

```
>>> square = lambda x: x*x
```

What the heck is that?

It's a lambda expression, and they're quite useful.

The word "lambda" begins a lambda expression.

```
lambda x: x*x
```

The variables used are noted here.

The action of the function goes last!

HOFs with lambdas

```
>>> nums = [1, 2, 3, 4]
>>> foo = list(map(lambda x: x*x, nums))
>>> foo
[1, 4, 9, 16]
>>> bar = list(filter(lambda y: y % 2 == 0, nums))
>>> bar
[2, 4]
```

*filter == keep, btw

HOFs

We can also use functions that we created using “def” and variables assigned to functions! Let’s use square from two slides back. odd(x) returns “True” if x is odd.

```
>>> my_list = [1, 2, 3, 4]
>>> list(map(square, my_list))
[1, 4, 9, 16]
>>> list(filter(odd, my_list))
[1, 3]
```

To the Prompts!

>>>

Greater than or equal to:

```
def greater_or_equal(x, y):
    """Return whether x is greater than or equal
    to y."""
```

Greater than or equal to:

```
def greater_or_equal(x, y):
return x >= y
>>> greater_or_equal(5, 5)
True
>>> greater_or_equal(5, 4)
True
>>> greater_or_equal(5, 6):
False
```

Greater than or equal to (Snap!):

Factorial (again…):

```
def factorial(x):
    “”””Return the factorial of x.”””
```

Factorial (again...):

```
def factorial(x):
  if x == 1:
    return 1
  else:
    return x * factorial(x - 1)
>>> factorial(5)
120
```

Factorial (Snap!):

Has seven?

```
def has_seven(n):
```

Has seven?

```
def has_seven(n):
    if n % 10 == 7:
        return True
    elif n == 0:
        return False
    else:
        return has_seven(n // 10)
>>> has_seven(453)
False
>>> has_seven(979)
True
```

Has seven (Snap!)?

```
+has+seven?+ (n) +
if <(n) mod (10) = (7)>
    report <true>
else
    if <(n) = (0)>
        report <false>
    else
        report <has seven? [floor] of ((n) / (10))>
```

Even numbers in a list:

```
def even_numbers(lst):
    “”””Given a list, return the even
    numbers in this list.”””
```

Even numbers in a list:

```
def even_numbers(lst):
    even_list = []
    for i in lst:
        if i % 2 == 0:
            even_list.append(i)
    return even_list

>>> even_numbers([1, 2, 3, 4])
[2, 4]
```

Even numbers (Snap!):

Swap items of a list:

```
def swap(item_x, item_y, lst):
    """Given a list, swap item_x and
    item_y in the list."""

# assume item_x and item_y are indices
```

Swap items of a list:

```
def swap(item_x, item_y, lst):
    temp = lst[item_x]
    lst[item_x] = lst[item_y]
    lst[item_y] = temp
>>> lst = [1, 2, 3, 4]
>>> swap(0, 2, lst)
>>> lst
[3, 2, 1, 4]
```

Swap items of a list (Snap!):

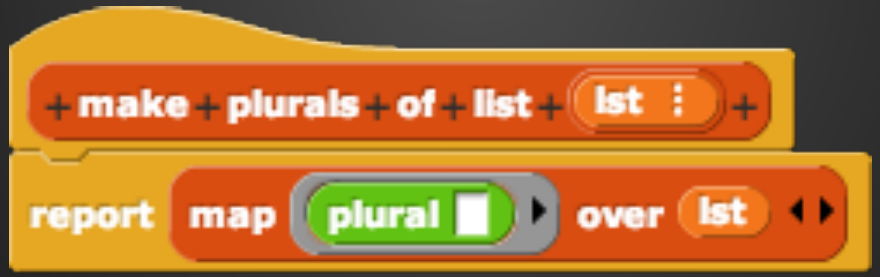
Make plurals:

```
def make_plurals(lst):
    “”””Given a list of words, return a list with
    all the words plural.”””
```

Make plurals:

```
def make_plurals(lst):
   return list(map(lambda string: string + ‘s’,
lst))
>>> lst = [‘car’, ‘cdr’]
>>> make_plurals(lst)
[‘cars’, ‘cdrs’]
>>> lst
[‘car’, ‘cdr’]
```

Make plurals (Snap!):



Factorion

```
def is_factorion(n):
    “”””Return whether the factorial of n’s digits add
    up to n.”””
    # The ‘#’ is used to create one-line comments.
    # You can assume factorial(n) is already written.
```

Factorion - Recursive

```
def is_factorion(n):
    def calc_factorion(n):
        if n == 0:
            return 0
        return calc_factorion(n//10) + factorial(n%10)
    return n == calc_factorion(n)
```

Factorion - Recursive

```
+is_factorion+ n +
report n = sum_factorials n
```

```
+sum_factorials+ n +
if n = 0
  report 0
else
  report
    sum_factorials floor of n / 10 + factorial n mod 10
```

Factorion - Iterative

```
def iter_factorion(n):
    result = 0
    x = n // 10
    y = n % 10
    while not(x==0 and y==0):
        result += factorial(y)
        y = x % 10
        x = x // 10
    return n == result
```

Factorion - Iterative

```
+ iter_factorion + n +
script variables result x y
set result to 0
set x to floor of n / 10
set y to n mod 10
repeat until x = 0 and y = 0
    change result by factorial y
    set y to x mod 10
    set x to floor of x / 10
report n = result
```

Binary Search

```
def bin_search(item, lst, low, high):
  """Search for ITEM in range LOW to HIGH of a sorted list LST."""
```

Binary Search

```
def bin_search(item, lst, low, high):
  """Search for ITEM in range LOW to HIGH of a sorted list LST."""
  while high >= low:
    midpoint = (high + low) // 2
    if lst[midpoint] == item: #return index
      return midpoint
    elif lst[midpoint] < item: #search upper half
      low = midpoint + 1
    else: #search lower half #search lower half
      high = midpoint - 1
  return "Item not found."
```

Binary Search

```
+binary_search+ item + list ⋮ + low + high +
script variables midpoint
repeat until not high > low
    set midpoint to floor of high + low / 2
    if item = item midpoint of list
        report midpoint
    else
        if item midpoint of list < item
            set low to midpoint + 1
        else
            set high to midpoint + -1
report "Item not found."
```

Find Missing

```
def find_missing(lst):
  """Given an unordered list from 0 to n but
missing one number (e.g. [5,3,0,4,1]), find the
missing number."""
```

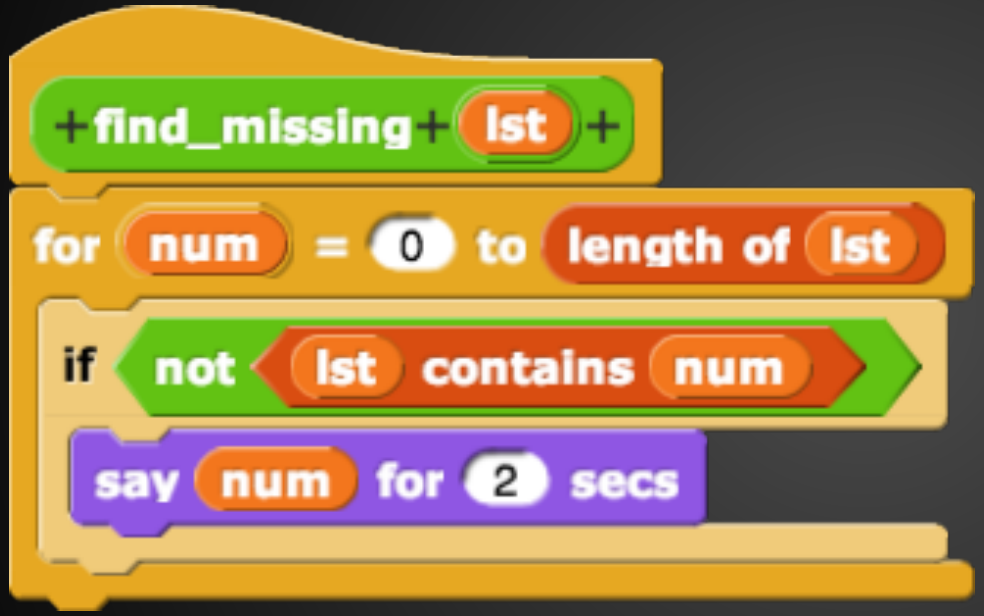
```
>>> alist = list(range(5))
>>> alist
[0,1,2,3,4]
```

```
>>> 5 in alist
False
```

Find Missing

```
def find_missing(lst):
  """Given an unordered list from 0 to n but
missing one number (e.g. [5,3,0,4,1]), find the
missing number."""
   for num in range(len(lst)+1):
      if num not in lst:
         print(num)
```

Find Missing



Leading Questions

Why is it `(length of (lst))` and not ((length of (lst)) + 1))?

What would happen if we tried to report inside a for loop in Snap!? What about in Python?