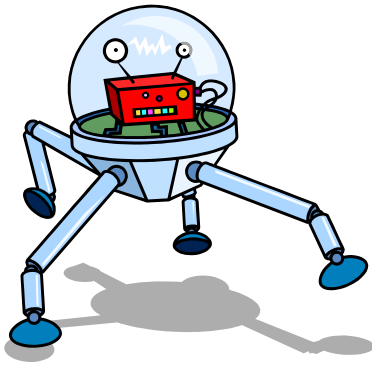
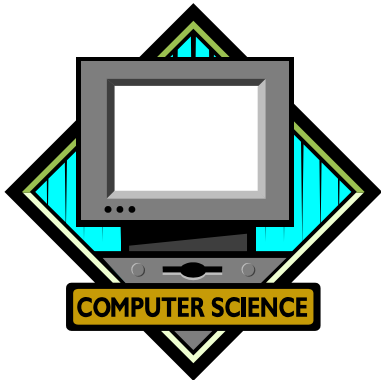
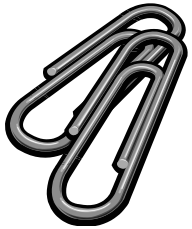
Caltech/LEAD Summer 2012 Computer Science

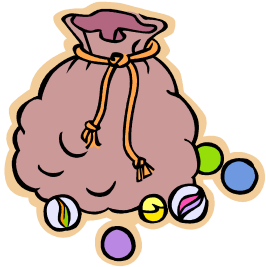




Lecture 6: July 12, 2012

Miscellaneous topics





LEAD
Leadership Education And Development

This lecture

- Function evaluation in depth
- Coding style
- Method chaining
- The `range()` function
- `while` loops and `break`
- Random numbers and the `random` module

Function evaluation

- We've seen how to define and use functions
- However, let's look at the process in more depth
- Example function: a function to compute the sum of squares of two numbers
- Here is a possible definition of this function:

```
def sum_of_squares(x, y):
    s = x * x + y * y
    return s
```

Function evaluation

- We can call this function as follows:

```
>>> sum_of_squares(3, 4)
25
```

- How does Python compute this result?

Function evaluation

```
>>> sum_of_squares(3, 4)
```

- The *arguments* of this function are...?
- `3` and `4`
- In the definition of the function:

```
def sum_of_squares(x, y):
    s = x * x + y * y
    return s
```

- The arguments are variables with the names `x` and `y`

Function evaluation

- The *body* of the function is the indented code that comes immediately after the **`def`** line:

```
def sum_of_squares(x, y):
    s = x * x + y * y
    return s
```

body of function

Function evaluation

- When we evaluate `sum_of_squares(3, 4)`, we
  - set `x` equal to `3` and `y` equal to `4`
  - evaluate the body of `sum_of_squares`, line by line
- This means that we have to evaluate

```
s = 3 * 3 + 4 * 4
return s
```

Function evaluation

- The first line is:

```
s = 3 * 3 + 4 * 4
```

- We evaluate an assignment statement like this by evaluating the right-hand side, then making the name on the left-hand side refer to that value
- The right hand side is:

```
3 * 3 + 4 * 4
```

- which is just `25`

Function evaluation

- So the line:

```
s = 3 * 3 + 4 * 4
```

- is equivalent to:

```
s = 25
```

- The next line:

```
return s
```

- returns the value of `s` (`25`) to the caller of this function

Function return values

- So the function call

```
>>> sum_of_squares(3, 4)
```

- gives the result:

```
25
```

- If we call a function from the Python shell, the return value (if any) is just printed
- Inside a file of code, functions with return values are usually used inside another statement (often an assignment statement)

Function return values

- Example function that uses `sum_of_squares`:

```
from math import sqrt

def root_sum_of_squares(x, y):
    ss = sum_of_squares(x, y)
    return sqrt(ss)
```

Function return values

- Notice that `sum_of_squares` is part of an assignment statement:

```
def root_sum_of_squares(x, y):
    ss = sum_of_squares(x, y)
    return sqrt(ss)
```

- The return value of `sum_of_squares` is given the name `ss` here
- The square root of `ss` is what's returned from this function

Function return values

- If we left out the assignment statement:

```
def root_sum_of_squares(x, y):
    sum_of_squares(x, y)  # useless!
    return sqrt(ss)
```

- then the second line wouldn't do anything useful
- The sum of squares would just be thrown away after being calculated
- `ss` doesn't mean anything here, so → error

return vs print

- If we used `print` instead of `return` in `sum_of_squares`:

```
def sum_of_squares(x, y):
    s = x * x + y * y
    print s
def root_sum_of_squares(x, y):
    ss = sum_of_squares(x, y)
    return sqrt(ss)
```

- What's the problem?

Coding style

- There are many correct ways to write code
- Some of these are more readable than others
- Rules of thumb for making code readable are called "coding style"
- Bad coding style will not prevent a program from running, but it will make the code harder for someone else to read and understand

Coding style

- Entire "style guides" exist on how to write code in a good style
- Right now, we'll just talk about some very basic kinds of style guidelines
- Make sure that your code doesn't violate any of these, and your graders will be very happy ☺

Style guideline 1: no tabs

- Don't use the "tab" character in your code!
- A "tab" character doesn't have a fixed width
  - depends on the setting of your terminal
- Code that looks great with one tab setting (*e.g.* one tab = 8 spaces) will probably look terrible with a different tab setting (*e.g.* one tab = 4 spaces)
  - code won't line up correctly

Style guideline 2: spaces

- You should make sure you have spaces in certain places:
  - around operators
  - after commas
  - after open-comment symbol
- Let's see examples of these

Spaces around operators

- This is bad:

```
a=b*c+d
```

- This is good:

```
a = b * c + d
```

- The second version is much more readable!

Spaces after commas

- This is bad:

```
def foo(a,b,c):
    return [a,b,c]
```

- This is good:

```
def foo(a, b, c):
    return [a, b, c]
```

- The second version is much more readable!

Spaces after open comment symbol

- This is bad:

```
#This is a comment.
```

- This is good:

```
# This is a comment.
```

- The second version is much more readable!

Style guideline 3: blank lines

- You should make sure you have at least one blank line between functions
- Bad:

```
def function1(x):
    return 2 * x
def function2(x):
    return 3 * x
```

Style guideline 3: blank lines

- You should make sure you have at least one blank line between functions
- Good:

```
def function1(x):
    return 2 * x

def function2(x):
    return 3 * x
```

Style guidelines

- There are many other style guidelines that experienced Python programmers use, but this will do for now
- The main point: make sure that your code can easily be read by other people!

Method chaining

- In Python, "objects" have methods associated with them
- Example: strings and lists are objects:

```
>>> s = 'I am a string'
>>> s.count('a')  # method call
2
>>> lst = ['I', 'am', 'a', 'list']
>>> lst.index('am')  # method call
1
```

Method chaining

- Most methods return values:

```
>>> s = '    This is another string.    '
>>> s.strip()
'This is another string.'
```

- If the returned value is an object too (*e.g.* another string), you can call a method on it too:

```
>>> s2 = s.strip()
>>> s2.upper()
'THIS IS ANOTHER STRING.'
```

Method chaining

- You can rewrite this in a shorter way:

```
>>> s = '    This is another string.    '
>>> s.strip()
'This is another string.'
>>> s.strip().upper()
'THIS IS ANOTHER STRING.'
```

- Same as:

```
>>> (s.strip()).upper()
```

- This is called 'method chaining'

range

- The `range` function creates lists of consecutive integers:

```
>>> range(0, 5)
[0, 1, 2, 3, 4]
>>> range(10, 20)
[10, 11, 12, 13, 14, 15, ..., 19]
>>> range(-10, -5)
[-10, -9, -8, -7, -6]
```

`range`

- The **`range`** function takes two arguments:
  - the first number in the list
  - the number *one after* the last number in the list
- So **`range(1, 10)`** gives you

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

and not

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- This seems weird, but will turn out to be useful

`range` with 3 arguments

- The `range` function can take an optional third argument:

```
>>> range(0, 10, 2)
[0, 2, 4, 6, 8]
```

- The third argument is the *step size*
  - the difference between consecutive list items
- Can even have negative step sizes:

```
>>> range(10, 0, -2)
[10, 8, 6, 4, 2]
```

`range` with 3 arguments

- Another way to think about it: `range` "always" has three arguments, but the last one is `1` by default when you use the two-argument form
- So:

```
range(0, 10)
```

really means:

```
range(0, 10, 1)
```

- Python allows you to define functions which can take different numbers of arguments (will see later in course)

`range` with 1 argument

- `range` can also be used with one argument
- In this case:
  - starting value is `0`
  - step size is `1`

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- Equivalent to:

```
>>> range(0, 10, 1)
```

- This is the most commonly-used form of `range`

Use of `range`

- `range` is often used with `for` loops:

```
>>> for i in range(1000):
...     print i
0
1
2
3
...
```

Use of `range`

- `range` is often used with `for` loops and lists:

```
>>> mylist = ['Caltech', 'is', 'great']
>>> for i in range(len(mylist)):
...     mylist[i] = mylist[i] + '-YEAH!'
>>> mylist
['Caltech-YEAH!', 'is-YEAH!', 'great-YEAH!']
```

- This `range(len(...))` pattern is very common in Python code

range(len(<list>))

- `range(len(<list>))` gives you a list of the valid indices of a particular list `<list>`
- For example:

```
>>> lst = ['a', 'list', 'of', 'strings']
>>> len(lst)
4
>>> range(len(lst))
[0, 1, 2, 3]
```

- `0`, `1`, `2`, and `3` are the valid indices of the list `lst`

`for i in range(len(<list>))`

- `for i in range(len(<list>))` used when need to have the indices of the list `<list>` inside a `for` loop (*e.g.* so you can change the list elements)
- If don't need to change list elements, usually can get by with just

```
for <element> in <list>: ...
```

Example

- Need to double every element of a list of numbers

```
nums = [23, 12, 45, 68, -101]
for i in range(len(nums)):
    nums[i] = nums[i] * 2
```

- Or could write second line as just

```
    nums[i] *= 2
```

More on loops

- Last time, we saw the `for` loop
- `for` is natural when working with lists
  - does something with each element of the list
- Sometimes, we're not working with lists
- Sometimes, we don't have a fixed number of things to loop over
- Sometimes, we don't know in advance how many times we will have to go through the loop

The `while` loop

- Python has a more primitive (simple) loop statement called a **`while`** loop
- Structure:

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

The `while` loop

- Note similarities with `if` and `for` forms

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

The `while` loop

- Note similarities with `if` and `for` forms

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

- Statement starts with the keyword `while`

The `while` loop

- Note similarities with `if` and `for` forms

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

- There is a colon (`:`) at the end of the first line
- It *must* be there, or else a syntax error!

The `while` loop

- Note similarities with `if` and `for` forms

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

- There is an indented block of code
  - which can be one or multiple lines

The `while` loop

- Evaluation of **`while`** loop:

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

1. Evaluate the **`<boolean expression>`**
2. If it evaluates to **`True`**, evaluate the **`<block of code>`** and repeat from the beginning
3. Otherwise, continue with the next line after the **`while`** loop

Example

- Starting at the number 10, print all the numbers from 10 down to 1

```
>>> num = 10
>>> while num > 0:
...         print num
...         num -= 1
10
9
```

... until reach `1`

Example

- When `num` is no longer > 0, the loop ends and execution continues on the line following the `while` loop

```
num = 10
while num > 0:
    print num
    num -= 1
print 'done with the while loop!'
```

A bad example?

- This example is unrealistic
- Could easily write this with a `for` loop:

```
for num in [10,9,8,7,6,5,4,3,2,1]:
    print num
```

- We know how many times through the loop in advance (10)

A bad example?

- Can rewrite using the `range` function to make it shorter:

```
for num in range(10, 0, -1):
    print num
```

- `range(10, 0, -1)` is equal to `[10,9,8,7,6,5,4,3,2,1]`

A better example

- Use `raw_input` to read numbers from the user and print them, stopping when a negative number is read
- In this case, we cannot know how many times we will have to go through the loop
  - because we can't control what the user does!
- This is a much more natural situation in which to use a `while` loop

A better example

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

Sample run

```
Enter a number: 10
Your number was: 10
Enter a number: 3
Your number was: 3
Enter a number: 1729
Your number was: 1729
Enter a number: 2716057
Your number was: 2716057
Enter a number: -91
Done!
```

Ugly code

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Why is this code ugly?

Ugly code

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Why is this code ugly?
- The exact same line is repeated twice!
  - a 'programming sin'

D.R.Y.

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- *Programming principle*: **D.R.Y.**
- Stands for **D**on't **R**epeat **Y**ourself

D. R. Y.

```
num = int(raw_input('Enter a number: '))
while num > 0:
    print 'Your number was: %d' % num
    num = int(raw_input('Enter a number: '))
print 'Done!'
```

- Repeated code usually means there is a better way to write the code
- Here, allows us to introduce some new tricks

Infinite loops

```
while True:
    <block of code>
```

- This is an *infinite loop*
- There is no way for the program to complete the `while` loop
  - (at least, no way that we know yet)
  - so it just goes on running
  - can halt it by typing `<Control>-C`

Infinite loops

- Infinite loops are not useless!
- But need some way to tell the loop when to stop executing
- Let's build up to that

New version of the example

```
while True:
    num = int(raw_input('Enter a number: '))
    print num
```

- This code is like previous code, except won't halt if `num` < 0
- Good thing: didn't have to write the `raw_input` line twice
- Bad thing: this never halts!
- Need a way to tell it when to stop

The `break` statement

```
while True:
    num = int(raw_input('Enter a number: '))
    if num < 0:
        break
    else:
        print num
```

- A **`break`** statement says "get out of this loop NOW!"
- It's a way to force a loop to end when some condition is met

The `break` statement

- A **`while`** loop works well when the condition to be tested can be tested before the body of the loop begins:

```
while <loop is not yet done>:
    <body of the loop>
```

The `break` statement

- A **`while`** loop needs a **`break`** statement if the condition to be tested occurs in the middle of the body of the loop:

```
while True:
    <body of the loop, part 1>
    if <loop is done>:
        break
    <body of the loop, part 2>
```

The `break` statement

- **`break`** statements are not needed often
- **`break`** statements are never "necessary"
  - can always re-write without using **`break`**
- But when a test naturally falls in the middle of a loop body, **`break`** can make code much cleaner (less repetition, not violating **`D.R.Y.`**)
- So, when appropriate:
  - give yourself a **`break`**!

Random numbers

- There are many times in programs where we want to do something involving random choices:
  - randomly choose one of a range of numbers
  - randomly shuffle the elements of a list
  - or just pick a random number inside some range
- In Python, we use the `random` module to do these kinds of tasks

Random numbers

- To use the **`random`** module, start by doing this:

```
import random
```

- Normally, we write this at the top of a file of code
  - even if we won't need it until much later in the code
- Now we can use all the functions in the **`random`** module

Random numbers

- There are many useful functions in this module:
  - `random.choice`
  - `random.shuffle`
  - `random.randint`
  - `random.random`
- Let's see what they do

random.choice

- `random.choice` takes a list as its argument, and returns a randomly-chosen value from the list

```
>>> random.choice(['foo', 'bar', 'baz'])
'bar'
>>> random.choice(['foo', 'bar', 'baz'])
'baz'
>>> random.choice(['foo', 'bar', 'baz'])
'bar'
```

- The list itself (`['foo', 'bar', 'baz']`) is not altered

random.shuffle

- `random.shuffle` takes a list as its argument, and randomly changes the order of all the list elements

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

- In this case, the list itself *is* altered
- There is no return value!
- We say that the list is shuffled 'in-place'

random.randint

- `random.randint` takes two integers as its arguments, and randomly chooses an integer in between the two (including the two)

```
>>> random.randint(1, 3)
1
>>> random.randint(1, 3)
3
>>> random.randint(1, 3)
2
>>> random.randint(1, 3)
3
```

random.random

- `random.random` takes no arguments and returns a random floating-point number in the range [0, 1) (>= 0, < 1)

```
>>> random.random()
0.17355380130879683
>>> random.random()
0.8038488498372278
>>> random.random()
0.9072302784530977
```

Next lectures

- Debugging
- Recursion