

# CS I

## Introduction to Computer Programming

*Lecture 6: October 15, 2012*

### Loops and decisions...



# Last time

- Docstrings
- Lists



# Today

- Loops
  - the **for** statement
- Making decisions
  - the **if** statement
  - the **else** and **elif** statements



# Loops

- So far, have seen multiple kinds of data
  - **ints**, **floats**, 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 (next lecture)



# 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.



# Continued...

- We'll talk more about loops next lecture
- We'll show you a different kind of loop
  - **while** loop



# Interlude

- Movie clip!



# 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>
```

```
    ...
```

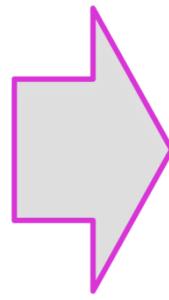
- Note that, like **for** statement, block of code can consist of multiple lines



# 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

```
temp = [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

```
temp_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:

```
temp_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

```
temp_above_72 = 0
for t in temps:
    if t > 72:
        temp_above_72 += 1
```

- And we're done!



# Back to our problem

- All the code:

```
temp = [67, 75, 59, 73, 81, 80, 71]
temp_above_72 = 0
for t in temp:
    if t > 72:
        temp_above_72 += 1
print "%d days above 72" % temp_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  $\approx 72$
  - days where temp is  $> 72$



# Multi-way tests (the bad way)

- We could write this (for a temperature `t`):

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

- This is bad code – why?



# Multi-way tests (the bad way)

- Let's say that `t` is **69**
- Let's follow the execution of the code...

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



# Multi-way tests (the bad way)

- First, execute the first `if` statement
- Causes `temps_below_72` to be incremented by 1

```
if t < 72:  
    temps_below_72 += 1
```

```
if t == 72:  
    temps_at_72 += 1
```

```
if t > 72:  
    temps_above_72 += 1
```



# Multi-way tests (the bad way)

- Then, execute the second **if** statement
- Nothing happens, because **t** isn't **72**

```
if t < 72:
```

```
    temps_below_72 += 1
```

```
if t == 72:
```

```
    temps_at_72 += 1
```

```
if t > 72:
```

```
    temps_above_72 += 1
```



# Multi-way tests (the bad way)

- Then, execute the third `if` statement
- Nothing happens, because `t` isn't  $> 72$

```
if t < 72:
```

```
    temps_below_72 += 1
```

```
if t == 72:
```

```
    temps_at_72 += 1
```

```
if t > 72:
```

```
    temps_above_72 += 1
```



# Multi-way tests (the bad way)

- The problem:
  - For many values of `t`, some of the three `if` statements may be executed unnecessarily
  - Can we figure out a better way to do this?
    - maybe using `else`?



# Second try

- Use `else`:

```
if t < 72:
```

```
    temps_below_72 += 1
```

```
else:
```

```
    if t == 72:
```

```
        temps_at_72 += 1
```

```
    else:
```

```
        temps_above_72 += 1
```



# Second try

- The good news: this works and is efficient

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



# Second try

- The bad news: nested `ifs` are ugly!

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



# Second try

- The bad news: nested `ifs` are ugly!

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

outer `if`



# Second try

- The bad news: nested `ifs` are ugly!

```
if t < 72:
```

```
    temps_below_72 += 1
```

```
else:
```

```
    if t == 72:
```

```
        temps_at_72 += 1
```

```
    else:
```

```
        temps_above_72 += 1
```

inner `if`



# Second try

- Is there a better way?

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



# Third try

- 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*"



# Third try's the charm

- This leads to:

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



# Third try's the charm

- This is both efficient and readable:

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



# Next time

- **while** loops
- the **break** statement
- Files and file input/output

