New topic

Modules and Functions

We have already seen that Python provides built-in function that we can use:

```
>>>
>>> len('abcdef')
6
>>> float(5)
5.0
>>> int(5.8)
5
>>> int('125')
125
>>>
```

These functions are available directly from the interactive shell. But …. say we want to get the square root of a number. It would seem reasonable that Python would provide a function to do that as well.

But when I try to use what I think should work, I get this:

```
>>>
>>> sqrt(9)
Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    sqrt(9)
NameError: name 'sqrt' is not defined
>>>
```

But the following will work.

```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

Why?

"**math**" is a **module (= a file)** containing a number of mathematical functions.

The "**import**" statement instructs Python to "load" the module and make these functions available for use.

Which functions are available in the math module?

We do it with the "dir" command:

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan',
 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'er
fc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gam
ma', 'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', '
log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 't
runc']
>>>
```

If we want to know what each one of these function does we use the "help" command:

```
>>> help(math)
Help on built-in module math:

NAME
    math

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

        Return the hyperbolic arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.

    asinh(...)
        asinh(x)

        Return the hyperbolic arc sine (measured in radians) of x.

    atan(...)
        atan(x)

        Return the arc tangent (measured in radians) of x.

    atan2(...)
```

And it goes on and on ..

You can also get help on a **single function**:

```
>>> help(math.tan)
Help on built-in function tan in module math:

tan(...)
    tan(x)

    Return the tangent of x (measured in radians).
```

How do we use functions in a module?

There are <u>three</u> ways.

1. The way we just saw: This just makes the module available but we need to use the "dot" syntax to actually access the function.

```
>>> import math
>>> math.sqrt(9)
3.0
>>>
```

2. We can **<u>import a single function</u>** from a module. The function is then available to be used "directly" like the len() function.

```
>>> from math import sqrt
>>> sqrt(9)
3.0
>>>
```

3. We can **<u>import all the functions from a module at one time</u>**. We can then use the function name directly as in 2 above.

```
>>> from math import *
>>> sqrt(9)
3.0
>>>
```

Question:

What are the advantages and disadvantages of each of the methods above?

Answer:

We can also do this:

```
>>>
>>> import math
>>> sqrt=math.sqrt
>>> sqrt(9)
3.0
>>>
```

We can create <u>our own functions and modules</u>!

But why would we want to? There are a number of reasons.

Answer:

How do we create functions? Easy. We use "**def**:

```
>>>
>>> def gt(x,y):
        if x>y:
            return True
        else:
            return False

>>> gt(3,4)
False
>>> gt(4,3)
True
>>>
```

<u>IMPORTANT:</u> **<u>Terminology</u>**: The x any above are called **<u>parameters</u>**, the 3 and 4 are called **<u>arguments</u>**. The arguments can be constants as in the above example, or variables as in the examples below.

Here is the **syntax**.

def function_name(parameter list): # the parameter list could be empty – but still need ().

code block

And the **semantics**:

1. A function needs to be defined, using the "def" construction above, before it is used. Otherwise Python will issue an error like this:

```
>>>
>>> is_prime(5)
Traceback (most recent call last):
  File "<pyshell#50>", line 1, in <module>
    is_prime(5)
NameError: name 'is_prime' is not defined
>>>
```

2. The parameter list is a list of variables that will refer to local copies of the arguments "**passed**" from the calling code. For example:

```
>>>
>>> def add(x,y):
        x+=1
        y+=1
        print(x,y)

>>> a=10
>>> b=20
>>> add(a,b)
11 21
>>> a
10
>>> b
20
>>>
```

Notice1: The variables in the parameter list are **local**. This means that any changes that you make to them in the function do not affect the values in the corresponding arguments in the main program. What happens in Vegas ….

Notice2: The actual story is a bit more subtle than **Notice1**. We have passed in simple types. Stay tuned for what happens later on.

Question: How does a function return a value back to the "caller"?

Answer: It uses the "return statement". It has two forms:

- **return <some value>**
- **return**
- **or … the function just "falls off" the last statement and implicitly returns to the caller.**

The first form terminates the function and makes <value> available at the place from which the function was called. Program execution resumes from that point as well.

The second form just terminates the function, and computation resumes from the point from which the program was called.

The third "form" behaves just like the one above.

Terminology: When a function doesn't return a value, but rather performs some function for us, we will sometimes call it a **procedure**.

```
>>> type(len('asd'))
<class 'int'>
>>>
```

This tells us that the len() function returns an int.

```
>>> type(print('asd'))
asd
<class 'NoneType'>
>>>
```

But the print function returns "NoneType", a catchall that says that the function returns nothing. The print function is not computing a value for us, it's basically "doing a job" for us, and then returning to the caller. We will call this a **procedure**.

Problem:

Write a function is_even(x) which returns True if x is even and False otherwise.

Answer:

Problem:

Write a function is_leap(x) which returns True if x is a leap year and False otherwise.

Answer:

Problem:

Write a function is_prime(x) which returns True if x is a prime number and False otherwise.

Answer:

Problem:

Write a program to ask the user for two integers, first and last. Write a program to print out all the primes between first and last (inclusive), five values per line.

Answer:

L4_func_isPrime.py

Problem:

Write a function sum_of_digits1(n) which returns the sum of the digits of n.

Answer:

Problem:

Write a function sum_of_digits2(n) which prints the sum of the digits of n.

Answer:

Just like there are conversions, int, float, str, there is a built in function called **bin**. The documentation says:

`bin`(*x*)

Convert an integer number to a binary string.

For example:

```
>>>
>>> bin(6)
'0b110'
>>>
```

Problem:

Write a function my_bin(n) which converts an integer number to a binary string representation of n. But Leave out the leading '0b' returned by the built in function bin.

Answer:

L4_binary_conv.py

Global Variables

Question: How can we get functions to change the values in variables outside of themselves?

We have already seen that variables in a function are local, so changing them doesn't affect (in the instances we considered) the arguments passed over. But what about other variables, like in the following example?

```
a=15

def change():
    a=100
    print(a)

change()
print(a)
```

We get:

```
>>>
100
15
>>>
```

but if we add a **global** statement:

```
a=15

def change():
    global a
    a=100
    print(a)

change()
print(a)
```

we get:

```
>>>
100
100
>>>
```

Lists Strings, Tuples and Other Sequences

The following 2 pages are for reference.

Sequences represent ordered sets of objects indexed by nonnegative integers and include strings, (including Unicode strings) lists, and tuples. Strings are sequences of characters, and lists and tuples are sequences of arbitrary Python objects. Strings and tuples are immutable; lists allow insertion, deletion, and substitution of elements. All sequences support iteration.

Don't worry! All the strange terms above will be explained below.

Operations and Methods Applicable to All Sequences

Item	Description
s[i]	Returns element i of a sequence
s[i:j]	Returns a slice
s[i:j:stride]	Returns an extended slice
len(s)	Number of elements in s
min(s)	Minimum value in s
max(s)	Maximum value in s

Operations Applicable to Mutable Sequences

Item	Description
s[i] = v	Item assignment
s[i:j] = t	Slice assignment
s[i:j:stride] = t	Extended slice assignment
del s[i]	Item deletion
del s[i:j]	Slice deletion
del s[i:j:stride]	Extended slice deletion

Lists are sequences of arbitrary objects.

You create a list as follows:

names = ["Dave", "Mark", "Ann", "Phil"]

Lists are indexed by integers, starting with zero. Use the indexing operator to access and modify individual items of the list:

a = names[2] # Returns the third item of the list, "Ann"
names[0] = "Jeff" # Changes the first item to "Jeff"

To append new items to the end of a list, use the append() method:
names.append("Kate")

To insert an item in the list, use the insert() method:
names.insert(2, “Sydney”)

You can extract or reassign a portion of a list by using the **slicing operator**:
b = names[0:2] # Returns [“Jeff”, “Mark”]
c = names[2:] # Returns [“Sydney”, “Ann”, “Phil”, “Kate”]
names[1] = ‘Jeff’ # Replace the 2nd item in names with ‘Jeff’
names[0:2] = [‘Dave’,’Mark’,’Jeff’] # Replace the first two items of
the list with the list on the right.

Use the plus (+) operator to **concatenate** lists:
a = [1,2,3] + [4,5] # Result is [1,2,3,4,5]

Lists can contain any kind of Python object, including other lists, as in the following example:

a = [1,”Dave”,3.14, [“Mark”, 7, 9, [100,101]], 10]

Nested lists are accessed as follows:
a[1] # Returns “Dave”
a[3][2] # Returns 9
a[3][3][1] # Returns 101

List Methods

Method	Description
list(s)	Converts s to a list.
s.append(x)	Appends a new element, x, to the end of s.
s.extend(t)	Appends a new list, t, to the end of s.
s.count(x)	Counts occurrences of x in s.
s.index(x [,start [,stop]])	Returns the smallest i where s[i] ==x. start and stop optionally specify the start- ing and ending index for the search.
s.insert(i,x)	Inserts x at index i.
s.pop([i])	Returns the element i and removes it from the list. If i is omitted, the last element is returned.
s.remove(x)	Searches for x and removes it from s.
s.reverse()	Reverses items of s in place.
s.sort([cmpfunc [, keyf [, reverse]]])	Sorts items of s in place. cmpfunc is a comparison function. keyf is a key function. reverse is a flag that sorts the list in reverse order.

We will start with

Lists

A list in Python is a mutable sequence of any time of Python object.

What does this mean??

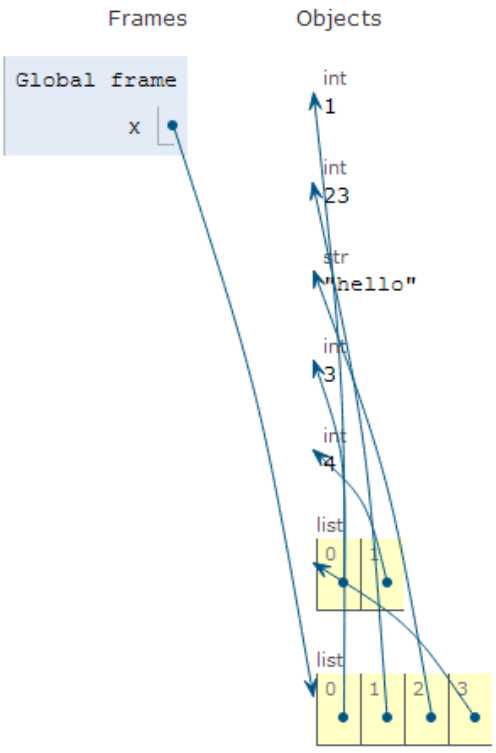
Example:

x=[1,23,”hello” , [3.4]]

x is the name of the list. It has 4 elements:

- the integer 1
- the integer 23
- the string “hello”
- the list [2,3]

It looks something like this in memory.



We create a new empty list in one of two ways:

- x=[]
- x=list()

or, as above, we can create a list with elements just by listing the elements in the square brackets "[" , "]".

Question:

How do we **access** individual elements of a list?

Answer:

We use **<u>square brackets with an integer</u>** to "index" into the list. For example:

```
>>> x=[1,23,"hello", [3,4]]
>>> print(x[0])
1
>>> print(x[3])
[3, 4]
>>>
>>> print(x[3][0])
3
>>> print(x[4])
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    print(x[4])
IndexError: list index out of range
>>>
```

Notice:

1. The positions in the list are numbered from 0 (not 1). In the above example, this means that the last element in the lost is accessed as x[3].

2. Since x[3] in our example is the list [3,4] we can access its elements by using a second index. That is why x[3][0] is the "zeroth" (i.e. first) element of [3,4], which is 3.

3. Since there is no element in the list x[4] (they are x[0], x[1], x[2], x[3]) we are trying to access a nonexistent element and Python prints an error message.

Question:

Can we change (i.e. replace) elements of a list?

Answer:

Yes. Here is an example where we modify the list x above.

```
>>>
>>> x[0]="Bob"
>>> x
['Bob', 23, 'hello', [3, 4]]
>>>
```

We can find out the size (=length) of a loop by using the len() function:

```
>>>
>>> x
['Bob', 23, 'hello', [3, 4]]
>>> len(x)
4
```

We say that a list is **mutable**. This means that is can be modified (i.e. "mutated).

Question:

How can we add elements to an existing list?

Answer:

There are a number of different ways. We start with two functions:

- append – add "something" to the end of a list
- extend – add all the elements of some **sequence** at the end of a list

```
>>>
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a.append(4)
>>> a
[1, 2, 3, 4]
>>> a.append("hello")
>>> a
[1, 2, 3, 4, 'hello']
>>> b=[5,6,7]
>>> a.append(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7]]
>>> a.extend(b)
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7]
>>> a.extend(8)
Traceback (most recent call last):
  File "<pyshell#35>", line 1, in <module>
    a.extend(8)
TypeError: 'int' object is not iterable
>>> a.extend([8])
>>> a
[1, 2, 3, 4, 'hello', [5, 6, 7], 5, 6, 7, 8]
>>>
```

Make sure the example above is absolutely clear!

Lists and loops

List and loops are made for each other!

```
>>>
>>> s=[]
>>> for i in range(1,11):
        s.append(i)

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

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list and print the sum.

L4_sum_list.py

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop, add up all the **elements** of the list **that are even** and print the sum.

L4_sum_even_list.py

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list that are odd and print the sum.

Problem:

Write a program that creates a list with the integers 1 – 10. Using a for loop add up all the elements of the list **that are in even positions** (0 is even) and print the sum.

When using lists it is often convenient to have Python generate some random values for us. Python provides a module called random that has some useful functions for this purpose. The two that we will use most are:

- random and
- randint

```
>>>
>>> from random import random,randint
>>> random()
0.5697709209009185
>>> help(random)
Help on built-in function random:

random(...)
    random() -> x in the interval [0, 1).

>>> randint(3,45)
38
>>> help(randint)
Help on method randint in module random:

randint(self, a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.

>>>
```

So ..

Function “random()” generates a random **floating point number** from zero up to but not including one.

Function randint(a,b) generates a random **integer** in the range a to b inclusive. Note that a and b here are integers.

Problem:

Write a program to fill a list of size 10 with random integers in the range 1 – 10 and print it out.

Problem:

Modify the program above so that we print the list as well as the maximum integer in the list. Do this two ways.

L4_random_fill3.py

Problem:

Modify the program above so that it also prints the position in the list where the maximum element was found.

Problem:

Using the code from the program above write a **function**

getmax(x,i) #x is a list and i is an integer

which will find and **return the maximum element among the first i elements of list x**.

getmax will return **two values**:

- the maximum element found, and
- the position in list x where that element was found.

For example, say a=[4,2,7,1,45,23], then getmax(a,4) will search for the maximum element in the first 4 element of list a.

So, in this case it will look at the following numbers: 2,4,7,1, and getmax(a,4) will return 7,2. This because in the first 4 elements, the largest is 7 and it is in position 2.

If we ran getmax(a,6) the function will return 45,4.

Problem:

Generate all the primes between 2 and 100.

Solution:

The **Sieve of Eratosthenes** provides an efficient solution. This algorithm is over **2200 years o**ld!

Here is the Wikipedia description: http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes

See there for an animation of the algorithm.

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the first prime number.
3. Starting from p, enumerate its multiples by counting to n in increments of p, and mark them in the list (these will be 2p, 3p, 4p, etc.; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.

When the algorithm terminates, all the numbers in the list that are not marked are prime.

Why does it work?

The main idea here is that as we go through the algorithm, every value for p is prime, because we will have already marked all the multiples of the numbers less than p. Note that some of the numbers being marked may have already been marked earlier (e.g. 15 will be marked both for 3 and 5).

The program below is a slight modification of the above algorithm.

This program will generate all primes in the range 2-100 using the sieve method.

We use two lists: x and primes.

list x will initially have the values 0-100 so that x[i]=i
list primes, initially empty, will grow as each new prime is found and appended onto it.

Each time a new prime p in x is located:

- p will be appended to list primes
- its place in list x will be set to zero,
- set all of its multiples in list x will be to zero.

When all elements in x are zero (we check this with the sum function), the main while loop terminates and the program prints list primes.

```
x=[]
for i in range(101):
    x.append(i)

primes=[2]

x[0]=x[1]=x[2]=0

p=2  # the first prime

while sum(x) != 0: # as long as sum(x)!=0, there are still non-zero entries in x.

    # Zero out all multiples of p
    i=1
    while i*p<=100:
        x[p*i]=0
        i=i+1


    # Now, look for the next prime

    p=p+1
    while x[p]==0: # it's at the next non-zero position of x
        p=p+1

    # We found it. Add it to the list of primes, and zero out its position in x

    primes.append(p)
    x[p]=0

# Done! Now print the list of primes.
print(primes)
```

Let's take a break!

Here is a **really** interesting

Microsoft/Google/Wall Street Interview Question!

1. Run the following program:

```
from math import sqrt
from random import random

count=0

for i in range(1000000):
    x=random()
    y=random()
    if sqrt(x*x+y*y)<1:
        count+=1

print(4*(count/1000000))
```

2. What is it calculating?

3. How/why does it work? What is the theory behind this?

Slicing lists

What is a slice of a list?

If x is a list then **the slice** **x[a:b]** is the "sub-list" of the elements of the elements of a **from** index position a **up to but not including** index position b.

```
>>>
>>> a=[1,2,3,4,5,6,7]
>>> b=a[3:5]
>>> b
[4, 5]
>>>
>>>
```

If we want to indicate that the slice starts at the beginning of the list, we can leave out the start value:

```
>>>
>>> c=a[:5]
>>> c
[1, 2, 3, 4, 5]
>>>
>>>
```

If we want to indicate that the slice goes all the way to the end of the list, we can leave out the end value:

```
>>>
>>> d=a[4:]
>>> d
[5, 6, 7]
>>>
>>>
```

Leaving out both the start and end indexes is the same as saying the whole list. So:

```
>>>
>>> e=a[:]
>>> e
[1, 2, 3, 4, 5, 6, 7]
>>> a
[1, 2, 3, 4, 5, 6, 7]
>>>
>>>
```

What can we do with a slice of a list?

1. As we saw above, we can create a new list form a slice.

2. We can assign to a slice and thereby replace one sub-list by another.

```
>>>
>>> a[2:5]=['a','b','c']
>>> a
[1, 2, 'a', 'b', 'c', 6, 7]
>>>
>>>
```

Notice that this is a **generalization** of accessing and replacing one list element as in **a[1]=12** which just replaced a single list element.

When we use a slice we can indicate a stride.

Huh?

The stride is the length of the "step" that you take going from one element to the next when creating the slice.

In the following example 2 is the stride.

```
>>>
>>> x=[10,20,30,40,50,60,70,80,90]
>>> y=x[1:8:2]
>>> y
[20, 40, 60, 80]
>>>
```

We can assign a list to a slice with a stride, **but** the list on the right hand side of the assignment must be the same size as the list produced by the slice. In the following example, both are of size 4.

```
>>>
>>> x[1:8:2]=['a','b','c','d']
>>> x
[10, 'a', 30, 'b', 50, 'c', 70, 'd', 90]
>>>
>>>
```

Note: The right hand side of a slice assignment can be any iterable (a string for example) as long as the lengths are the same.

```
>>> a=[1,2,3,4,5,6,7,8,9,10]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> a[1:10:2]='abcde'
>>> a
[1, 'a', 3, 'b', 5, 'c', 7, 'd', 9, 'e']
```

Sorting

Sorting is an operation on a list that orders the list elements in a specific order.

For example:

1. **A list of names** may be sorted in "lexical" =dictionary=alphabetical order.

Here is a formal definition of lexical order from Wikipedia:

The name of the lexicographic order comes from its generalizing the order given to words in a dictionary: a sequence of letters (that is, a word)

a1a2 ... ak

appears in a dictionary before a sequence

b1b2 ... bk

if and only if at the first i where ai and bi differ, ai comes before bi in the alphabet.

That comparison assumes both sequences are the same length. To ensure they are the same length, the shorter sequence is usually padded at the end with enough "blanks" (a special symbol that is treated as coming before any other symbol).

Note that we can order the names in reverse order, from latest to earliest. In this case the words still are in lexicographic order, but from the last element to the first.

2. **A list of integers** may be listed in either from smallest to largest or vice versa.

Why sort?

It turns out that sorting is one of the most important operations that programs perform. Two examples.

1. **Searching a list**. In order to find a specific element in a list we often sort it first. A sorted list can be searched much more quickly than one that is unsorted. Imagine looking up a phone number in a phone book (list) with a million entries. If the list is unsorted, we might need to look at 1,000,000 entries. If it is sorted, we don't need more than 20.

2. **A Scrabble dictionary**. We might want to bring all the words with the same letters next to each other in the list. So if we got the letters '**opts'** we would like to have stop, pots, and tops all next to one another. Imagine that we had a function called "signature()" that transforms each of stop, tops and post ➔ opts. Then if D is the list of dictionary words then

D.sort(key=signature)

would do this for us. We will actually do this later on.

Problem:

Given a list of integers, sort it so that its elements will be in ascending order.

Selection Sort

Here is the beginning of the Wikipedia entry. http://en.wikipedia.org/wiki/Selection_sort

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Here is an example of this sort algorithm sorting five elements:

```
64 25 12 22 11
11 25 12 22 64
11 12 25 22 64
11 12 22 25 64
11 12 22 25 64
```

L6_selection_sort.py

And here is a simple (but not very efficient) implementation.

It uses two new list functions.

```
a=[4, 2, 7, 1, 45, 23]

def select_sort(x):
    for i in range(len(x)-1):
        y=x[i:] # each time through the loop look for the minimum from position i to the end.
        m=min(y)
        pos=x.index(m,i,len(x)) # find the index of the first element with value m in the range [i,len(x) )
        x[i],x[pos]=x[pos],x[i] # swap the element at position i with the element at position pos

select_sort(a)
print(a)
```

Notice that this function uses two list functions **min()** and **index().** In the following, s is a list.

min(s) which returns the smallest item of *s*

s.index(x[, i[, j]]) which return smallest *k* such that s[k] == x and i <= k < j

In the index function i and j are optional. If omitted index searches the whole list. If item x is not found in list s, Python returns an error. In general, we should first as Python "x in s" before using the index function. In function select_sort() we don't have to do this since we know that m exists.

Question: Can you detect two inefficiencies in the implantation above?

Answer:

The following is a more efficient implementation of the same algorithm.

```
def select_sort(x):
    for i in range(len(x)-1):
        m=x[i]
        pos=i
        for j in range(i,len(x)):
            if x[j]<m:
                m=x[j]
                pos=j
        x[i],x[pos]=x[pos],x[i]
```

Questions:

Why is it more efficient?

Why does the outer for loop have range(**len(x)-1**) but the inner loop has range(i,**len(x)**)?

Here is **another elementary sort** called

Bubble Sort

Here is the beginning of its Wikipedia entry.

Bubble sort, sometimes incorrectly referred to as sinking sort, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list.

The full article and an animation is here: http://en.wikipedia.org/wiki/Bubble_sort

Here is the code.

```
# Bubble Sort

def bubble_sort(a):
    for i in range(len(a)):
        sorted=True
        for j in range(len(a)-i-1):
            if a[j]>a[j+1]:
                a[j],a[j+1]=a[j+1],a[j]
                sorted=False
        if sorted==True:
            return

# Lets test it.
a=[33,6,3,21,88,30]
bubble_sort(a)
print(a)
.
```

Sorting … a third way, using Python's built-in sort function.

L is a list.

```
>>>
>>> a.sort(
          L.sort(key=None, reverse=False) -- stable sort *IN PLACE*
```

Notice: function sort() takes 2 optional **key word** arguments:

- key
- reverse

key specifies a function of one argument that is applied to the list elements before the comparison is made.

reverse specifies that the list should be in reverse order. That means that ">" is used for comparison rather than "<".

Why is it called a "keyword argument"?

Because if you want to use it, you need to use the "keyword=value" syntax. We will see this below.

Say, **for example**, **we want to sort a list of strings**. String comparison **depends on capitalization** as in the following example. If we wanted to discount the capitalization in the comparison we could use the lower() function.

```
>>> 'abc'<'ABC'
False
>>>
>>>
>>>
>>> 'abc'>'ABC'
True
>>>
>>>
>>> 'abc'.lower()<'ABC'.lower()
False
>>>
>>>
>>> 'ABC'.lower()
'abc'
>>>
```

The default value for key is None.

Now, let's sort a list of strings.

```
>>>
>>> a=['ONE','two','one','TWO']
>>> b=['ONE','two','one','TWO']
>>> a
['ONE', 'two', 'one', 'TWO']
>>> b
['ONE', 'two', 'one', 'TWO']
>>> a.sort()
>>> a
['ONE', 'TWO', 'one', 'two']
>>> b.sort(key=str.lower)
>>> b
['ONE', 'one', 'two', 'TWO']
>>>
```

What about the keyword argument “reverse”?

Here is an example.

```
>>>
>>> a=[34,4,21,77,5,45,8]
>>>
>>> a.sort()
>>>
>>> a
[4, 5, 8, 21, 34, 45, 77]
>>>
>>>
>>> a.sort(reverse=True)
>>>
>>> a
[77, 45, 34, 21, 8, 5, 4]
>>>
>>>
```

Problem:

Given a list, print the elements of that list in reverse order. Do this in two ways.

Problem:

Given a list reverse the elements of the list. For example if

x=[1,2,3], then after it is reversed x would be [3,2,1].

Do this in two ways.

L6_reverse_list.py

Two dimensional lists

Many important applications use data that is represented in a 2-dimensional table.

10	20	30
40	50	60
70	80	90

How do we represent this in Python?

We simply use a list of lists.

a=[[10,20,30],[40,50,60],[70,80,90]]

Notice that the length of list a is 3 (len(a)==3), but it's made up of three lists, each one of length 3.

```
>>>
>>> a=[ [10,20,30],[40,50,60],[70,80,90] ]
>>> len(a)
3
>>> len(a[0])
3
>>>
>>>
```

Problem:

Change element with a 50 to 500.

Solution:

The 50 is the second element of the second list. Recalling that lists are indexed starting with 0, we write:

```
>>>
>>>
>>> a[1][1]=500
>>> a
[[10, 20, 30], [40, 500, 60], [70, 80, 90]]
>>>
>>>
```

Nested loops and two dimensional lists

Even though a list is "really" is a list of lists, when we program its useful to think of it as a two dimensional table.

So, for the list a above, **we can consider it a table with three rows and three columns**. The rows and columns are each indexed starting at 0. **We will say that he position with entry 500 is at row 1 and column 1.**

We saw how lists and loops are "made for each other." The same is true with two dimensional lists (we will sometimes refer to the as two dimensional "arrays". This is what they are called in many other programming languages (though they are implemented differently).

Problem:

Print list a above so that each "row" of it prints a separate row.

```
>>>
>>> for i in range(3):
        for j in range(3):
                print(a[i][j],end=' ')
        print()

10 20 30
40 500 60
70 80 90
>>>
>>>
```

And formatted …

```
>>>
>>> for i in range(3):
        for j in range(3):
                print(format(a[i][j],">6d"),end=' ')
        print()

    10     20     30
    40    500     60
    70     80     90
>>>
>>>
```

Problem:

Create a 4X4 array and initialize each of the elements to 0.

Solution:

```
a=[]
for i in range(4):
   a.append(4*[0])
```

What does 4*[8] mean?

Python lets us use + and * with lists.

```
>>>
>>> a=[1,2,3]
>>> a
[1, 2, 3]
>>> a+4
Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    a+4
TypeError: can only concatenate list (not "int") to list
>>> a+[4]
[1, 2, 3, 4]
>>>
>>>
```

So from here you see + is **concatenate** i.e. it acts like the list function **extend**. But you can only + a list to a list, not a string to a list like you can with extend.

What about "*" ?

```
>>>
>>> a=4*[8]
>>> a
[8, 8, 8, 8]
>>>
>>>
```

Make sure that you can explain each of the following:

```
>>> a=3*3*[[0]]
>>> a
[[0], [0], [0], [0], [0], [0], [0], [0], [0]]
>>> a=3*3*[0]
>>> a
[0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> a=3*[3*[0]]
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

The first example:

The second example:

The third example:

The fourth example (below). Does this produce the same result as third example above?

```
>>> a=[]
>>> for i in range(3):
        a.append(3*[0])


>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

Notice when we print list a, both seem to produce the same list:

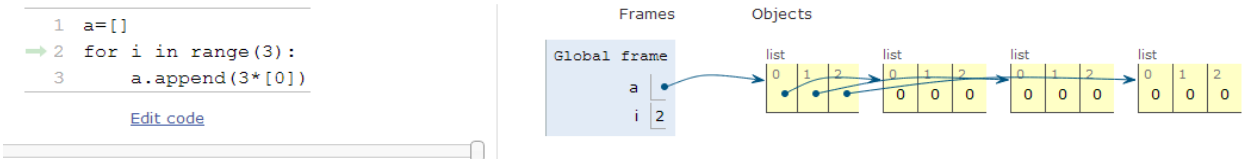
```
>>> a
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
>>>
```

However, internally they are represented very differently.

The first one produces the following when it runs:

What are the implications of this?

The second one, however, produces this:



What are the implications of this?