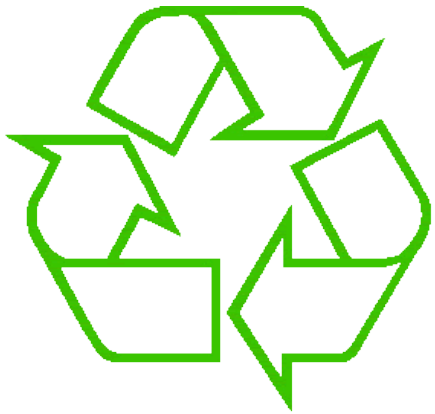
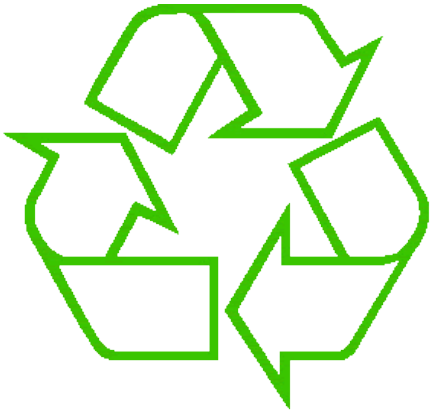


Control flow

Ruth Anderson

UW CSE 160

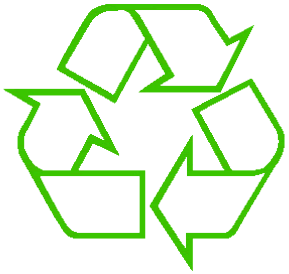
Winter 2016



Repeating yourself

Making decisions

Temperature conversion chart



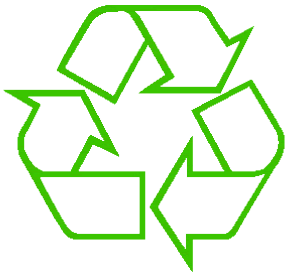
Recall exercise from previous lecture

```
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:

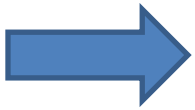
```
30 -1.11
40 4.44
50 10.0
60 15.56
70 21.11
All done
```

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: 1 4 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

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

The body can be multiple statements

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

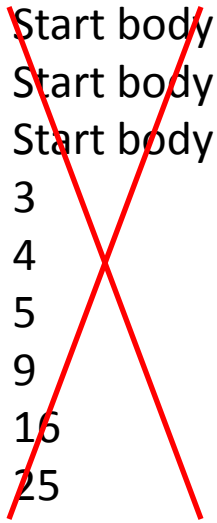
```
for i in [3,4,5]:
  print "Start body"
  print i
  print i*i
```

loop body:
3 statements

Output:

```
Start body
3
9
Start body
4
16
Start body
5
25
```

NOT:



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"
```

Nested Loops

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
```

What is the output?

Nested Loops

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

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
```

What is the output?

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
```

Test your understanding of loops

Output:

Puzzle 1:

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

Puzzle 2:

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

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

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

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
```

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)` produces [0,1,2,3,4]

Lower limit (*inclusive*)

`range(1,5)` produces [1,2,3,4]

step (distance between elements)

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

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?

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

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

```
# 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

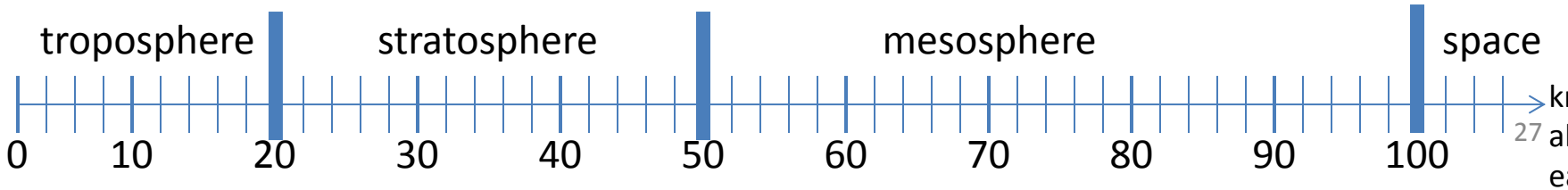
e

t

e

Execution gets here only if "height > 100" is 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

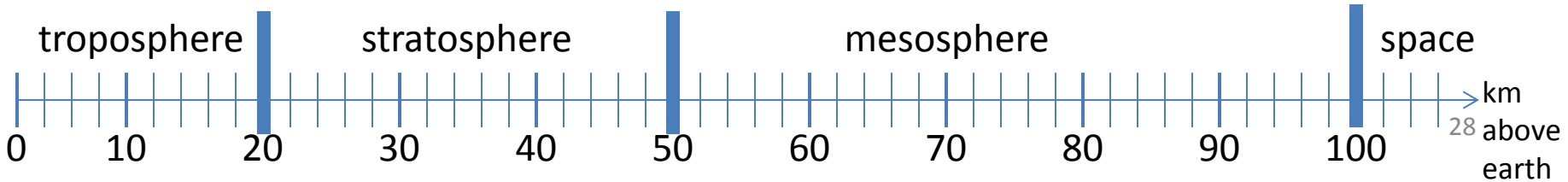
e

t

e

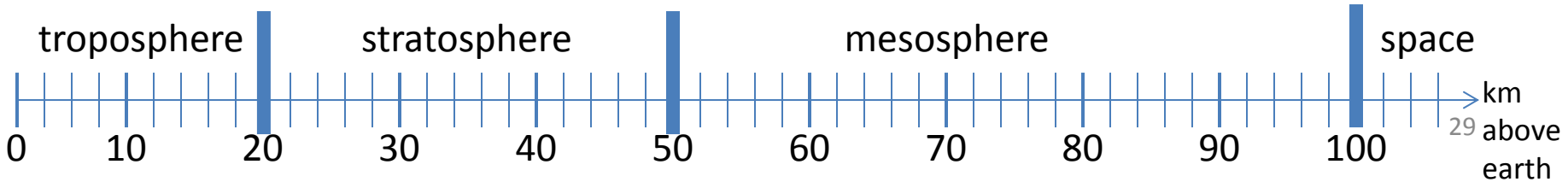
Execution gets here only if "height <= 100" is true

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



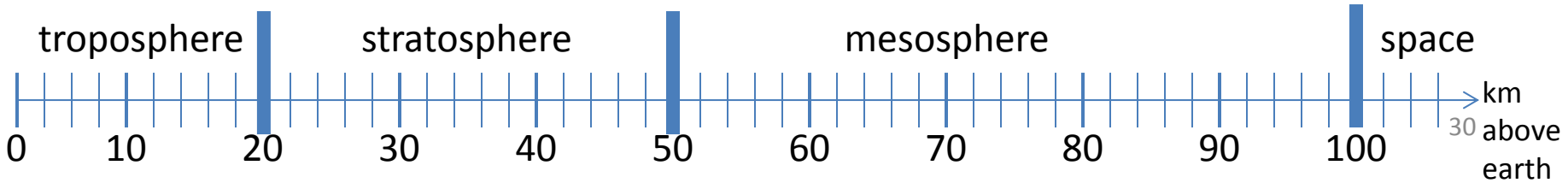
Version 1

```
# 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

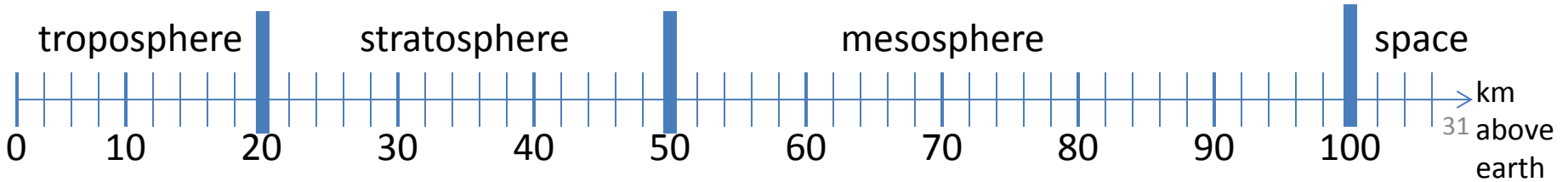
```
if height > 50:
  if height > 100:
    print "space"
  else:
    print "mesosphere"
else:
  if height > 20:
    print "stratosphere"
  else:
    print "troposphere"
```



Version 3 (Best)

```
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

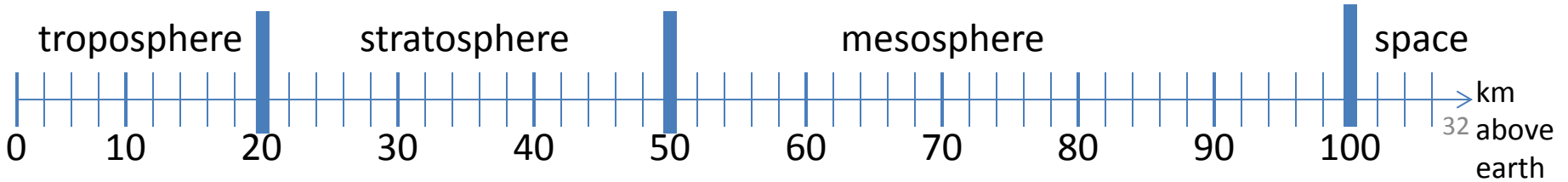


Order Matters

```
# 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"
```

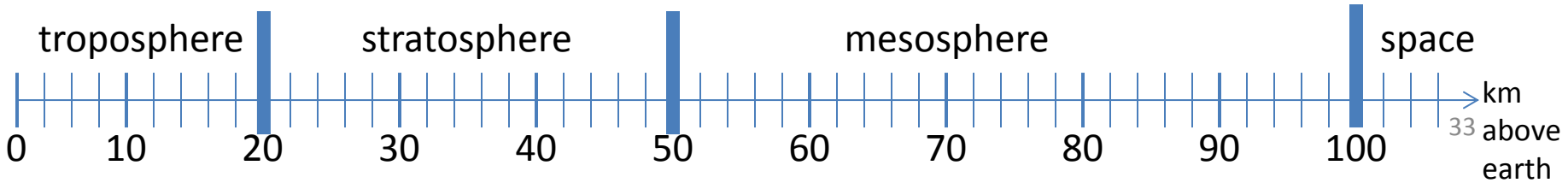
Try height = 72 on both versions, what happens?



Incomplete Version 3

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

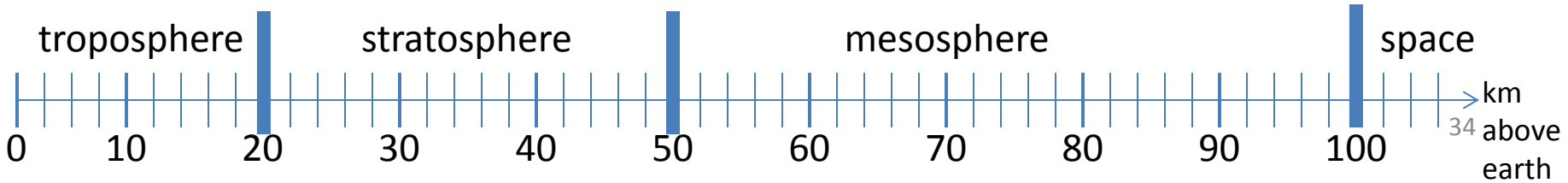
In this case it is possible that nothing is printed at all, when?



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?