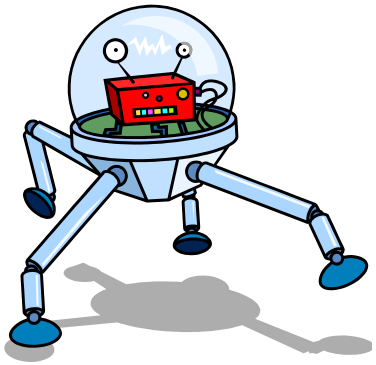
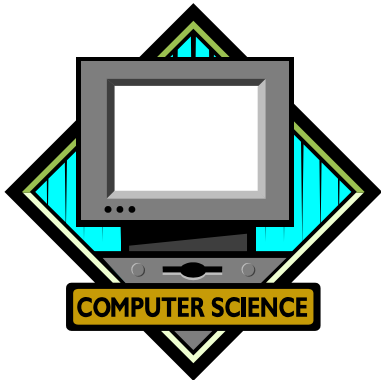
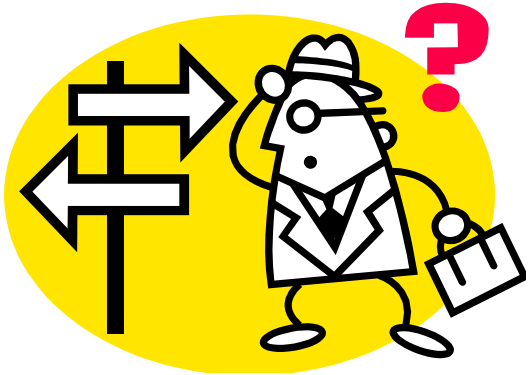
Caltech/LEAD Summer 2012
Computer Science





Lecture 4: July 11, 2012

Lists, loops and decisions



Today

- Lists
- Looping with the `for` statement
- Making decisions with the `if` statement

Lists

- A *list* is a sequence of Python values
  - a way to take multiple values and create a single value that contains all the other values
  - Recall: strings are a sequence of characters
  - Lists are a sequence of *any* kind of value

Why lists?

- Often have many related values that you'd like to store in a single object
  - *e.g.* average temperature each day for the last week
- Could define separate variables for each value
  - but it would quickly become tedious

Without lists

```
temp_sunday    = 59.6
temp_monday    = 72.4
temp_tuesday   = 68.5
temp_wednesday = 79.0
temp_thursday  = 66.4
temp_friday    = 77.1
temp_saturday  = -126.0  # new ice age?
```

- Hard to use this for anything

Without lists

```
avg_temp = (temp_sunday + ...) / 7.0
```

- Tedious, inflexible

With lists

```
temps = [59.6, 72.4, 68.5, 79.0,
         66.4, 77.1, -126.0]
```

- Much simpler, easier to work with:

```
avg_temp = sum(temps) / 7.0
```

- Makes working with multiple values as easy as working with single ones
- Lists are used everywhere in Python code!

Creating lists

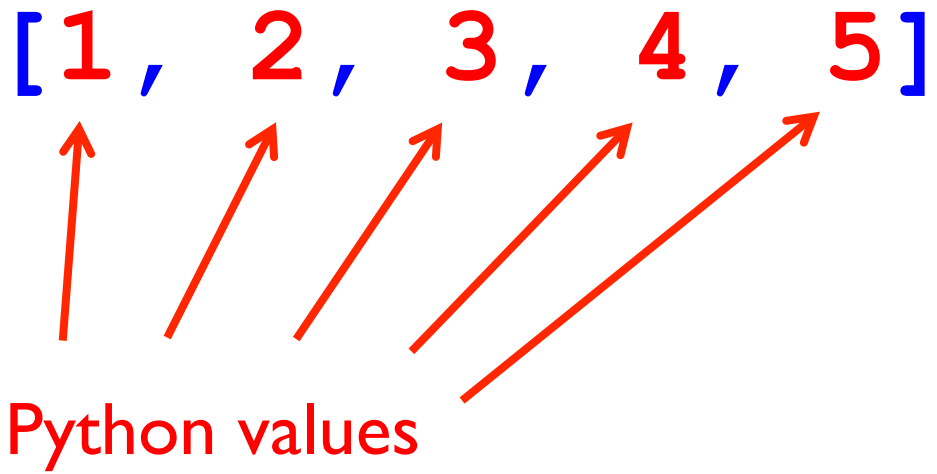
- Create lists by putting Python values or expressions inside square brackets, separated by commas:

```
[1, 2, 3, 4, 5]
```

- Items inside list are called the *elements* of the list

Creating lists

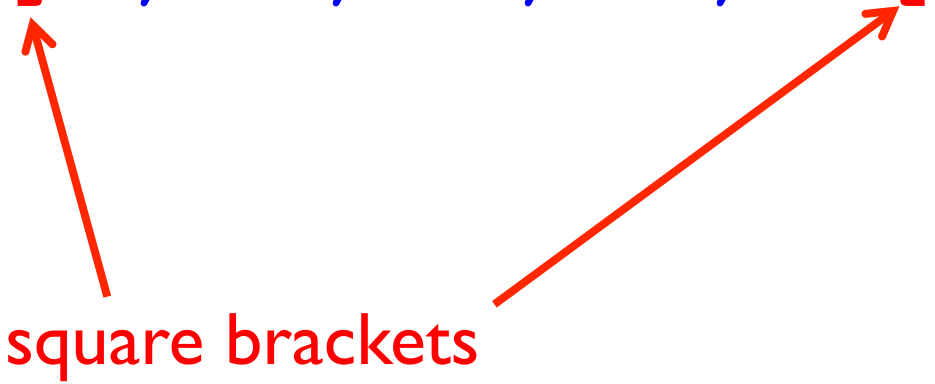
- Create lists by putting Python values or expressions inside square brackets, separated by commas:



Creating lists

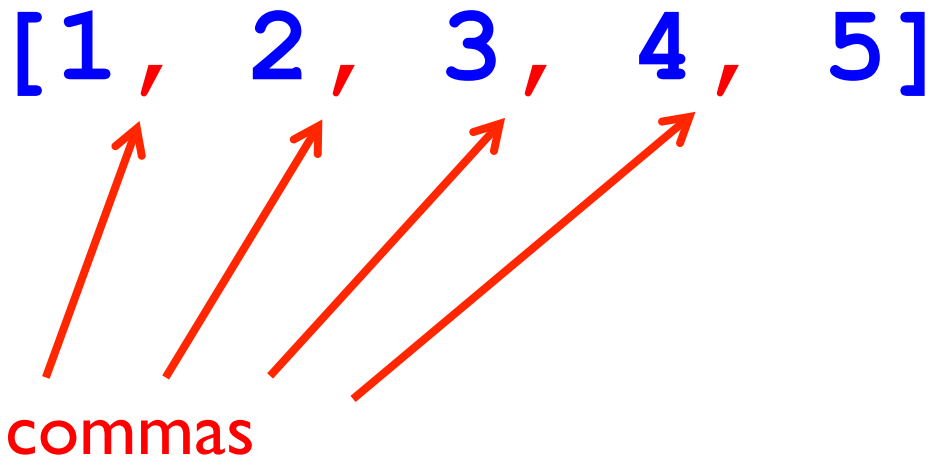
- Create lists by putting Python values or expressions inside square brackets, separated by commas:

```
[1, 2, 3, 4, 5]
```



Creating lists

- Create lists by putting Python values or expressions inside square brackets, separated by commas:



Creating lists

- Any Python expression can be inside a list:

```
[1 + 3, 2 * 2, 4]
```

- The expressions get evaluated when the list as a whole gets evaluated
  - so this list becomes `[4, 4, 4]`

Creating lists

- Lists can contain expressions with variables:

```
>>> a = 10
>>> [a, 2*a, 3*a]
[10, 20, 30]
```

Creating lists

- Lists can contain expressions with function calls:

```
>>> a = -4
>>> [a, 2*a, abs(a)]
[-4, -8, 4]
```

- (or any other Python expression)

Creating lists

- Lists can contain other lists:

```
>>> a = 4
>>> [[a, 2*a], [3*a, 4*a]]
[[4, 8], [12, 16]]
```

- This is called a *nested list*

Creating lists

- Lists can contain values of different types:

```
[1, 3.14, 'foobar', [0, 1]]
```

- But most of the time they have values of the same type:

```
[1, 2, 3, 4, 5]
[3.14, 2.718, 1.618]
['foo', 'bar', 'baz']
```

Accessing list elements

- Once a list is created, need to be able to get elements of list:

```
>>> temps = [59.6, 72.4, 68.5, 79.0,
             66.4, 77.1, -126.0]
>>> temps[0]
59.6
>>> temps[2]
68.5
```

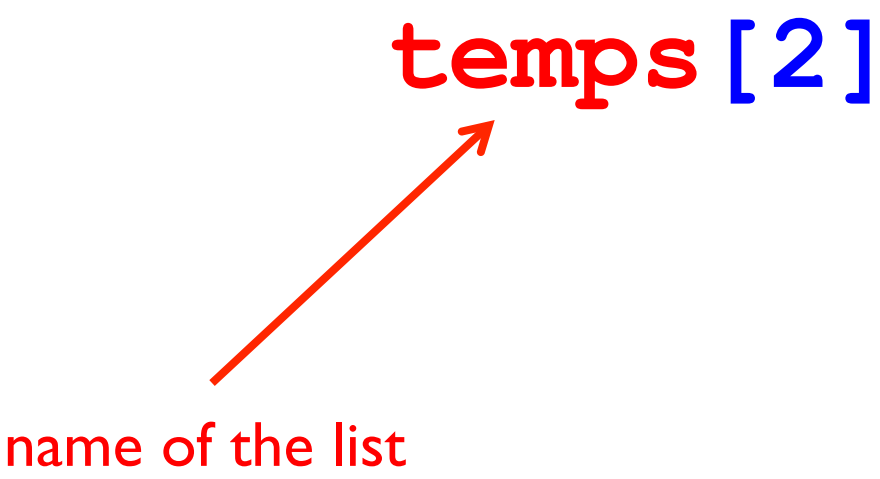
Accessing list elements

- Syntax:

```
temps[2]
```

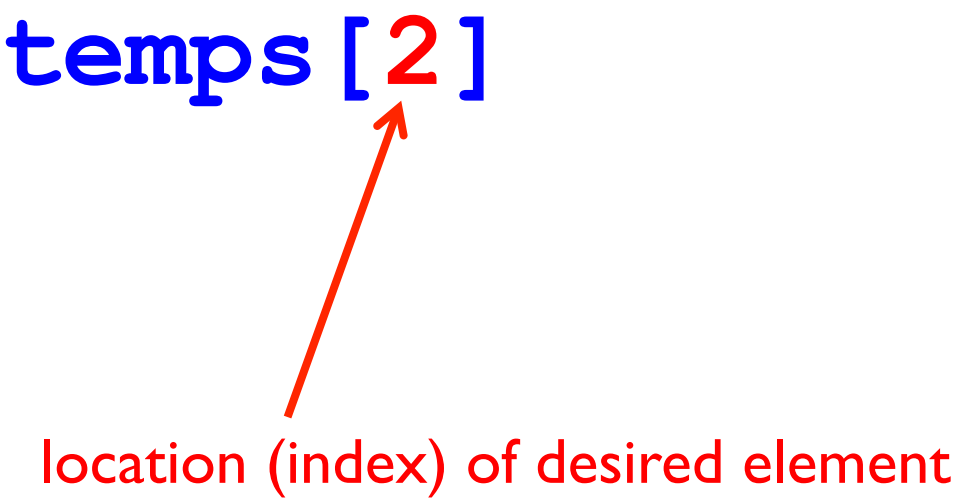
Accessing list elements

- Syntax:



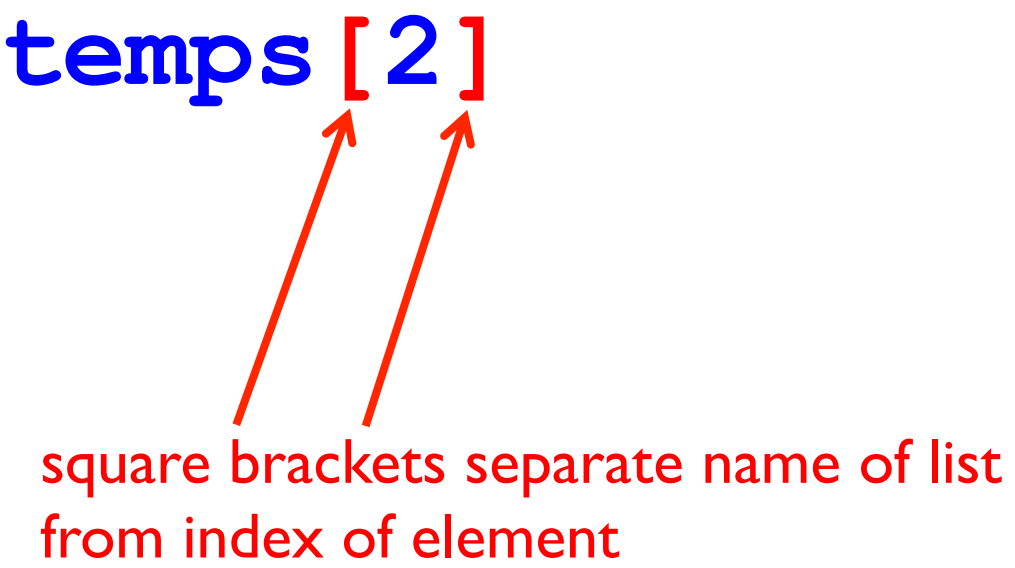
Accessing list elements

- Syntax:



Accessing list elements

- Syntax:



Accessing list elements

- NOTE: square brackets are being used for two distinct things:
  1. *creating* lists: `[1, 2, 3, 4, 5]`
  2. *accessing elements* of lists: `temps[2]`
- These are *completely different!*
  - Recall: Python likes to 'overload' syntax to mean different (but sometimes related) things

Accessing list elements

- Can even have both meanings in one expression:

```
>>> [1, 2, 3, 4, 5][2]
3
```

- (almost never see this in practice)

Accessing list elements

- List indices start at `0`, not `1`

```
>>> nums = [12, 42, 31, 51, -32]
>>> nums[0]  # first element of nums
12
>>> nums[1]  # second element of nums
42
```

- This is common in computer languages
  - but easy to make mistakes

Accessing list elements

- Can also access from the end of a list!

```
>>> nums = [12, 42, 31, 51, -32]
>>> nums[-1]  # last element
-32
>>> nums[-2]  # second-last element
51
```

- (but can't "wrap around")

Accessing list elements

- Accessing off the ends of the list is an error:

```
>>> nums = [12, 42, 31, 51, -32]
>>> nums[5]    # last element: nums[4]
IndexError: list index out of range
>>> nums[-6]   # first element: nums[-5]
IndexError: list index out of range
```

Empty list

- The empty list is written `[]`

```
>>> empty = []
>>> empty[0]
IndexError: list index out of range
```

- We'll see uses for this later

Modifying lists

- Recall: Python strings are *immutable*
  - means: can't change them after making them
- Lists are *mutable*
  - means: *can* change them after making them

Modifying lists

- Example:

```
>>> nums = [4, 6, 19, 2, -3]
>>> nums
[4, 6, 19, 2, -3]
>>> nums[2] = 501
>>> nums
[4, 6, 501, 2, -3]
```

Modifying lists

- Syntax:

```
nums[2] = 501
```

Modifying lists

- Syntax:

```
nums[2] = 501
```

element being modified

Modifying lists

- Syntax:

```
nums[2] = 501
```

new value at that location in list

Modifying lists

- Evaluation rule:
  - evaluate right-hand side expression
  - assign to location in list on left-hand side

```
>>> nums[2] = 3 * nums[2]
```

- `nums[2]` is `501`, so `3 * nums[2]` is `1503`

```
>>> nums
[4, 6, 1503, 2, -3]
```

Modifying lists

- Can change an element to an element with a different type:

```
>>> nums
[4, 6, 1503, 2, -3]
>>> nums[2] = 'foobar'
[4, 6, 'foobar', 2, -3]
>>> nums[0] = [42, 'hello']
>>> nums
[[42, 'hello'], 6, 'foobar', 2, -3]
```

List operators

- Operators on lists behave much like operators on strings
- The + operator on lists means list concatenation
  - (like + with strings means string concatenation)

```
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
>>> [1, 2, 3] + []
[1, 2, 3]
```

List operators

- The * operator on lists means list replication
  - (like * with strings means string replication)

```
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 0 * [1, 2, 3]
[]
>>> [1, 2, 3] * -1
[]
```

List functions

- Some built-in functions work on lists
- The **`len`** function returns the length of a list:

```
>>> len([1, 2, 3, 4, 5])
5
```

- The **`list`** function converts other sequences to lists, if possible:

```
>>> list('foobar')
['f', 'o', 'o', 'b', 'a', 'r']
```

List methods

- *Lots* of useful methods on lists
- **append**

```
>>> lst = [1,2,3,4,5]
>>> lst.append(6)
>>> lst
[1,2,3,4,5,6]
```

- **append** adds a new element to the end of a list

List methods

- Note that **`append`** changes the list it acts on
- Can use this to build up lists, starting from empty list

```
>>> lst = []
>>> lst.append(1)
>>> lst.append(2)
>>> lst.append(3)
>>> lst
[1, 2, 3]
```

List methods

- To find an element's index in a list, use the **`index`** method:

```
>>> lst = [1, 2, 3]
>>> lst.index(2)
1
>>> lst.index(42)
ValueError: list.index(x): x not
  in list
```

List methods

- If an element has multiple copies in a list, **`index`** returns the index of first copy:

```
>>> lst = [1, 2, 3, 2, 4]
>>> lst.index(2)
1    # index of 1st 2 in list
```

List methods

- To remove an element from a list, use the **`remove`** method:

```
>>> lst = [1, 2, 3]
>>> lst.remove(2)
>>> lst
[1, 3]
```

- Only removes first occurrence of element in list
- Error if element not found in list

List methods

- To reverse a list, use the **`reverse`** method:

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst.reverse()
>>> lst
[5, 4, 3, 2, 1]
```

- NOTE: the **`reverse`** method doesn't return the reversed list
  - it reverses the list 'in-place' and doesn't return anything

Interlude

- In "Python" ☺

Loops

- So far, have seen multiple kinds of data
  - `int`s, `float`s, strings, lists
- Have seen how to write functions with `def` and `return`
- Now we introduce another fundamental concept: a loop

Loops

- A loop is a chunk of code that executes repeatedly
  - though something must change each time the chunk is repeated (or else the program will never terminate)
- Python has two kinds of loop statements:
  - `for` loops (this lecture)
  - `while` loops (later in course)

Loops

- Loops are often associated with lists
- Basic idea:
  - for each element of this list
  - do the following ... [chunk of code] ...
- Example:
  - for each element of a list
  - print the element

Loops

```
title_words = ['Monty', 'Python',
               'and', 'the',
               'Holy', 'Grail']
for word in title_words:
    print word
```

for loop

Loops

- Result:

```
Monty
Python
and
the
Holy
Grail
```

Loops

- Structure of a `for` loop:

```
for <name> in <list>:
    <chunk of code>
```

- Chunk of code is executed once for each element of the list
- Each time through, `<name>` is bound to the next element of `<list>` and `<chunk of code>` is executed

Loops

```
title_words = ['Monty', 'Python', ...]
for word in title_words:
    print word
```

- First time through:
  - `word` is `'Monty'`
  - `print` prints `Monty`
- Second time through:
  - `word` is `'Python'`
  - `print` prints `Python`
- etc. until no more elements in list

Loops

- Another way to look at this:

```
for word in title_words:
    print word
```

- This is equivalent to:

```
word = 'Monty'
print word
word = 'Python'
print word
word = 'and'
print word  # etc.
```

Loops

- Chunk of code in a `for` loop is called a *block*
- Blocks can consist of multiple lines:

```
for word in title_words:
    print word
    print '---'
```

- This puts a line of `'---'` between words:

```
Monty
---
Python
---    (etc.)
```

Loops

- Syntax:
  1. Must have a colon (`:`) at the end of the `for` line
  2. Every line in block must be indented the same amount (or else it's a syntax error)
  3. End of block indicated when indent goes back to value before `for` loop began

Loop syntax errors

- No colon at end of `for` line:

```
for word in title_word
                       ^
SyntaxError: invalid syntax
```

Loop syntax errors

- Irregular indentation:

```
for word in title_word:
    print word
      print '---'
      ^
IndentationError: unexpected indent
for word in title_word:
    print word
  print '---'
  ^
IndentationError: unindent does not match any
  outer indentation level
```

Application: summing

- Want to sum elements of a list of numbers

```
nums = [-32, 0, 45, -101, 334]
sum_nums = 0
for n in nums:
    sum_nums = sum_nums + n
print sum_nums
```

- Result: `246`

The += operator and friends

- When you see a line of the form

```
x = x + 10
```

you can write

```
x += 10
```

(meaning is the same)

- Similarly, can write

```
x *= 10
```

for

```
x = x * 10
```

The += operator and friends

- In general, many operators `op` have `op=` counterparts:
  - `+=  -=  *=  /=  %=`
- You should use them where applicable
  - makes code more concise, readable

Application: summing

- Another way to do this:

```
nums = [-32, 0, 45, -101, 334]
sum_nums = 0
for n in nums:
    sum_nums += n
print sum_nums
```

- Result: `246`

Application: summing

- Yet another way to do this:

```
nums = [-32, 0, 45, -101, 334]
sum_nums = sum(nums)  # !!!
```

- Result: `246`
- `sum` is a built-in function on lists
- *Moral*: there is usually more than one way to accomplish any task!

Loops and strings

- Can use a `for` loop to loop through the characters of a string:

```
>>> for c in 'Python':
...     print c
P
y
t
h
o
n
```

Loops and strings

- Lists and strings are both *sequences*, so a `for` loop works similarly for both of them
  - much like the `len` function works for both lists and strings in a similar way

Loops and strings

- However, strings are immutable, so can't do this:

```
>>> s = 'Python'
>>> s[0] = 'J'
TypeError: 'str' object does not
  support item assignment
```

Nested loops

- Can nest one `for` loop inside another:

```
title = ['Monty', 'Python']
for word in title:
    for char in word:
        print char
```

```
M
o
n
t
...
```

Nested loops

- Can nest one `for` loop inside another:

```
title = ['Monty', 'Python']
for word in title:
    for char in word:
        print char
```

- First time through outer loop: `word` is `'Monty'`
- Inner loop: `char` is `'M'`, then `'o'`, then `'n'`, then `'t'`, then `'y'`
- Second time through outer loop: `word` is `'Python'`
- Inner loop: `char` is `'P'`, then `'y'`, etc.

LEAD
Leadership Education And Development

Final topic (whew!)

So far...

- Our programs have been "straight-line" programs
  - always do the same thing no matter what
  - We lacked the ability to make *decisions* based on the data
- Now we'll fix that
- Introduce the `if` statement

Problem

- Given a list of temperatures
  - say, temperature at noon every day this week
  - How many temps are above 72 degrees?
- Can't solve this with what we know so far

Two subproblems

1. How do we test to see whether or not a particular temperature is greater than 72 degrees?
2. How do we use that information to control our program?

Testing a number

- To test a number against some other number, we need a *relational operator*
- Examples: `<` `<=` `>` `>=` `==` `!=`
- Relational operators return a boolean value (`True` or `False`)

Relational operators

- `x == y` (is `x` equal to `y`?)
- `x != y` (is `x` not equal to `y`?)
- `x < y` (is `x` less than `y`?)
- `x <= y` (is `x` less than or equal to `y`?)
- `x > y` (is `x` greater than `y`?)
- `x >= y` (is `x` greater than or equal to `y`?)
- All of these operators return boolean (`True`/`False`) values

== vs. =

- Note: the == operator is completely different from the = (assignment) operator
  - really easy to make mistakes with this!

```
>>> a = 10    # assign a the value 10
>>> a == 10   # is a equal to 10?
```

- Completely different!

Testing the temperature

- Want to test if a temperature is greater than 72 degrees

```
>>> temp = 85
>>> temp > 72
True
```

The `if` statement

- Let's use this to do something:

```
>>> temp = 85
>>> if temp > 72:
...     print "Hot!"
Hot!
```

`if` statement

The `if` statement

- Structure of an `if` statement:

```
if <boolean expression>:
    <block of code>
```

The `if` statement

- Structure of an `if` statement:

```
if <boolean expression>:
    <block of code>
```

- Note that, like `for` statement, colon (`:`) has to come at the end of the `if` line

The `if` statement

- Structure of an `if` statement:

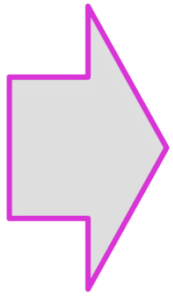
```
if <boolean expression>:
    <line of code>
    <line of code>
    ...
```

- Note that, like `for` statement, block of code can consist of multiple lines

The `if` statement

- Structure of an `if` statement:

```
if <boolean expression>:
    <line of code>
    <line of code>
    ...
```



- Note that, like `for` statement, block of code must be indented relative to `if` line

The `if` statement

- Interpretation of an **`if`** statement:

```
if <boolean expression>:
    <block of code>
```

- If the **`<boolean expression>`** evaluates to **`True`**, then execute the **`<block of code>`**
- Otherwise, don't!
- In either case, continue by executing the code after the **`if`** statement

Back to our problem

- We have a list of temperatures, one for each day this week

```
temps = [67, 75, 59, 73, 81, 80, 71]
```

- We want to compute how many of these temperatures are above 72
- How do we go about doing this?

Back to our problem

- For any given temperature, we know how to compare it with 72 and do something based on the result
  - need a relational operator (`>`) and an `if` statement
- But we have a whole list of temperatures
  - so will need a `for` loop as well
- This pattern (`if` inside a `for` loop) is a *very* common programming pattern!

Back to our problem

- Also need to keep track of the number of temperatures seen so far which are above 72
  - so need a variable to store the current count of temperatures above 72

Back to our problem

- Given all that, let's write our code
- To start with, haven't examined any temps
  - so count of temps > 72 is 0

```
temps_above_72 = 0
```

Back to our problem

- Now we have to examine each temp to see if it's above 72
  - use a `for` loop

```
temps_above_72 = 0
for t in temps:
    ???
```

Back to our problem

- For any given temp `t`, use an `if` statement and the `>` operator to test it:

```
temps_above_72 = 0
for t in temps:
    if t > 72:
        ???
```

Back to our problem

- If temperature t is > 72, add it to the count
  - otherwise do nothing

```
temps_above_72 = 0
for t in temps:
    if t > 72:
        temps_above_72 += 1
```

- And we're done!

Back to our problem

- All the code:

```
temps = [67, 75, 59, 73, 81, 80, 71]
temps_above_72 = 0
for t in temps:
    if t > 72:
        temps_above_72 += 1
print "%d days above 72" % temps_above_72
```

- Prints:

```
4 days above 72
```

Note

- This is a trivial example
  - Can easily do it in your head
- Would become less trivial if list contained 1,000,000 or 1,000,000,000 elements
- *Moral*: Computers aren't just about doing difficult computations
  - also about doing large numbers of *simple* calculations

More decisions

- An `if` statement allows you to either
  - do something (execute a block of code) when some condition is true
  - otherwise do nothing
- More generally, we might want to
  - do something when some condition is true
  - otherwise, do something `else` ...

`if` and `else`

- An **`if`** statement can *optionally* include a second part called the **`else`** clause
  - executed only if the **`<boolean expression>`** in the **`if`** statement was false

```
if <boolean expression>:
    <block of code>
else:
    <different block of code>
```

Example of `if` and `else`

```
temp = 69
if temp > 72:
    print "greater than 72!"
else:
    print "less than or equal to 72!"
```

- This will print:

```
less than or equal to 72!
```

But wait! There's more!

- Temperature can be compared more precisely than we did
- Might want to keep records of
  - days where temp is < 72
  - days where temp is == 72
  - days where temp is > 72

elif

- How would we say this in English?
- "*If* the temperature is less than 72, do <thing1>, *else if* the temperature is 72, do <thing 2>, *else* do <thing 3>."
- We can express this in Python using an **`elif`** statement inside an **`if`** statement
- **`elif`** is short for *"else if"*

elif

- This leads to:

```
if t < 72:
    temps_below_72 += 1
elif t == 72:
    temps_at_72 += 1
else:
    temps_above_72 += 1
```

elif

- We could even have multiple distinct **`elif`** blocks in our code if we wanted
- You can also have an **`elif`** without a closing **`else`** block
- Most of the time we can get by with just **`if`** and (maybe) **`else`**

Next time

- Learn how to organize our code into modules (AKA "libraries")
- Learn how to document our code
- Learn how to consult Python documentation