Python Seminar

- Needed Applications
  - Chrome (website: c9.io)

- GradQuant Resources
  - http://gradquant.ucr.edu/workshop-resources/

- Audience
  - No programing experience.
  - Beginning Python.

This is part 2 of 3 Python seminars.

Data Manipulation with Python

Part 2

Presented by GradQuant

Objectives

Part 1

- Variables (int, float)
- Math (+, -, *, /, %, **)
- Conditional Expressions
- Saved Programs

Part 2

- Strings (input, manipulation, and formatting)
- Lists
- Control Flow (Loops and Branches)

Strings

The String Data Type

- The most common use of personal computers is word processing.
- Text is represented in programs by the *string* data type.
- A string is a sequence of characters enclosed within quotation marks (") or apostrophes (').

The String Data Type

```
>>> str1="Hello"
>>> str2='spam'
>>> print(str1, str2)
Hello spam
>>> type(str1)
<class 'str'>
>>> type(str2)
<class 'str'>
```

The String Data Type

- Getting a string as input

```
>>> firstName = raw_input("Please enter your name: ")
Please enter your name: John
>>> print("Hello", firstName)
Hello John
```

The String Data Type

- We can access the individual characters in a string through *indexing*.
- The positions in a string are numbered from the left, starting with 0.
- The general form is <string>[<expr>], where the value of expr determines which character is selected from the string.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet = "Hello Bob"
>>> greet[0]
'H'
>>> print(greet[0], greet[2], greet[4])
H l o
>>> x = 8
>>> print(greet[x - 2])
B
```

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

- In a string of *n* characters, the last character is at position *n-1* since we start counting with 0.
- We can index from the right side using negative indexes.

```
>>> greet[-1]
'b'
>>> greet[-3]
'B'
```

The String Data Type

- Indexing returns a string containing a single character from a larger string.
- We can also access a contiguous sequence of characters, called a *substring*, through a process called *slicing*.

The String Data Type

- Slicing:
  <string>[<start>:<end>]
- start and end should both be ints
- The slice contains the substring beginning at position start and runs up to **but doesn't include** the position end.

The String Data Type

H	e	l	l	o		B	o	b
0	1	2	3	4	5	6	7	8

```
>>> greet[0:3]
'Hel'
>>> greet[5:9]
' Bob'
>>> greet[:5]
'Hello'
>>> greet[5:]
' Bob'
>>> greet[:]
'Hello Bob'
```

The String Data Type

- If either expression is missing, then the start or the end of the string are used.
- Can we put two strings together into a longer string?
- *Concatenation* “glues” two strings together (+)
- *Repetition* builds up a string by multiple concatenations of a string with itself (*)

The String Data Type

- The function *len* will return the length of a string.

```
>>> "spam" + "eggs"
'spameggs'
>>> "Spam" + "And" + "Eggs"
'SpamAndEggs'
>>> 3 * "spam"
'spamspamspam'
>>> "spam" * 5
'spamspamspamspamspam'
>>> (3 * "spam") + ("eggs" * 5)
'spamspamspameggseggseggseggseggs'
```

The String Data Type

Operator	Meaning
+	Concatenation
*	Repetition
<string>[]	Indexing
<string>[:]	Slicing
len(<string>)	Length
for <var> in <string>	Iteration through characters

Other String Methods

- There are a number of other string methods. Try them all!
  - s.capitalize() – Copy of s with only the first character capitalized
  - s.title() – Copy of s; first character of each word capitalized
  - s.center(width) – Center s in a field of given width

Other String Operations

- s.count(sub) – Count the number of occurrences of sub in s
- s.find(sub) – Find the first position where sub occurs in s
- s.join(list) – Concatenate list of strings into one large string using s as separator.
- s.ljust(width) – Like center, but s is left-justified

Other String Operations

- s.lower() – Copy of s in all lowercase letters
- s.lstrip() – Copy of s with leading whitespace removed
- s.replace(oldsub, newsub) – Replace occurrences of oldsub in s with newsub
- s.rfind(sub) – Like find, but returns the right-most position
- s.rjust(width) – Like center, but s is right-justified

Other String Operations

- s.rstrip() – Copy of s with trailing whitespace removed
- s.split() – Split s into a list of substrings
- s.upper() – Copy of s; all characters converted to uppercase

String Formatting

- String formatting is an easy way to get beautiful output!

```
Change Counter

Please enter the count of each coin type.
Quarters: 6
Dimes: 0
Nickels: 0
Pennies: 0

The total value of your change is 1.5
```

- Shouldn’t that be more like $1.50??

String Formatting

- We can format our output by modifying the print statement as follows:

```
print("The total value of your change is ${0:0.2f}".format(total))
```

- Now we get something like:

```
The total value of your change is $1.50
```

- Key is the string format method.

String Formatting

- <template-string>.format(<values>)
- {} within the template-string mark "slots" into which the values are inserted.
- Each slot has description that includes *format specifier* telling Python how the value for the slot should appear.

String Formatting

```
print("The total value of your change is ${0:0.2f}".format(total)
```

- The template contains a single slot with the description: 0:0.2f
- Form of description: <index>:<format-specifier>
- Index tells which parameter to insert into the slot. In this case, total.

String Formatting

- The formatting specifier has the form: <width>.<precision><type>
- f means "fixed point" number
- <width> tells us how many spaces to use to display the value. 0 means to use as much space as necessary.
- <precision> is the number of decimal places.

String Formatting

```
>>> "Hello {0} {1}, you may have won ${2}" .format("Mr.", "Smith", 10000)
'Hello Mr. Smith, you may have won $10000'

>>> 'This int, {0:5}, was placed in a field of width 5'.format(7)
'This int,     7, was placed in a field of width 5'

>>> 'This int, {0:10}, was placed in a field of witdh 10'.format(10)
'This int,         10, was placed in a field of witdh 10'

>>> 'This float, {0:10.5}, has width 10 and precision 5.'.format(3.1415926)
'This float,     3.1416, has width 10 and precision 5.'

>>> 'This float, {0:10.5f},  is fixed at 5 decimal places.'.format(3.1415926)
'This float,    3.14159, has width 0 and precision 5.'
```

String Formatting

- If the width is wider than needed, numeric values are right-justified and strings are left-justified, by default.
- You can also specify a justification before the width.

```
>>> "left justification: {0:<5}.format("Hi!")
'left justification: Hi!  '
>>> "right justification: {0:>5}.format("Hi!")
'right justification:  Hi!'
>>> "centered: {0:^5}".format("Hi!")
'centered:  Hi! '
```

Lists

Ruth Anderson

CSE 140

University of Washington

What is a list?

- A list is an ordered sequence of values

3	1	4	4	5	9

“Four”	“score”	“and”	“seven”	“years”

- What operations should a list support efficiently and conveniently?
  – Creation
  – Querying
  – Modification

List creation

```
a = [ 3, 1, 2*2, 1, 10/2, 10-1 ]
```

3	1	4	1	5	9

```
b = [ 5, 3, 'hi' ]
```

```
c = [ 4, 'a', a ]
```

List querying

- Extracting part of the list:
  - Single element: **`mylist[index]`**
  - Sublist (“slicing”): **`mylist[startidx : endidx]`**
- Find/lookup in a list
  - **`elt in mylist`**
    - Evaluates to a boolean value
  - **`mylist.index(x)`**
    - Return the int index in the list of the first item whose value is x. It is an error if there is no such item.
  - **`list.count(x)`**
    - Return the number of times x appears in the list.

List mutation

- Insertion
- Removal
- Replacement
- Rearrangement

List insertion

- myist.append(x)
  - Extend the list by inserting x at the end
- mylist.extend(L)
  - Extend the list by appending all the items in the argument list
- mylist.insert(i, x)
  - Insert an item before the a given position.
  - a.insert(0, x) inserts at the front of the list
  - a.insert(len(a), x) is equivalent to a.append(x)

List removal

- list.remove(x)
  - Remove the first item from the list whose value is x
  - It is an error if there is no such item
- list.pop([i])
  - Remove the item at the given position in the list, and return it.
  - If no index is specified, a.pop() removes and returns the last item in the list.

Notation from the Python Library Reference:
The square brackets around the parameter, “[i]”,
means the argument is *optional.*
It does *not* mean you should type square brackets
at that position.

List replacement

- mylist[index] = newvalue
- mylist[start : end] = newsublist
  - Can change the length of the list
  - mylist[start : end] = [] removes multiple elements
  - a[len(a):] = L is equivalent to a.extend(L)

List rearrangement

- list.sort()
  - Sort the items of the list, in place.
  - “in place” means by modifying the original list, not by creating a new list.
- list.reverse()
  - Reverse the elements of the list, in place.

How to evaluate a list expression

There are two new forms of expression:

- [a, b, c, d]   list creation
  - To evaluate:
    - evaluate each element to a value, from left to right
    - make a list of the values
  - The elements can be arbitrary values, including lists
    - ["a", 3, 3.14*r*r, fahr_to_cent(-40), [3+4, 5*6]]

Same tokens "[]" with two *distinct* meanings

List expression

Index expression

- a[b]   list indexing or dereferencing
  - To evaluate:
    - evaluate the list expression to a value
    - evaluate the index expression to a value
    - if the list value is not a list, execution terminates with an error
    - if the element is not in range (not a valid index), execution terminates with an error
    - the value is the given element of the list value (counting from zero)

List slicing

`mylist[startindex : endindex]` evaluates to a sublist of the original list

  - **`mylist[index]`** evaluates to an element of the original list

- Arguments are like those to the **`range`** function
  - **`mylist[start : end : step]`**
  - start index is inclusive, end index is exclusive
  - *All* 3 indices are *optional*
- Can assign to a slice: **`mylist[s : e] = yourlist`**

List slicing examples

```
test_list = ['e0', 'e1', 'e2', 'e3', 'e4', 'e5', 'e6']
```

From e2 to the end of the list:

```
test_list[2:]
```

From beginning up to (but not including) e5:

```
test_list[:5]
```

Last element:

```
test_list[-1]
```

Last four elements:

```
test_list[-4:]
```

Everything except last three elements:

```
test_list[:-3]
```

Reverse the list:

```
test_list[::-1]
```

Get a copy of the whole list:

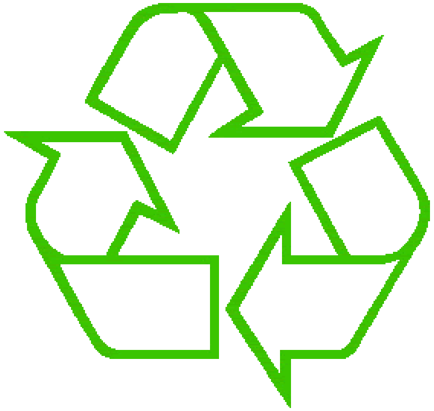
```
test_list[:]
```

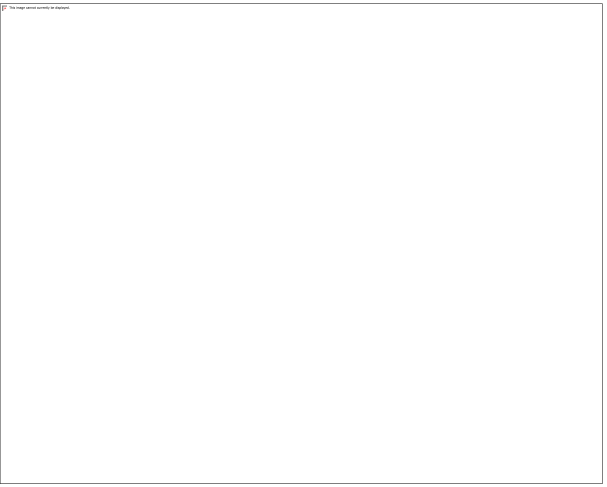
Control flow

Ruth Anderson

UW CSE 140

Winter 2014





Repeating yourself

Making decisions

Temperature conversion chart

Recall exercise from previous lecture

```
fahr = 30
cent = (fahr -32)/9.0*5
print fahr, cent
fahr = 40
cent = (fahr -32)/9.0*5
print fahr, cent
fahr = 50
cent = (fahr -32)/9.0*5
print fahr, cent
fahr = 60
cent = (fahr -32)/9.0*5
print fahr, cent
fahr = 70
cent = (fahr -32)/9.0*5
print fahr, cent
print "All done"
```

Output:

```
30 -1.11
40 4.44
50 10.0
60 15.56
70 21.11
All done
```

Temperature conversion chart

A better way to repeat yourself:

```
for f in [30,40,50,60,70]:
    print f, (f-32)/9.0*5
print "All done"
```

`for` loop

loop variable or iteration variable

A list

Colon is required

Loop *body* is indented

Execute the body 5 times:

- once with f = 30
- once with f = 40
- …

Indentation is significant

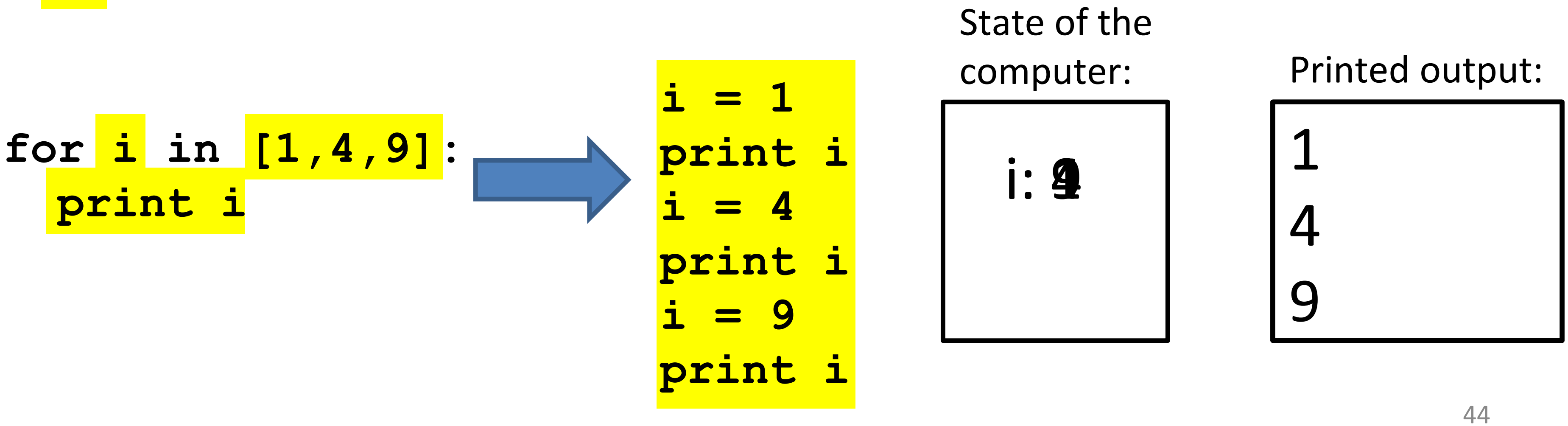
Output:

```
30 -1.11
40 4.44
50 10.0
60 15.56
70 21.11
All done
```

How a loop is executed: Transformation approach

Idea: convert a **`for`** loop into something we know how to execute

1. Evaluate the sequence expression
2. Write an assignment to the loop variable, for each sequence element
3. Write a copy of the loop after each assignment
4. Execute the resulting statements



```
for i in [1,4,9]:
  print i
```

```
i = 1
print i
i = 4
print i
i = 9
print i
```

State of the computer:

i: 1 4 9

Printed output:

```
1
4
9
```

How a loop is executed: Direct approach

1. Evaluate the sequence expression
2. While there are sequence elements left:
   a) Assign the loop variable to the next remaining sequence element
   b) Execute the loop body

Current location in list

```
for i in [1,4,9]:
  print i
```

State of the computer:

i: 9

Printed output:

```
1
4
9
```

The body can be multiple statements

Execute whole body, then execute whole body again, etc.

```
for i in [3,4,5]:
  print "Start body"
  print i
  print i*i
```

loop body:
3 statements

```
Output:       NOT:
Start body    Start body
3             Start body
9             Start body
Start body    3
4             4
16            5
Start body    9
5             16
25            25
```

Convention: often use i or j as loop variable if values are integers

This is an exception to the rule that
variable names should be descriptive

Indentation is significant

- Every statement in the body must have exactly the same indentation
- That’s how Python knows where the body ends

```
for i in [3,4,5]:
    print "Start body"
     print i
    print i*i
```

Error!

- Compare the results of these loops:

```
for f in [30,40,50,60,70]:
    print f, (f-32)/9.0*5
print "All done"

for f in [30,40,50,60,70]:
    print f, (f-32)/9.0*5
    print "All done"
```

The body can be multiple statements

How many statements does this loop contain?

```
for i in [0,1]:
   print "Outer", i
   for j in [2,3]:
      print " Inner", j
      print "  Sum", i+j
   print "Outer", i
```

“nested” loop body: 2 statements

loop body: 3 statements

What is the output?

Output:

```
Outer 0
 Inner 2
  Sum 2
 Inner 3
  Sum 3
Outer 0
Outer 1
 Inner 2
  Sum 3
 Inner 3
  Sum 4
Outer 1
```

Understand loops through the transformation approach

Key idea:

1. Assign each sequence element to the loop variable
2. Duplicate the body

```
for i in [0,1]:
  print "Outer", i
  for j in [2,3]:
    print " Inner", j
```

```
i = 0
print "Outer", i
for j in [2,3]:
  print " Inner", j
i = 1
print "Outer", i
for j in [2,3]:
  print " Inner", j
```

```
i = 0
print "Outer", i
j = 2
print " Inner", j
j = 3
print " Inner", j
i = 1
print "Outer", i
for j in [2,3]:
  print " Inner",j
```

Fix this loop

```
# Goal:  print 1, 2, 3, …, 48, 49, 50
for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print tens_digit * 10 + ones_digit
```

What does it actually print?

How can we change it to correct its output?

Moral: Watch out for *edge conditions* (beginning or end of loop)

Some Fixes

```
for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print tens_digit * 10 + ones_digit + 1

for tens_digit in [0, 1, 2, 3, 4]:
  for ones_digit in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    print tens_digit * 10 + ones_digit

for ones_digit in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print ones_digit
for tens_digit in [1, 2, 3, 4]:
  for ones_digit in [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print tens_digit * 10 + ones_digit
print 50
```

Test your understanding of loops

Puzzle 1:

```
for i in [0,1]:
  print i
print i
```

Output:

```
0
1
1
```

Puzzle 2:

```
i = 5
for i in []:
  print i
```

(no output)

Puzzle 3:

```
for i in [0,1]:
  print "Outer", i
  for i in [2,3]:
    print " Inner", i
  print "Outer", i
```

Reusing loop variable (don't do this!)

inner loop body

outer loop body

```
Outer 0
  Inner 2
  Inner 3
Outer 3
Outer 1
  Inner 2
  Inner 3
Outer 3
```

The range function

A typical for loop does not use an explicit list:

```
for i in range(5):
   … body …
```

The list [0,1,2,3,4]

Upper limit (*exclusive*)

range(5) = [0,1,2,3,4]

Lower limit (*inclusive*)

range(1,5) = [1,2,3,4]

step (distance between elements)

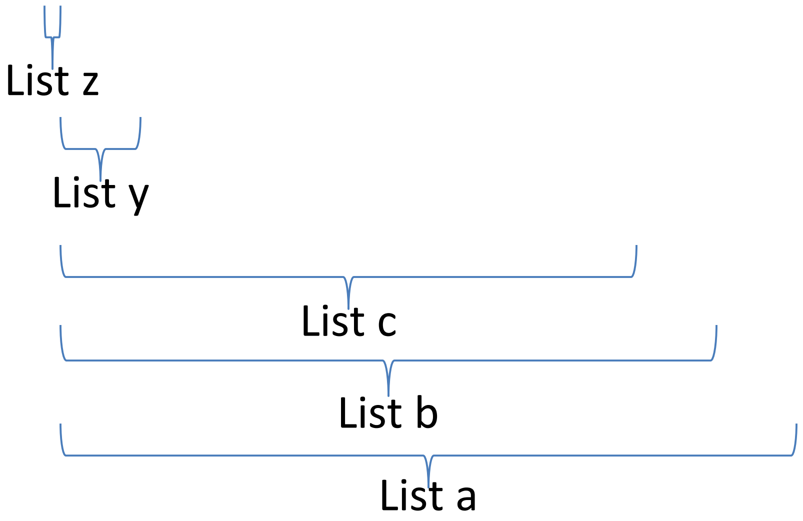
range(1,10,2) = [1,3,5,7,9]

Decomposing a list computation

- To compute a value for a list:
  - Compute a partial result for all but the last element
  - Combine the partial result with the last element

Example: sum of a list:

[3, 1, 4, 1, 5, 9, 2, 6, 5]



sum(List a) = sum(List b) + 5
sum(List b) = sum(List c) + 6
…
sum(List y) = sum(List z) + 3
sum(empty list) = 0

How to process a list: One element at a time

- A common pattern when processing a list:

```
result = initial_value
for element in list:
   result = updated result
use result
```

```
# Sum of a list
result = 0
for element in mylist:
   result = result + element
print result
```

- ***initial_value*** is a correct result for an empty list
- As each element is processed, **`result`** is a correct result for a prefix of the list
- When all elements have been processed, **`result`** is a correct result for the whole list

Some Loops

```
# Sum of a list of values, what values?
result = 0
for element in range(5):
  result = result + element
print "The sum is: " + str(result)

# Sum of a list of values, what values?
result = 0
for element in range(5,1,-1):
  result = result + element
print "The sum is:", result

# Sum of a list of values, what values?
result = 0
for element in range(0,8,2):
  result = result + element
print "The sum is:", result

# Sum of a list of values, what values?
result = 0
size = 5
for element in range(size):
  result = result + element
print "When size = " + str(size) + " result is " + str(result)
```

Some More Loops

```
for size in [1, 2, 3, 4]:
  result = 0
  for element in range(size):
    result = result + element
  print "size=" + str(size) + " result=" + str(result)
print " We are done!"
```

What happens if we move `result = 0`
to be the first line of the program instead?

Examples of list processing

```
result = initial_value
for element in list:
  result = updated result
```

- Product of a list:

```
result = 1
for element in mylist:
  result = result * element
```

- Maximum of a list:

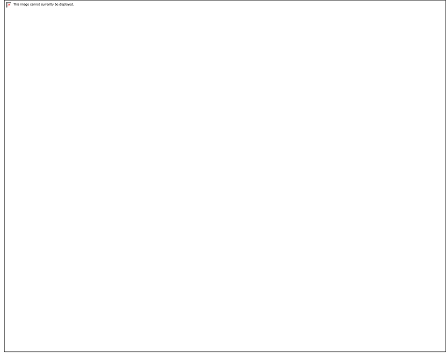
```
result = mylist[0]
for element in mylist:
  result = max(result, element)
```

The first element of the list (counting from zero)

- Approximate the value 3 by 1 + 2/3 + 4/9 + 8/27 + 16/81 + … = $(2/3)^0 + (2/3)^1 + (2/3)^2 + (2/3)^3 + \ldots + (2/3)^{10}$

```
result = 0
for element in range(11):
  result = result + (2.0/3.0)**element
```

Making decisions



- How do we compute absolute value?

abs(5) = 5

abs(0) = 0

abs(-22) = 22

Absolute value solution

If *the value is negative*, negate it.

Otherwise, use the original value.

```
val = -10

# calculate absolute value of val
if val < 0:
    result = - val
else:
    result = val

print result
```

Another approach
that does the same thing
without using `result`:

```
val = -10

if val < 0:
    print - val
else:
    print val
```

In this example, `result` will always be assigned a value.

Absolute value solution

As with loops, a sequence of statements could be used in place of a single statement:

```
val = -10

# calculate absolute value of val
if val < 0:
    result = - val
    print "val is negative!"
    print "I had to do extra work!"
else:
    result = val
    print "val is positive"
print result
```

Absolute value solution

What happens here?

```
val = 5

# calculate absolute value of val
if val < 0:
    result = - val
    print "val is negative!"
else:
    for i in range(val):
        print "val is positive!"
    result = val
print result
```

Another if

It is **not required that anything happens**...

```
val = -10

if val < 0:
     print "negative value!"
```

What happens when val = 5?

The if body can be any statements

then clause

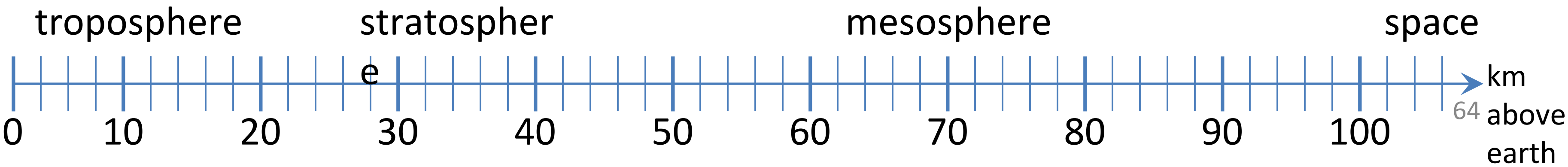
else clause

```
# height is in km
if height > 100:
  print "space"
else:
  if height > 50:
    print "mesosphere"
  else:
    if height > 20:
      print "stratosphere"
    else:
      print "troposphere"
```

```
# height is in km
if height > 100:
  print "space"
elif height > 50:
  print "mesosphere"
elif height > 20:
  print "stratosphere"
else:
  print "troposphere"
```

Execution gets here only if “height > 100” is false

Execution gets here only if “height > 100” is false AND “height > 50” is true



Version 1

```
# height is in km
if height > 100:
  print "space"
else:
  if height > 50:
    print "mesosphere"
  else:
    if height > 20:
      print "stratosphere"
    else:
      print "troposphere"
```

then clause

else clause

t

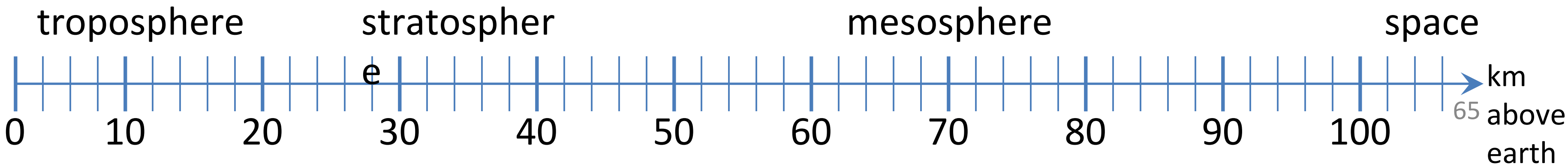
e

t

e

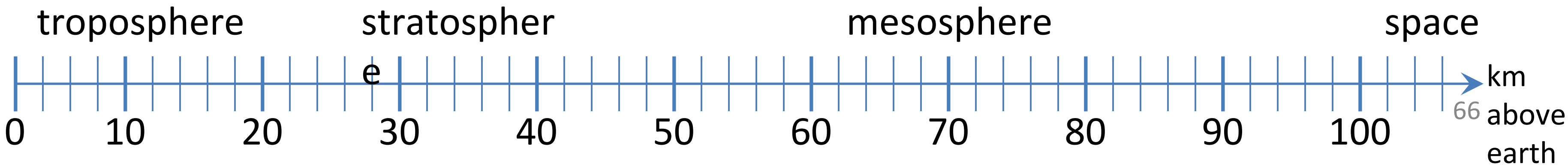
Execution gets here only if “height <= 100” is true

Execution gets here only if “height <= 100” is true AND “height > 50” is true



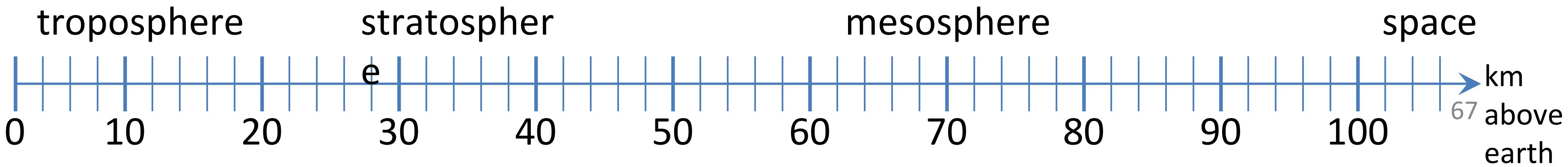
Version 1

```
# height is in km
if height > 100:
  print "space"
else:
  if height > 50:
    print "mesosphere"
  else:
    if height > 20:
      print "stratosphere"
    else:
      print "troposphere"
```



Version 2

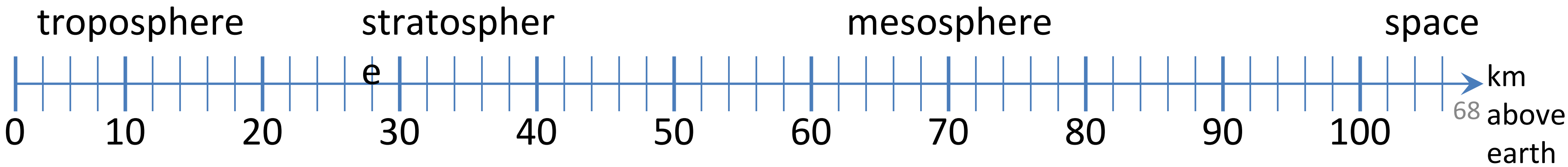
```
if height > 50:
  if height > 100:
    print "space"
  else:
    print "mesosphere"
else:
  if height > 20:
    print "stratosphere"
  else:
    print "troposphere"
```



Version 3

```
if height > 100:
  print "space"
elif height > 50:
  print "mesosphere"
elif height > 20:
  print "stratosphere"
else:
  print "troposphere"
```

ONE of the print statements is guaranteed to execute: whichever condition it encounters **first** that is true

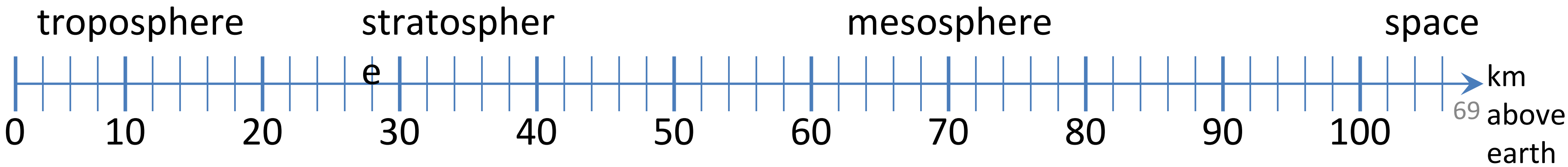


Order Matters

```
# version 3
if height > 100:
  print "space"
elif height > 50:
  print "mesosphere"
elif height > 20:
  print "stratosphere"
else:
  print "troposphere"
```

```
# broken version 3
if height > 20:
  print "stratosphere"
elif height > 50:
  print "mesosphere"
elif height > 100:
  print "space"
else:
  print "troposphere"
```

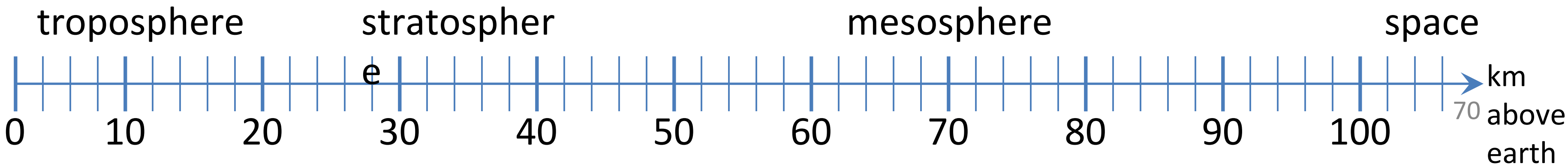
Try height = 72 on both versions, what happens?



Version 3

```
# incomplete version 3
if height > 100:
  print "space"
elif height > 50:
  print "mesosphere"
elif height > 20:
  print "stratosphere"
```

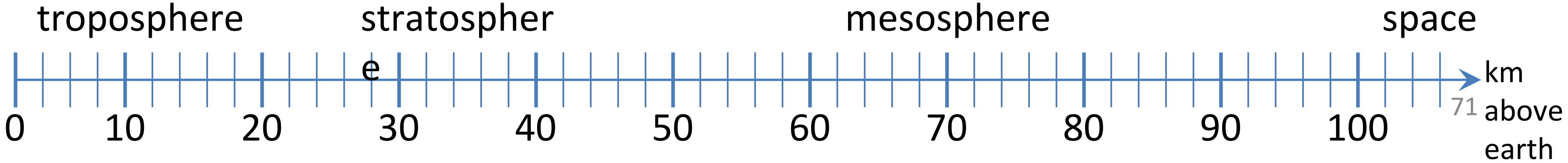
In this case it is possible that nothing is printed at all, when?



What Happens here?

```
# height is in km
if height > 100:
  print "space"
if height > 50:
  print "mesosphere"
if height > 20:
  print "stratosphere"
else:
  print "troposphere"
```

Try height = 72



The then clause *or* the else clause is executed

```
speed = 54
limit = 55
if speed <= limit:
    print "Good job!"
else:
    print "You owe $", speed/fine
```

What if we change speed to 64?

Resources

- Python's website
  - http://www.python.org/
- Python Tutorial - Codecademy
  - http://www.codecademy.com/tracks/python
- GradQuant Resources
  - http://gradquant.ucr.edu/workshop-resources/
  - http://bit.ly/1KlJcEU (slides)
  - http://bit.ly/1Ew4FzZ (code examples)

- Google
  - Search for "python …"
- Stack Overflow website
  - http://stackoverflow.com/

Next Python Seminar

- More data types
  - Sets
  - Dictionaries
- Files
  - Read and write to files.
- Functions
  - Reuse code

GradQuant

- One-on-one Consultations
  - Make appointment on the website
  - http://gradquant.ucr.edu
- Python Seminars
  - *Python Fundamentals (Part 1)*
  - Data Manipulation with Python (Part 2)
  - Advanced Python (Part 3)

Remember to fill out the seminar survey. Thank you!