

# BBM 101

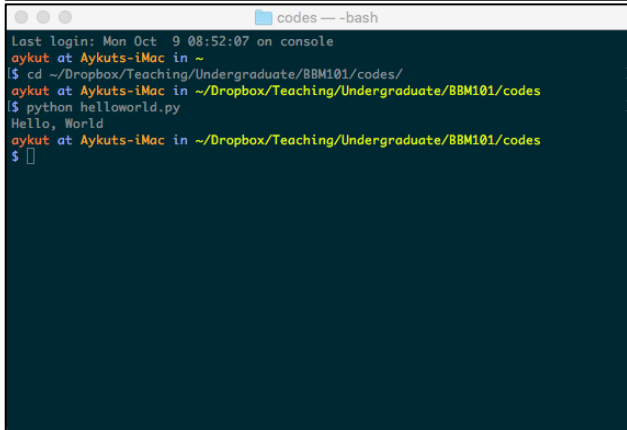
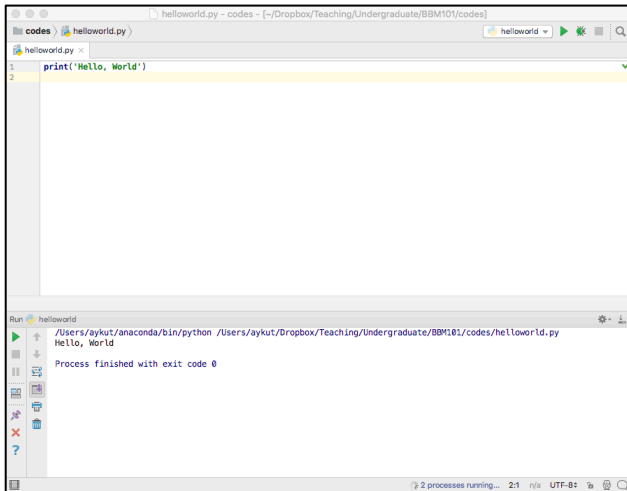
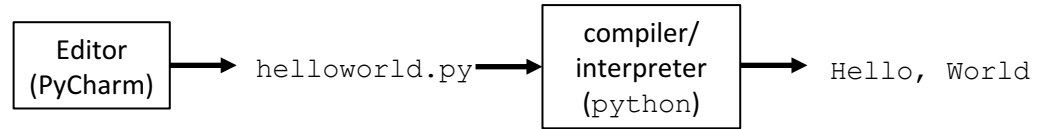
## Introduction to Programming I

Lecture #04 – Control Flow, Functions



# Last time... How to build computers

## Programming in Python



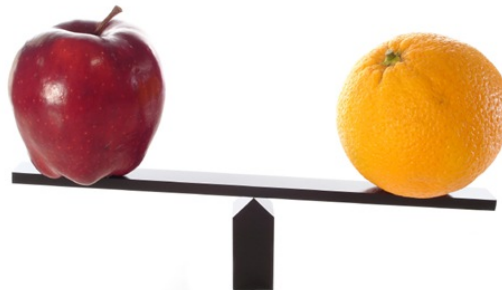
1. Python is like a calculator



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
- 1/2 teaspoon salt
- 4 teaspoons baking powder
- 3 tablespoons sugar
- 1 egg
- 1 cup milk
- 1/4 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.



# Administrative

- Your midterm exam will be held next week on October 25, Wednesday, 13:00.
- Your exam will cover all the material covered in the lectures and the corresponding chapters in the textbooks (including this week's material).
- Topics include:
  1. What is computation?
  2. Binary representations and the Von Neumann architecture
  3. Introduction to Python and Programming
  4. Control flow
  5. Functions

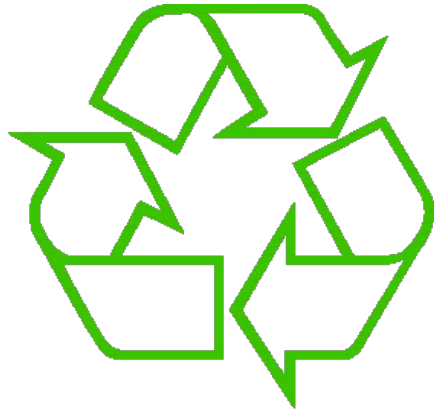
# Lecture Overview

- Control Flow
- Functions

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

# Lecture Overview

- Control Flow
- Functions

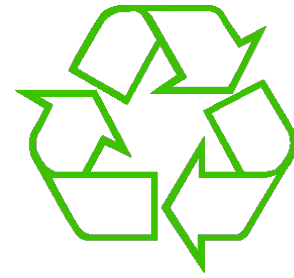


Repeating yourself



Making decisions

# Temperature Conversion Chart



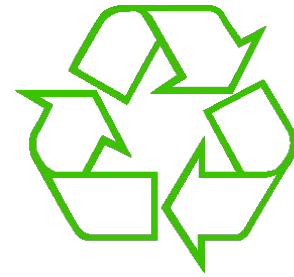
Recall the exercise from the previous lecture

```
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` loop

loop variable or iteration variable

A list

Colon is required

Loop *body* is indented

```
for f in [30, 40, 50, 60, 70]:
```

Execute the body  
**5 times:**

- once with  $f = 30$
- once with  $f = 40$
- once with  $f = 50$
- once with  $f = 60$
- once with  $f = 70$

```
    print(f, (f-32)/9.0*5)
```

```
    print("All done")
```

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

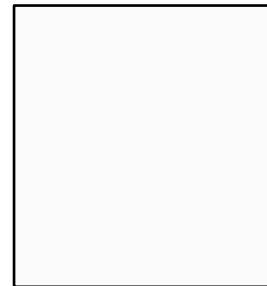
1. Evaluate the sequence expression
2. While there are sequence elements left:
  - a) Assign the loop variable to the next remaining sequence element
  - b) Execute the loop body

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

Current location in list



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")
```

Error!  print(i)  
print(i\*i)

- 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

```
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)
```

# 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)
```

```
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
```

- Analyze each of the above



# 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

As an implicit list:

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

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

```
    ... body ...
```

Upper limit  
(*exclusive*)

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

Lower limit  
(*inclusive*)

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

step (distance  
between elements)

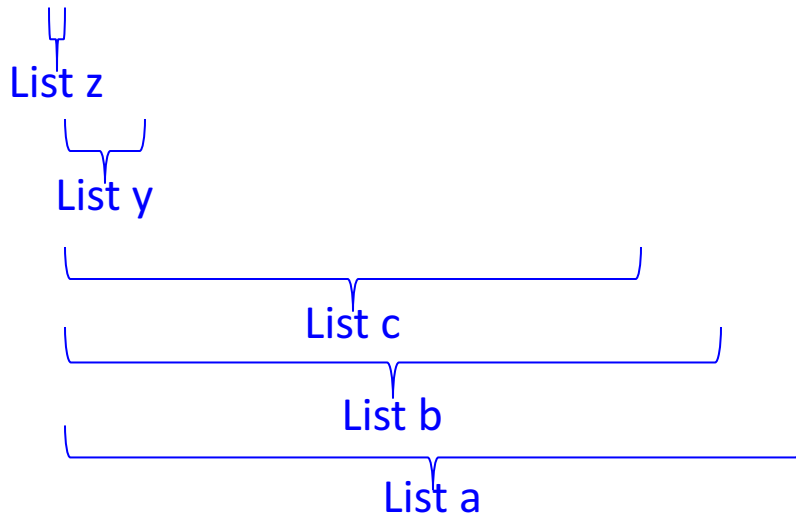
```
range(1, 10, 2) = [1,3,5,7,9]
```

# 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 ]



$\text{sum}(\text{List a}) = \text{sum}(\text{List b}) + 5$   
 $\text{sum}(\text{List b}) = \text{sum}(\text{List c}) + 6$   
...  
 $\text{sum}(\text{List y}) = \text{sum}(\text{List z}) + 3$   
 $\text{sum}(\text{empty list}) = 0$

# 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  $1 + 2/3 + 4/9 + 8/27 + 16/81 + \dots = (2/3)^0 + (2/3)^1 + (2/3)^2 + (2/3)^3 + \dots + (2/3)^{10}$

```
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!

Notice this form of the assignment statement!

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

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'



# 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 %d: %d" % (n,s))
```

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

# Making Decisions



- How do we compute absolute value?

$$\text{abs}(5) = 5$$

$$\text{abs}(0) = 0$$

$$\text{abs}(-22) = 22$$

# 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?

```
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

Written differently! but more efficient!

```
# height is in km
if height > 100:
    print("space")
else:
```

then  
clause {

Execution gets here only  
if "height > 100" is false

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

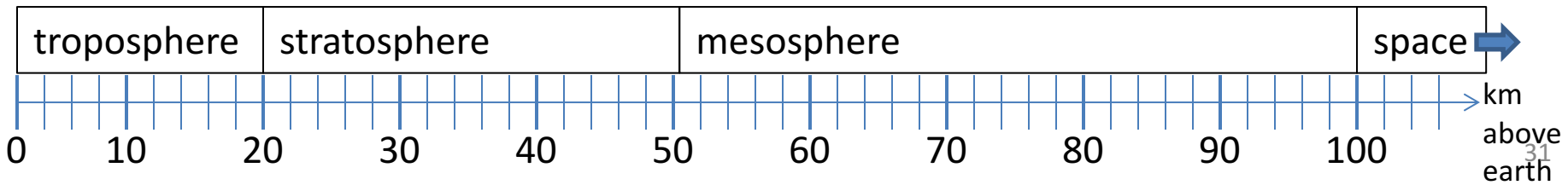
else  
clause {

f {

```
# height is in km
if height > 100:
    print("space")
```

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

```
elif height > 50:
    print("mesosphere")
elif height > 20:
    print("stratosphere")
else:
    print("troposphere")
```



# Version 1

```
# height is in km
```

```
if height > 100:
```

then  
clause {

```
    print("space")
```

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

```
else:
```

```
    if height > 50:
```

else  
clause {

```
        t{ print("mesosphere")
```

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

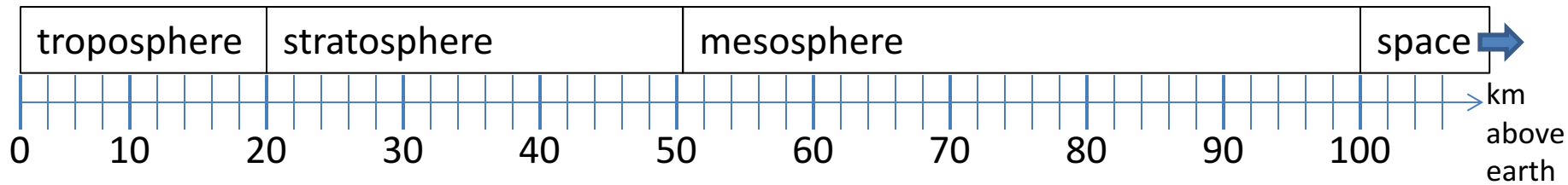
```
    else:
```

```
        if height > 20:
```

```
            t{ print("stratosphere")
```

```
            else:
```

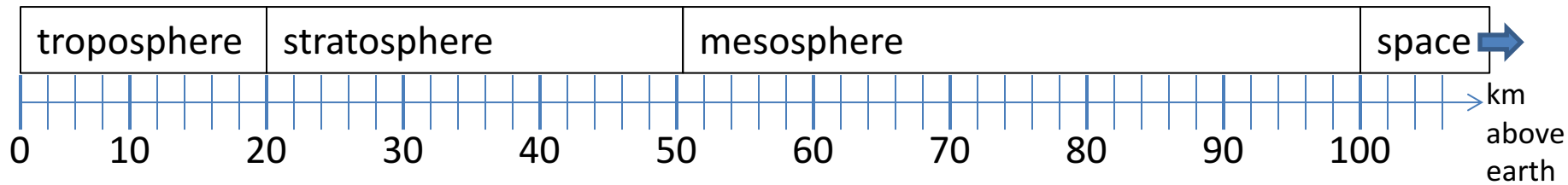
```
                e{ print("troposphere")
```





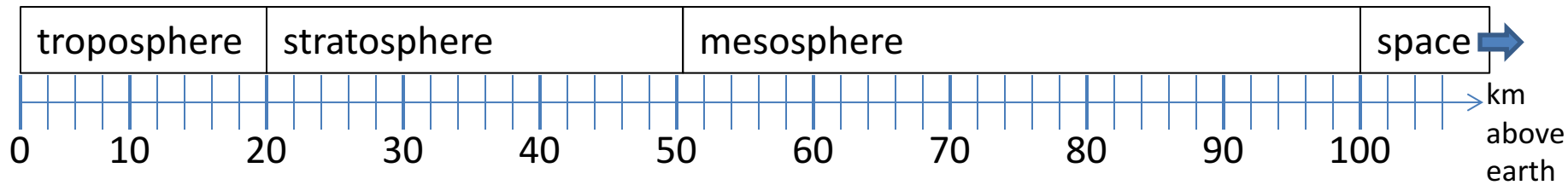
# Version 1

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



# Version 2

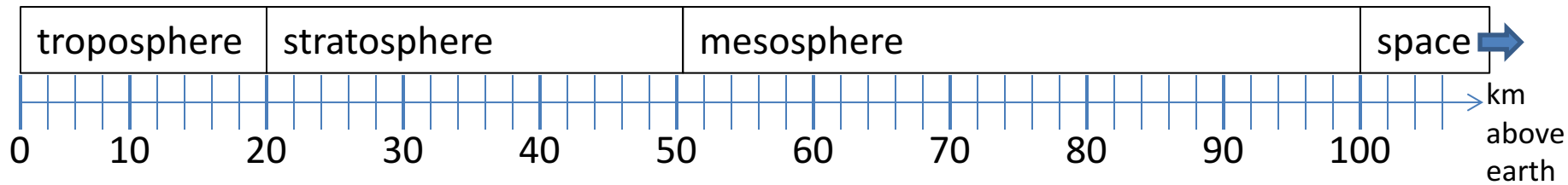
```
if height > 50:
    if height > 100:
        print("space")
    else:
        print("mesosphere")
else:
    if height > 20:
        print("stratosphere")
    else:
        print("troposphere")
```



# 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

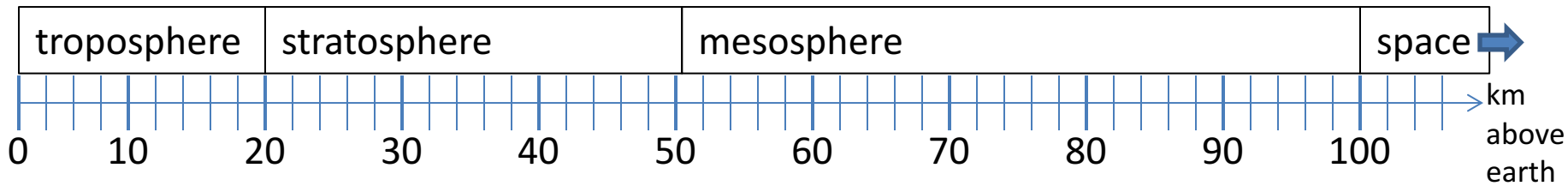


# Order Matters

```
# 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")
```

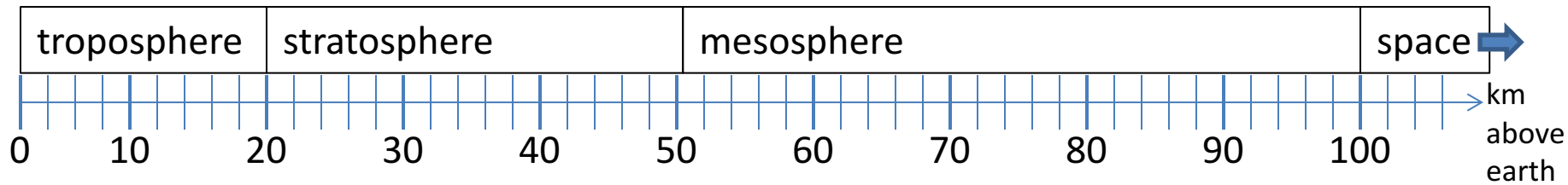
Try height = 72 on both versions, what happens?



# Version 3

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

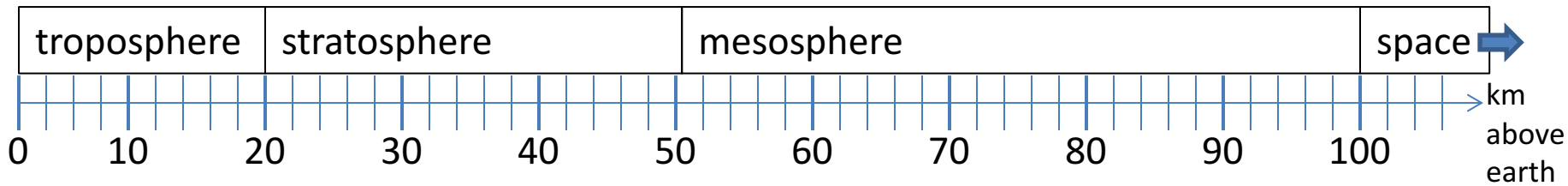
In this case it is possible that nothing is printed at all, when?



# 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 = 65
```

```
limit = 70
```

```
if speed <= limit:
```

```
    print("Good job, safe driver!")
```

```
else:
```

```
    print("You owe $", speed/fine)
```



What if we change speed to 75?

# 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)
```

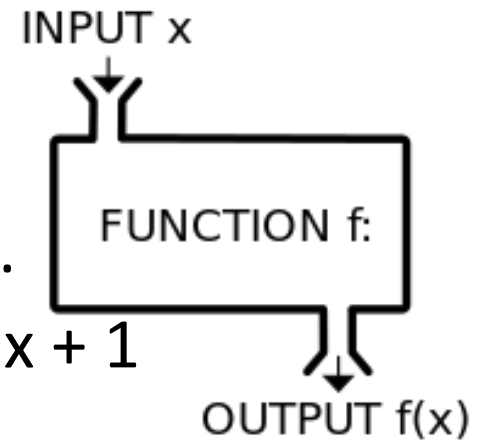
```
Current Letter : h  
Current Letter : o  
Current Letter : y  
Current Letter : w  
Current Letter : o  
Current Letter : o  
Current Letter : d
```

# Lecture Overview

- Control Flow
- Functions

# Functions

- In math, you **use** functions: sine, cosine, ...
- In math, you **define** functions:  $f(x) = x^2 + 2x + 1$



- A function packages up and names a computation
- Enables re-use of the computation (generalization)
- **Don't Repeat Yourself** (DRY principle)
- Shorter, easier to understand, less error-prone
  
- Python lets you **use** and **define** functions
- We have already seen some Python functions:
  - **len, float, int, str, range**

# Using (“calling”) a Function

```
len("hello")
```

```
len("")
```

```
round(2.718)
```

```
round(3.14)
```

```
pow(2, 3)
```

```
range(1, 5)
```

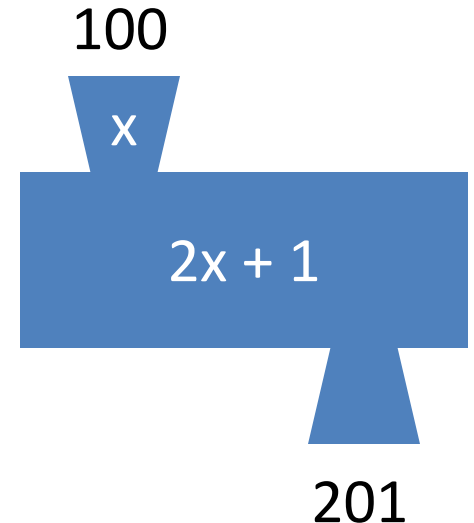
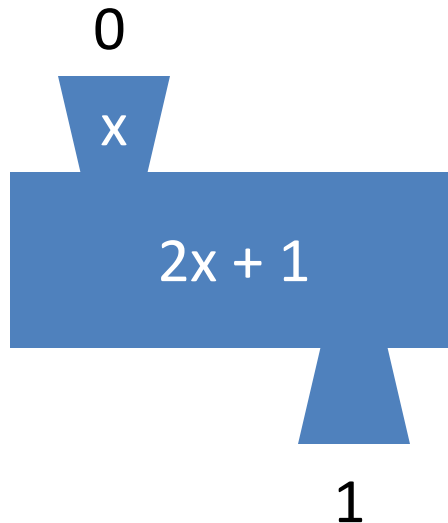
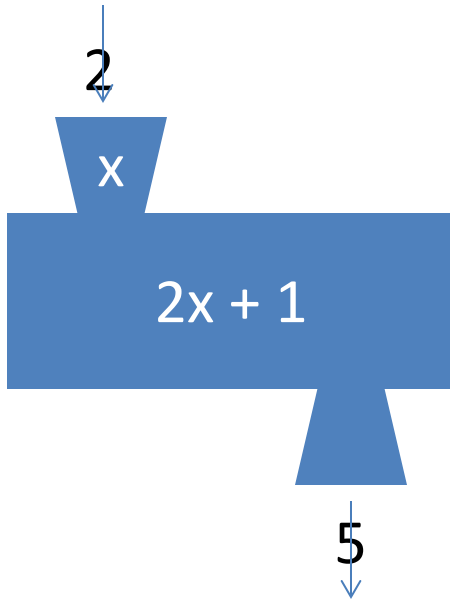
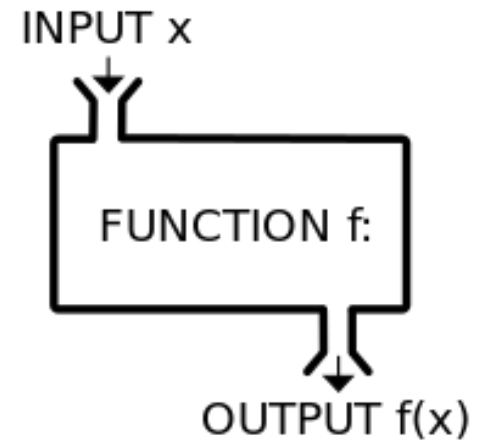
```
math.sin(0)
```

```
math.sin(math.pi / 2)
```

- Some need no input:  
`random.random()`
- All produce output

# A Function is a Machine

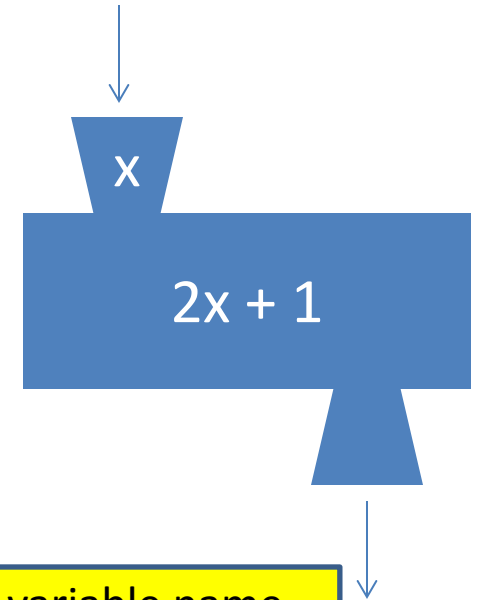
- You give it input
- It produces a result (output)



In math:  $\text{func}(x) = 2x + 1$

# Creating a Function

Define the machine,  
including the input and the result



Name of the function.  
Like “ $y = 5$ ” for a variable

Keyword that means:  
I am **def**ining a function

Input variable name,  
or “formal parameter”

```
def dbl_plus(x) :
```

```
    return 2*x + 1
```

Keyword that means:  
This is the result

Return expression  
(part of the **return** statement)

# More Function Examples

Define the machine, including the input and the result

```
def square(x):  
    return x * x
```

```
def fahr_to_cent(fahr):  
    return (fahr - 32) / 9.0 * 5
```

```
def cent_to_fahr(cent):  
    result = cent / 5.0 * 9 + 32  
    return result
```

```
def abs(x):  
    if x < 0:  
        return - x  
    else:  
        return x
```



```
def print_hello():  
    print("Hello, world")
```

No return statement  
Returns the value None  
Are also called 'procedures'

```
def print_fahr_to_cent(fahr):  
    result = fahr_to_cent(fahr)  
    print(result)
```

What is the result of:

```
x = 42  
square(3) + square(4)  
print(x)  
boiling = fahr_to_cent(212)  
cold = cent_to_fahr(-40)  
print(result)  
print(abs(-22))  
print(print_fahr_to_cent(32))
```

# Python Interpreter

- An expression evaluates to a value
  - Which can be used by the containing expression or statement
- **print("test")** statement writes text to the screen
- The Python interpreter (command shell) reads statements and expressions, then executes them
- If the interpreter executes an expression, it prints its value
- In a program, evaluating an expression does not print it
- In a program, printing an expression does not permit it to be used elsewhere



# An example

```
def lyrics():  
    print("The very first line")  
print(lyrics())
```

```
The very first line  
None
```

# How Python Executes a Function Call

Function definition

```
def square(x):  
    return x * x
```

Formal parameter  
(a variable)

square(3 + 4)

Actual argument

Function call or  
function invocation

Current expression:

1 + square(3 + 4)

1 + square(7)

evaluate this expression

1 + 49

50

return x \* x

return 7 \* x

return 7 \* 7

return 49

Variables:

x: 7

1. Evaluate the **argument** (at the call site)
2. Assign the **formal parameter name** to the argument's value
  - A *new* variable, not reuse of any existing variable of the same name
3. Evaluate the **statements** in the body one by one
4. At a **return** statement:
  - Remember the value of the expression
  - Formal parameter variable disappears – exists only during the call!
  - The call expression evaluates to the return value

# Example of Function Invocation

```
def square(x):  
    return x * x
```

```
square(3) + square(4)
```

```
return x * x  
return 3 * x  
return 3 * 3  
return 9
```

```
9 + square(4)
```

```
return x * x  
return 4 * x  
return 4 * 4  
return 16
```

```
9 + 16
```

```
25
```

Variables:

(none)

x: 3

x: 3

x: 3

x: 3

(none)

x: 4

x: 4

x: 4

x: 4

(none)

(none)

# Expression with Nested Function Invocations: Only One Executes at a Time

```
def fahr_to_cent(fahr):  
    return (fahr - 32) / 9.0 * 5
```

```
def cent_to_fahr(cent):  
    return cent / 5.0 * 9 + 32
```

```
fahr_to_cent(cent_to_fahr(20))  
    return cent / 5.0 * 9 + 32  
    return 20 / 5.0 * 9 + 32  
    return 68
```

```
fahr_to_cent(68)  
return (fahr - 32) / 9.0 * 5  
return (68 - 32) / 9.0 * 5  
return 20  
20
```

## Variables:

```
(none)  
cent: 20  
cent: 20  
cent: 20  
(none)  
fahr: 68  
fahr: 68  
fahr: 68  
(none)
```



# Expression with Nested Function Invocations: Only One Executes at a Time

```
def square(x):  
    return x * x
```

```
square(square(3))
```

```
    return x * x  
    return 3 * x  
    return 3 * 3  
    return 9
```

```
square(9)
```

```
    return x * x  
    return 9 * x  
    return 9 * 9  
    return 81
```

81

Variables:

(none)

x=3

x=3

x=3

x=3

(none)

x=9

x=9

x=9

x=9

(none)

# Function that Invokes Another Function: Both Function Invocations are Active

```
import math

def square(z):
    return z*z

def hypoten_use(x, y):
    return math.sqrt(square(x) + square(y))
```

```
hypoten_use(3, 4)
    return math.sqrt(square(x) + square(y))
    return math.sqrt(square(3) + square(y))
        return z*z
        return 3*3
        return 9
    return math.sqrt(9 + square(y))
    return math.sqrt(9 + square(4))
        return z*z
        return 4*4
        return 16
    return math.sqrt(9 + 16)
    return math.sqrt(25)
    return 5
```

## Variables:

```
(none)
x:3 y:4
x:3 y:4
z: 3
z: 3
z: 3
x: 3 y:4
x: 3 y:4
z: 4
z: 4
z: 4
x: 3 y:4
x: 3 y:4
x:3 y:4
(none)
```

# Shadowing of Formal Variable Names

```
import math
def square(x):
    return x*x
def hypotenuse(x, y):
    return math.sqrt(square(x) + square(y))
```

Same formal parameter name

```
hypotenuse(3, 4)
return math.sqrt(square(x) + square(y))
return math.sqrt(square(3) + square(y))
return x*x
return 3*3
return 9
return math.sqrt(9 + square(y))
return math.sqrt(9 + square(4))
return x*x
return 4*4
return 16
return math.sqrt(9 + 16)
return math.sqrt(25)
return 5
```

## Variables:

```
(none)
x: 3 y:4
x: 3 y:4
x: 3
x: 3
x: 3
x: 3 y:4
x: 3 y:4
x: 4
x: 4
x: 4
x: 3 y:4
x: 3 y:4
x:3 y:4
(none)
```

Formal parameter is a *new* variable

# Shadowing of Formal Variable Names

```
import math
def square(x):
    return x*x
def hypotenuse(x, y):
    return math.sqrt(square(x) + square(y))

hypotenuse(3, 4)
return math.sqrt(square(x) + square(y))
return math.sqrt(square(3) + square(y))
    return x*x
    return 3*3
    return 9
return math.sqrt(9 + square(y))
return math.sqrt(9 + square(4))
    return x*x
    return 4*4
    return 16
return math.sqrt(9 + 16)
return math.sqrt(25)
return 5
```

Same diagram, with *variable scopes* or *environment frames* shown explicitly

## Variables:

(none) hypotenuse()

square()

x: 3

x: 3

x: 3

square()

x: 4

x: 4

x: 4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

x:3 y:4

(none)





# In a Function Body, Assignment Creates a Temporary Variable (like the formal parameter)

```
stored = 0
def store_it(arg):
    stored = arg
    return stored
```

★ `y = store_it(22)`

```
print(y)
```

★ `print(stored)`

Show evaluation of the starred expressions:

```
y = store_it(22)
    stored = arg; return stored
    stored = 22; return stored
    return stored
    return 22
```

```
y = 22
```

```
print(stored)
```

```
print(0)
```

Variables:

Global or  
top level

store\_it()

```
arg: 22
arg: 22
arg: 22 stored: 22
arg: 22 stored: 22
```

```
stored: 0
stored: 0
stored: 0
stored: 0 y: 22
stored: 0 y: 22
stored: 0 y: 22
```

# How to Look Up a Variable

Idea: find the nearest variable of the given name

1. Check whether the variable is defined in the **local scope**
2. ... check any intermediate scopes ...
3. Check whether the variable is defined in the **global scope**

If a local and a global variable have the **same name**, the global variable is inaccessible (“**shadowed**”)

This is confusing; try to avoid such shadowing

```
x = 22
stored = 100
def lookup():
    x = 42
    return stored + x
lookup()
x = 5
stored = 200
lookup()
```

```
def lookup():
    x = 42
    return stored + x
x = 22
stored = 100
lookup()
x = 5
stored = 200
lookup()
```

What happens if we define **stored** after **lookup**?

# Local Variables Exist Only while the Function is Executing

```
def cent_to_fahr(cent):  
    result = cent / 5.0 * 9 + 32  
    return result
```

```
tempf = cent_to_fahr(15)  
print(result)
```



# Use Only the Local and the Global Scope

```
myvar = 1
```

```
def outer():  
    myvar = 1000  
    return inner()
```

```
def inner():  
    return myvar
```

```
print(outer())
```



# Abstraction



- Abstraction = ignore some details
- Generalization = become usable in more contexts
- Abstraction over **computations**:
  - functional abstraction, a.k.a. procedural abstraction
- As long as you know what the function **means**, you don't care **how** it computes that value
  - You don't care about the *implementation* (the function body)

# Defining Absolute Value

```
def abs(x):  
    if val < 0:  
        return -1 * val  
    else:  
        return 1 * val
```

```
def abs(x):  
    if val < 0:  
        return - val  
    else:  
        return val
```

```
def abs(x):  
    if val < 0:  
        result = - val  
    else:  
        result = val  
    return result
```

```
def abs(x):  
    return math.sqrt(x*x)
```

# Defining Round (for positive numbers)

```
def round(x):  
    return int(x+0.5)
```

```
def round(x):  
    fraction = x - int(x)  
    if fraction >= .5:  
        return int(x) + 1  
    else:  
        return int(x)
```

# Each Variable Should Represent One Thing

```
def atm_to_mbar(pressure):  
    return pressure * 1013.25  
  
def mbar_to_mmHg(pressure):  
    return pressure * 0.75006  
  
# Confusing  
pressure = 1.2 # in atmospheres  
pressure = atm_to_mbar(pressure)  
pressure = mbar_to_mmHg(pressure)  
print(pressure)
```

```
# Better  
in_atm = 1.2  
in_mbar = atm_to_mbar(in_atm)  
in_mmHg = mbar_to_mmHg(in_mbar)  
print(in_mmHg)
```

```
# Best  
def atm_to_mmHg(pressure):  
    in_mbar = atm_to_mbar(pressure)  
    in_mmHg = mbar_to_mmHg(in_mbar)  
    return in_mmHg  
print(atm_to_mmHg(1.2))
```

Corollary: Each variable should contain values of only one type

```
# Legal, but confusing: don't do this!  
x = 3  
...  
x = "hello"  
...  
x = [3, 1, 4, 1, 5]  
...
```

If you use a descriptive variable name, you are unlikely to make these mistakes



# Exercises

```
def cent_to_fahr(c):  
    print(cent / 5.0 * 9 + 32)  
  
print(cent_to_fahr(20))
```

```
def myfunc(n):  
    total = 0  
    for i in range(n):  
        total = total + i  
    return total  
  
print(myfunc(4))
```

```
def c_to_f(c):  
    print("c_to_f")  
    return c / 5.0 * 9 + 32  
  
def make_message(temp):  
    print("make_message")  
    return ("The temperature is "  
+ str(temp))  
  
for tempc in [-40,0,37]:  
    tempf = c_to_f(tempc)  
    message = make_message(tempf)  
    print(message)
```

```
double(7)
```

```
abs(-20 - 2) + 20
```



# What Does This Print?

```
def myfunc(n):  
    total = 0  
    for i in range(n):  
        total = total + i  
    return total  
  
print(myfunc(4))
```

6

# What Does This Print?

```
def c_to_f(c):  
    print("c_to_f")  
    return c / 5.0 * 9 + 32  
  
def make_message(temp):  
    print("make_message")  
    return "The temperature is " + str(temp)  
  
for tempc in [-40, 0, 37]:  
    tempf = c_to_f(tempc)  
    message = make_message(tempf)  
    print(message)
```

```
c_to_f  
make_message  
The temperature is -40.0  
c_to_f  
make_message  
The temperature is 32.0  
c_to_f  
make_message  
The temperature is 98.6
```

# Decomposing a Problem

- Breaking down a program into functions is the fundamental activity of programming!
- How do you decide when to use a function?
  - One rule: DRY (Don't Repeat Yourself)
  - Whenever you are tempted to copy and paste code, don't!
- Now, how do you design a function?

# Review: How to Evaluate a Function Call

1. Evaluate the function and its arguments to values
  - If the function value is not a function, execution terminates with an error
2. Create a new stack frame
  - The parent frame is the one where the function is defined
  - A frame has bindings from variables to values
  - Looking up a variable starts here
    - Proceeds to the next older frame if no match here
    - The oldest frame is the “global” frame
    - All the frames together are called the “environment”
  - Assignments happen here
3. Assign the actual argument values to the formal parameter variable
  - In the new stack frame
4. Evaluate the body
  - At a return statement, remember the value and exit
  - If at end of the body, return **None**
5. Remove the stack frame
6. The call evaluates to the returned value

# Functions are Values: The Function can be an Expression

```
import math

def double(x):
    return 2*x
print(double)
myfns = [math.sqrt, int, double, math.cos]
myfns[1](3.14)
myfns[2](3.14)
myfns[3](3.14)

def doubler():
    return double
doubler()(2.718)
```



# Nested Scopes

- In Python, one can always determine the scope of a name by looking at the program text.
  - **static** or **lexical scoping**

```
def f(x):  
    def g():  
        x = "abc"  
        print("x =", x)  
    def h():  
        z = x  
        print("z =", z)  
    x = x+1  
    print("x =", x)  
    h()  
    g()  
    print("x =", x)  
    return g
```

```
x = 3  
z = f(x)  
print("x =", x)  
print("z =", z)  
z()
```

```
x = 4  
z = 4  
x = abc  
x = 4  
x = 3  
z = <function f.<locals>.g at  
0x7f06d7fa2ea0>  
x = abc
```



# Two Types of Documentation

1. Documentation for **users/clients/callers**
  - Document the *purpose* or *meaning* or *abstraction* that the function represents
  - Tells **what** the function does
  - Should be written for *every* function
2. Documentation for **programmers** who are reading the code
  - Document the *implementation* – specific code choices
  - Tells **how** the function does it
  - Only necessary for tricky or interesting bits of the code

For **users**: a string as the first element of the function body

For **programmers**: arbitrary text after #

called  
docstring

```
def square(x):  
    """Returns the square of its argument."""  
    # "x*x" can be more precise than "x**2"  
    return x*x
```



# Multi-line Strings

- New way to write a string – surrounded by three quotes instead of just one
  - `"hello"`
  - `'hello'`
  - `"""hello"""`
  - `'''hello'''`
- Any of these works for a documentation string
- Triple-quote version:
  - can include newlines (carriage returns), so the string can span multiple lines
  - can include quotation marks

# Don't Write Useless Comments

- Comments should give information that is not apparent from the code
- Here is a counter-productive comment that merely clutters the code, which makes the code *harder* to read:

```
# increment the value of x  
x = x + 1
```

# Where to Write Comments

- By convention, write a comment *above* the code that it describes (or, more rarely, on the same line)
  - First, a reader sees the English intuition or explanation, then the possibly-confusing code

```
# The following code is adapted from
# "Introduction to Algorithms", by Cormen et al.,
# section 14.22.
while (n > i):
    ...
```

- A comment may appear anywhere in your program, including at the end of a line:

```
x = y + x      # a comment about this line
```

- For a line that starts with #, indentation must be consistent with surrounding code