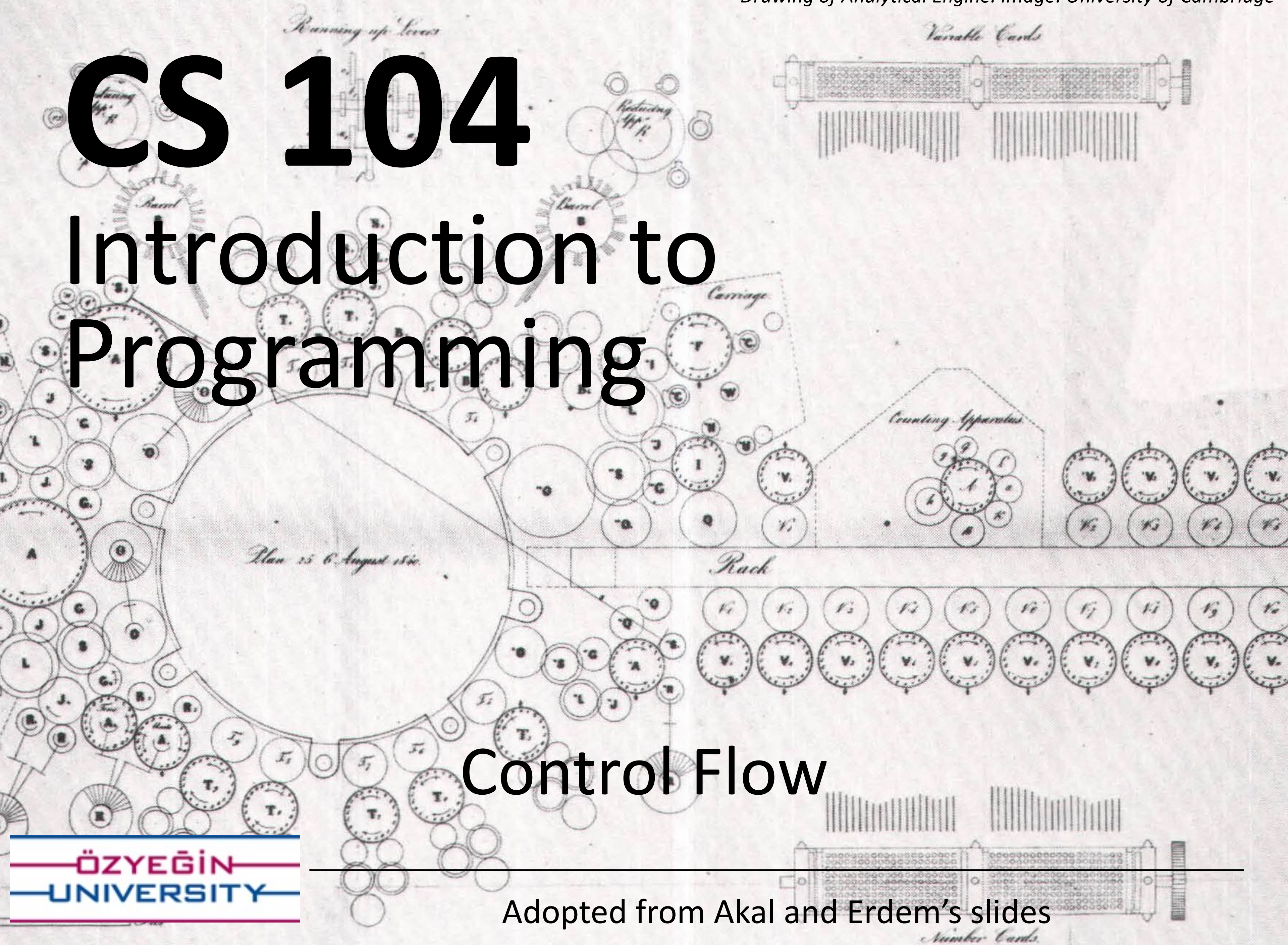


Drawing of Analytical Engine. Image: University of Cambridge

CS 104

Introduction to Programming

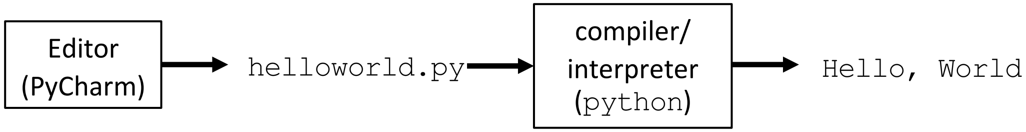
Control Flow

ÖZYEĞİN UNIVERSITY

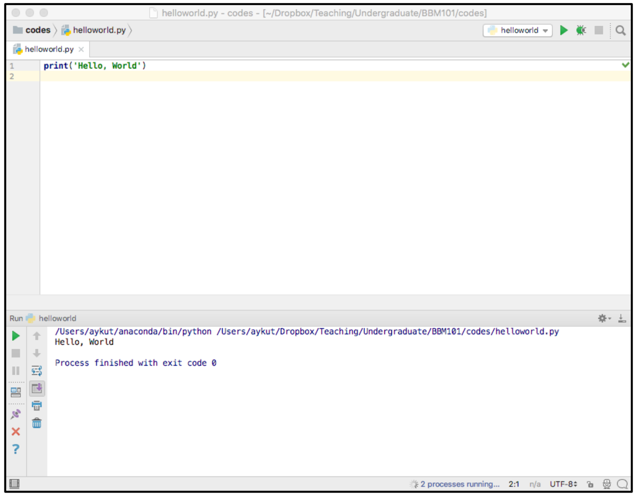
Adopted from Akal and Erdem's slides

Last time… **Introduction to Python**

Programming in Python



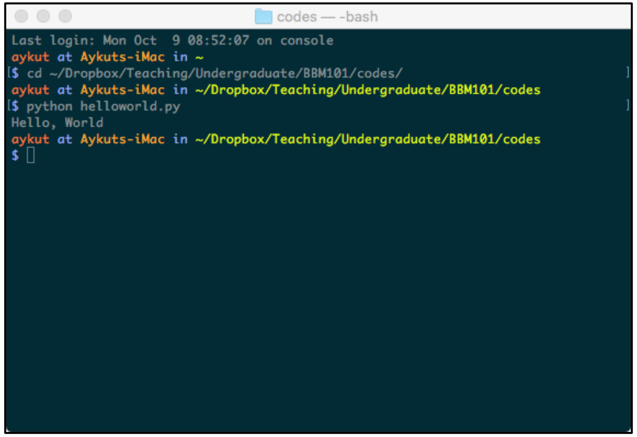
Editor (PyCharm) → helloworld.py → compiler/ interpreter (python) → Hello, World



```python
print('Hello, World')
```

```
/Users/aykut/anaconda/bin/python /Users/aykut/Dropbox/Teaching/Undergraduate/BBM101/codes/helloworld.py
Hello, World

Process finished with exit code 0
```

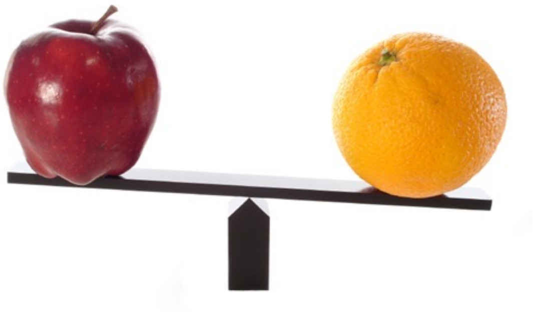


```
Last login: Mon Oct  9 08:52:07 on console
aykut at Aykuts-iMac in ~
[$ cd ~/Dropbox/Teaching/Undergraduate/BBM101/codes/
aykut at Aykuts-iMac in ~/Dropbox/Teaching/Undergraduate/BBM101/codes
[$ python helloworld.py
Hello, World
aykut at Aykuts-iMac in ~/Dropbox/Teaching/Undergraduate/BBM101/codes
$
```

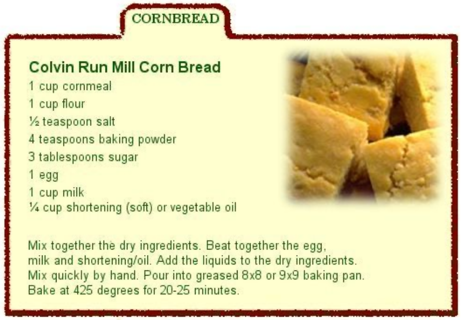
1. Python is like a calculator

2. A variable is a container

3. Different types cannot be compared



4. A program is a recipe



CORNBREAD

Colvin Run Mill Corn Bread

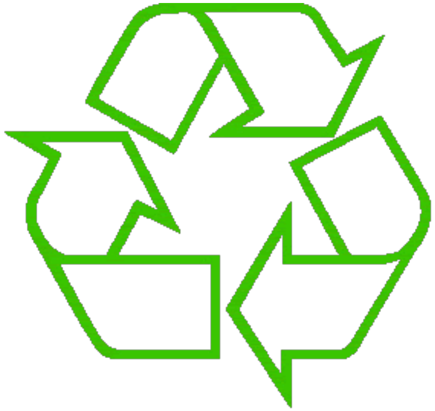
1 cup cornmeal
1 cup flour
½ teaspoon salt
4 teaspoons baking powder
3 tablespoons sugar
1 egg
1 cup milk
¼ cup shortening (soft) or vegetable oil

Mix together the dry ingredients. Beat together the egg, milk and shortening/oil. Add the liquids to the dry ingredients. Mix quickly by hand. Pour into greased 8x8 or 9x9 baking pan. Bake at 425 degrees for 20-25 minutes.

Lecture Overview

- Control Flow

Disclaimer: Much of the material and slides for this lecture were borrowed from
—Ruth Anderson, Michael Ernst and Bill Howe's CSE 140 class



Repeating yourself

Making decisions

Temperature Conversion Chart

Recall the exercise from the previous lecture

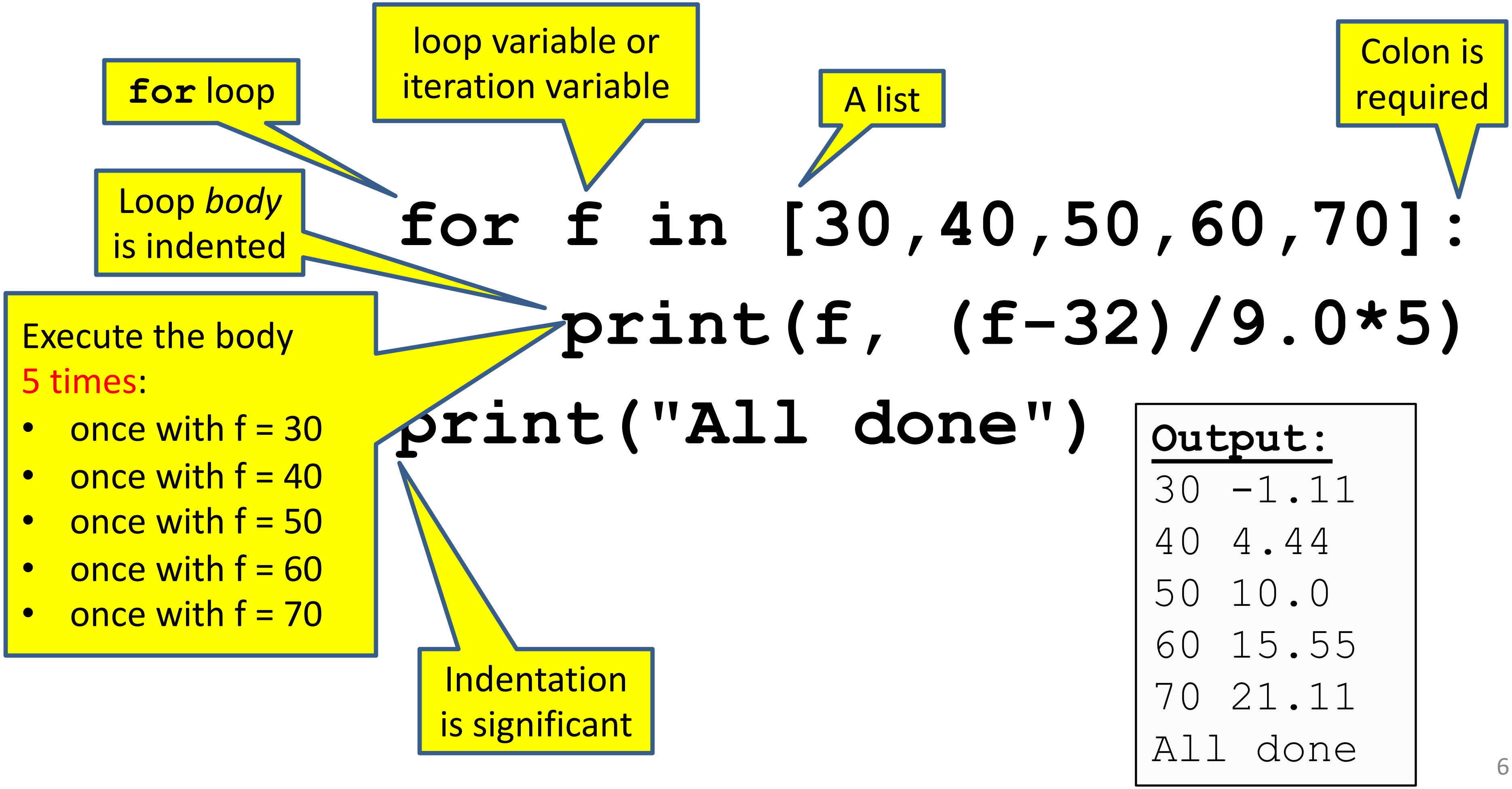
```
fahr = 30
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 40
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 50
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 60
cent = (fahr -32)/9.0*5
print(fahr, cent)
fahr = 70
cent = (fahr -32)/9.0*5
print(fahr, cent)
Print("All done")
```

Output:

```
30 -1.11
40 4.44
50 10.0
60 15.55
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
- once with f = 50
- once with f = 60
- once with f = 70

Indentation is significant

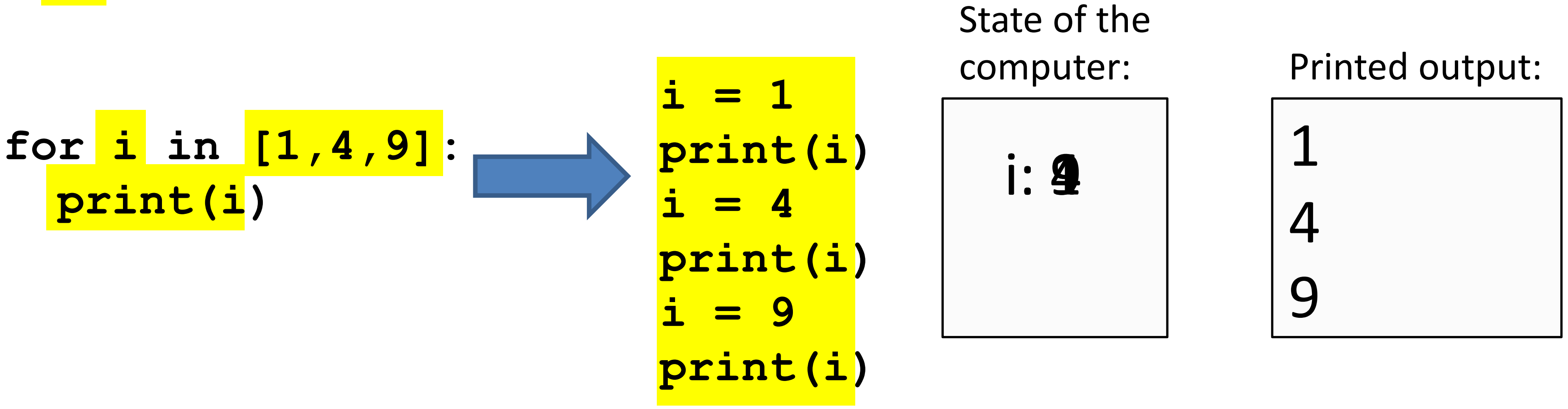
Output:

```
30 -1.11
40 4.44
50 10.0
60 15.55
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: 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:

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:

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

NOT:

~~Start body~~
~~Start body~~
~~Start body~~
~~3~~
~~4~~
~~5~~
~~9~~
~~16~~
~~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 in Loop 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)
```

loop body:
3 statements

“nested”
loop body:
2 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

```
# Goal:  print 1, 2, 3, …, 48, 49, 50

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)
```

- Analyze each of the above

Test Your Understanding of Loops

Output:

Puzzle 1:

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

```
0
1
1
```

Puzzle 2:

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

```
(no output)
```

Puzzle 3:

Reusing loop variable (don't do this!)

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

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

As an implicit 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)

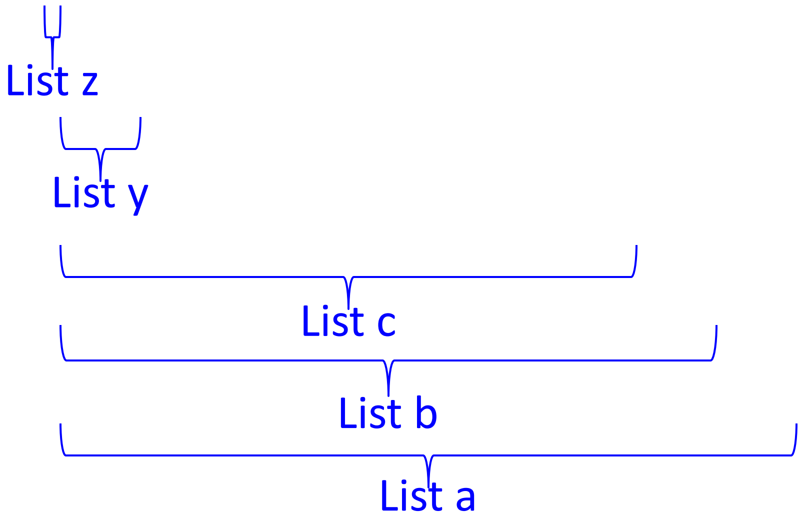
`range(1,10,2)` = [1,3,5,7,9]

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]



List z

List y

List c

List b

List a

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): # [0,1,2,3,4]
  result = result + element
print("The sum is: " + str(result))
```

```
The sum is: 10
```

```
# 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)
```

```
5, 4, 3, 2
The sum is: 14
```

```
# 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)
```

```
0, 2, 4, 6
The sum is: 12
```

```
# 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) + ", the result is " + str(result))
```

```
0, 1, 2, 3, 4
When size = 5, the result is 10
```

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 + … = $(2/3)^0 + (2/3)^1 + (2/3)^2 + (2/3)^3 + … + (2/3)^{10}$

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

Exercise with Loops

- Write a simple program to add values between two given inputs a, b
- e.g., if a=5, b=9, it returns sum of (5+6+7+8+9)
- Hint: we did some 'algorithmic thinking' and 'problem solving' here!

```
a, b = 5, 9
total = 0
for x in range(a, b+1):
    total += x
print(total)
```

Notice this form of the assignment statement!

Another Type of Loops – **while**

- The **while** loop is used for repeated execution as long as an expression is true

```
n = 100
s = 0
counter = 1
while counter <= n:
    s = s + counter
    counter += 1

print("Sum of 1 until ” + str(n) + ": ” + str(s))
```

```
Sum of 1 until 100: 5050
```

Question Break!

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 inside an if 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?

```python
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

f

t

f

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

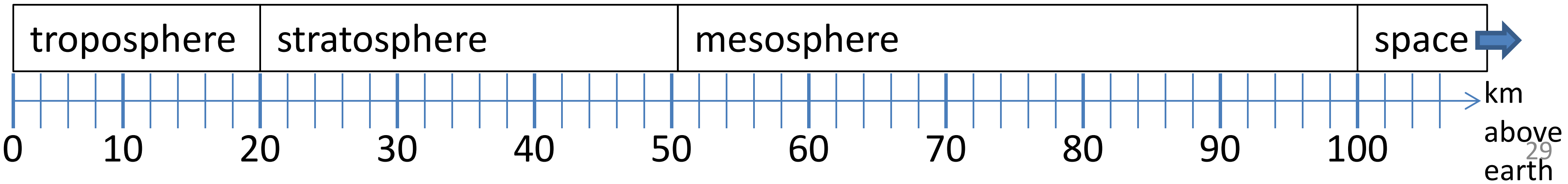
Execution gets here only if “height > 100” is false AND “height > 50” is true

Written differently!

Written differently, but more efficient!

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

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



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

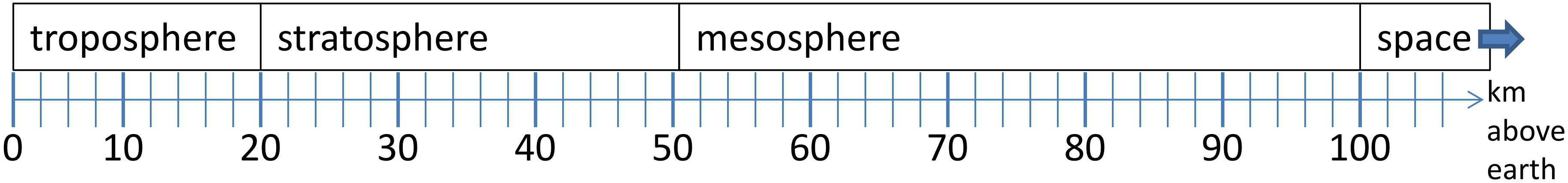
e

t

e

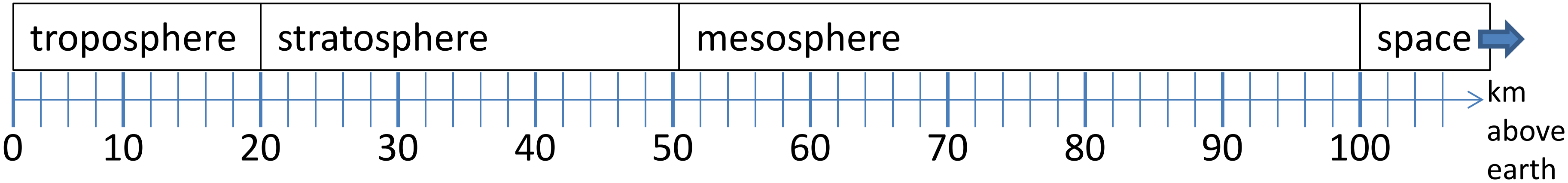
Execution gets here only if “height <= 100” is true

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



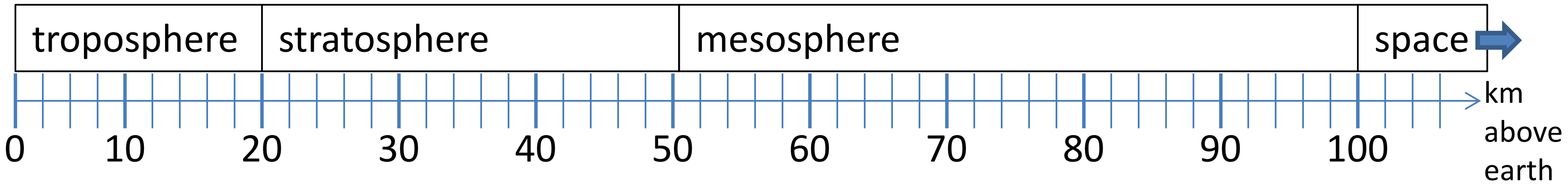
Version 1

```
# 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

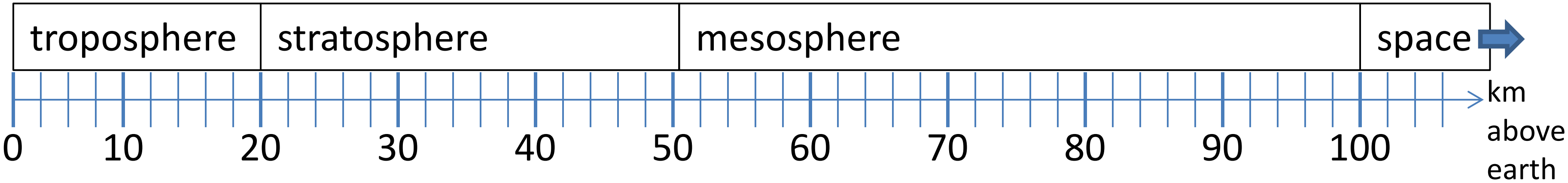
```
if height > 50:
  if height > 100:
    print("space")
  else:
    print("mesosphere")
else:
  if height > 20:
    print("stratosphere")
  else:
    print("troposphere")
```



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

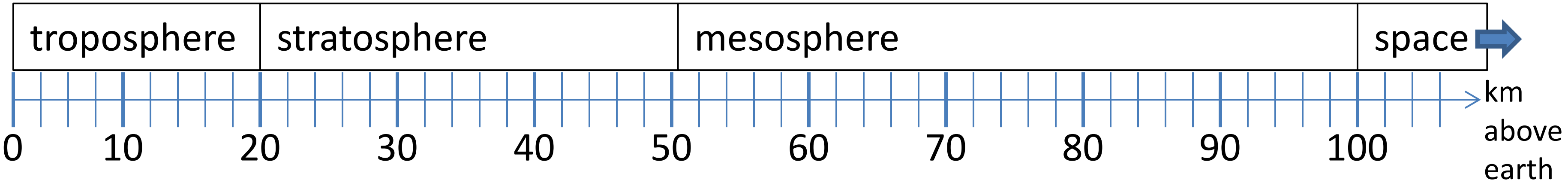


Order Matters

```
# 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")
```

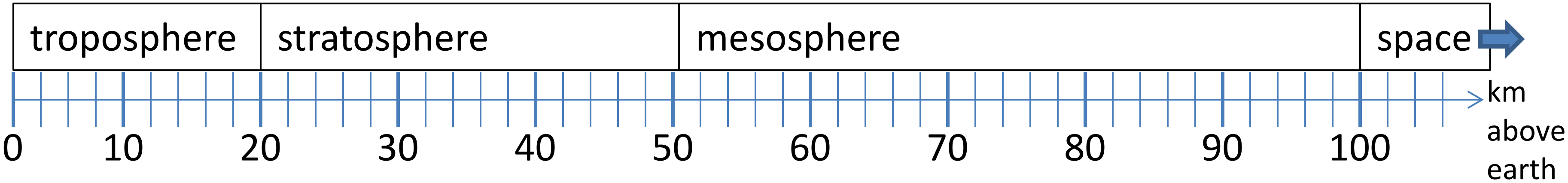
Try height = 72 on both versions, what happens?



Version 3

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

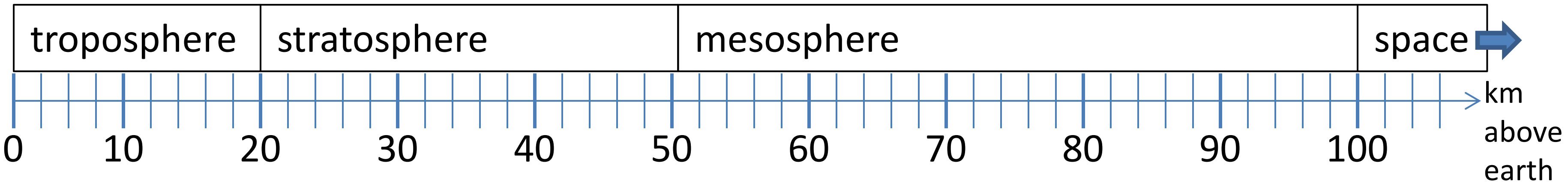
In this case it is possible that nothing is printed at all, when?



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



divisorpattern.py: Accept integer command-line argument *n*. Write to standard output an *n*-by-*n* table with an asterisk in row i and column j if either i divides j or j divides i.

```
import sys

n = int(sys.argv[1])
for i in range(1, n + 1):
    for j in range(1, n + 1):
        if (i % j == 0) or (j % i == 0):
            print(‘* ’, end=‘’)
        else:
            print(‘  ’, end=‘’)
    print(i)
```

```
$ python divisorpattern.py 3
* * * 1
* *   2
*   * 3
```

```
$ python divisorpattern.py 10
* * * * * * * * * * 1
* *   *   *   *   * 2
*   *     *     *   3
* *   *       *     4
*       *         * 5
* * *     *         6
*           *       7
* *   *       *     8
*   *           *   9
* *     *         * 10
```

Variable trace (`n = 3`)

```
i    j    output
----------------
1    1    '* '
1    2    '* '
1    3    '* 1\n'
2    1    '* '
2    2    '* '
2    3    '  2\n'
3    1    '* '
3    2    '  '
3    3    '* 3\n'
```

The **break** Statement

- The **break** statement terminates the current loop and resumes execution at the next statement

```
for letter in 'hollywood':
   if letter == 'l':
      break
   print ('Current Letter :', letter)
```

```
Current Letter : h
Current Letter : o
```

The **continue** Statement

- The **continue** statement in Python returns the control to the beginning of the while loop.

```
for letter in 'hollywood':
   if letter == 'l':
      continue
   print ('Current Letter :', letter)
```

```
Current Letter : h
Current Letter : o
Current Letter : y
Current Letter : w
Current Letter : o
Current Letter : o
Current Letter : d
```