Looping Part 2

Objectives

- Using functions and methods for strings
- Write while-loops
- Use sentinel value to control a while-loop
- Using a `while`-loop to write a game loop
- Understand relationship between user inputs and events
- Read from a file
- Write to a file
- Use the `exit` function in the `sys` module to halt a program
- Use Python documentation

Strings

Remember strings? Here's one: `"boy, i'm broke"`

A string is really very similar to a list. The list [1, 2, 3] is made up of 1 and 2 and 3. The string `"hello world"` is made up of `"h"`, `"e"`, `"l"`, `"l"`, `"o"`, `" "`, `"w"`, `"o"`, `"r"`, `"l"`, and `"d"`.

Not only do they look the same, several "features" of a list is also available to a string. We will look into that in this section. But before we do that there are two things I want to talk about. The first is string input from the keyboard. The second is about characters.

String input from the keyboard is easy. For integer input you use the `input` function. For string input you use the `raw_input` function. Try this program:

```
x = input("enter an integer: ")
y = raw_input("enter a string: ")
print "x =", x
print "y =", y
```

Run this program entering `42` for `x` and your name for `y`.

That's all there is. You need two different functions because strings are not the same as integers. Note in particular that the string `"42"` and the integer `42` are different.

Let me first talk about things like `"a"`. If you have a list of numbers, for instance `[1,2,3]`, the 1 and the 2 and the 3 are values in the list. What about a string? The first value in the string `"hello world"` is `"h"`. The smallest unit of a string is just like a string of length one. These are called **characters**. Note that a space is consider a character. Each character is like something you see on the keyboard. So for instance `"$"` is a character. But there are more ...

There are special characters that you do not really "see" like the way you can see `"h"` or `"$"`. Try the following program:

```
print "1\n2\nbuckle my shoe"
```

You do not see \n in the output on your screen. But each time there is a \n there is a line skip. The \ and the n is a single character, not two. The `"\n"` is the **newline character**.

Another special character is `"\t"`, the tab character. Try this program:

```
print "table of cubes"
print "i\ti**3"
print "=\t===="
for i in range(10):
    print i,"\t", i**3
```

The special characters like `"\n"` and `"\t"` are called escape characters.

OK, enough of characters. We want to build strings which is made up of characters.

Before we go on, the list we will use can be long. Remember the \? When a statement is too long and you want to continue on the next line, you have to use \.

Again a string is like a list in the sense that they are both contains values laid out in a linear fashion.

Try this

```
x =  ["h", "e", "l", "l", "o", " ", "w", \
     "o", "r", "l", "d"]
y = "hello world"

print len(x)
print len(y)
```

AHA! You can compute the length of a string using the `len` function. This is just like a list.

Try this

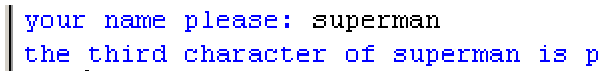
```
x =  ["h", "e", "l", "l", "o", " ", "w", \
     "o", "r", "l", "d"]
y = "hello world"

print "print x ..."
for c in x:
    print c

print "print y ..."
for c in y:
    print c
```

In fact the correct way to think of a list or a strings is to think of a “thingy” that contains values and you can scan across the values from the first to the last. These are called **sequence type objects**. They share a lot in common.

Exercise. There is the operator + on lists. (Don't remember it? No problem. Bring up IDLE and try it now and try + on two lists.) Can you use + on strings?

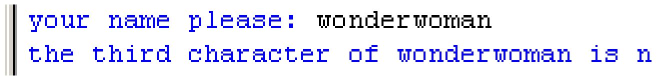
Remember what I said about the importance of jargon? You know some features of lists and I told you lists are sequence objects, you more or less know that strings should have more or less these features as well. It allows us to learn rapidly.

Exercise. Quickly review the `[]` operator on lists. Write a program that prompts the user for his/her name and prints the third character in the name. Here's what I get when I run my program:



```
your name please: superman
the third character of superman is p
```

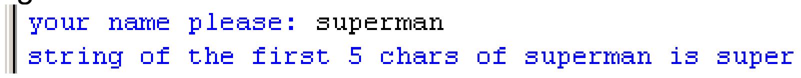
and for a second run:



```
your name please: wonderwoman
the third character of wonderwoman is n
```

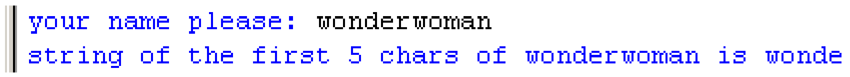
(Don't forget that the third character is at index position 2 because index values start with 0.)

Exercise. Quickly review the `[:]` operator on lists. Write a program that prompts the user for his/her name and prints the first 5 characters of the string. Here's the first run of my program:



```
your name please: superman
string of the first 5 chars of superman is super
```

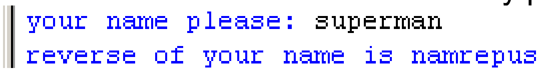
and here's a second.



```
your name please: wonderwoman
string of the first 5 chars of wonderwoman is wonde
```

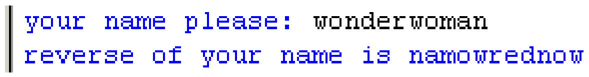
By the way a section of a string (all the characters from an index position to another) is called a **substring** of the original string. So for instance `"ello w"` is a substring of `"hello world"`.

Exercise. Quickly review the `[::]` operator on lists. Write a program that prompts the user for his/her name and prints the reverse of the name. Here's a first run of my program:



```
your name please: superman
reverse of your name is namrepus
```

and here's a second run:



```
your name please: wonderwoman
reverse of your name is namowrednow
```

Mutable and Immutable Values

There is however a **big difference** between a list and a string. You can change the value of an entry in a list. However you cannot do that for a string.

This concept is important enough to have another jargon. If you can modify the values inside a object (or variable), we say that the object is **mutable** (i.e. can change). Otherwise it is **immutable**.

A list is mutable.

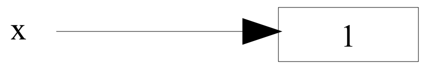
A string is immutable.

In fact an integer is also immutable. Now you scratch your head ... huh? ...surely you can change the value of an integer variable?!? ... like this:

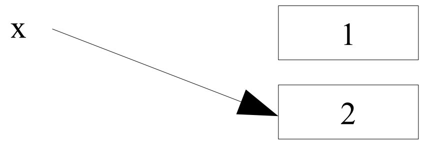
```
x = 1
x = 2
```

Ahhh ... good point.

Actually the second statement above does ***not*** change the value that `x` is referring to. It actually makes a ***new*** box for x. Here's a sequence of pictures that describes what happens when you run the above code. After executing the first statement Python has the following in its memory:



After executing the second statement we have this picture of Python's memory:

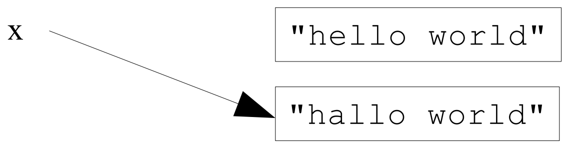


So how do you change a character in a string then? What if you want to change the value of `x` from `"hello world"` to `"hallo world"`? ***<u>You can't.</u>*** You have to create a new value for `x` to refer to. Here's how you do it.

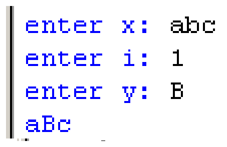
You basically take the part of the string before `e` and the string `"a"` and the part of the string after `e` in `"hello world"` and join them together like this:

```
x = "hello world"
x = x[:1] + "a" + x[2:]
print x
```

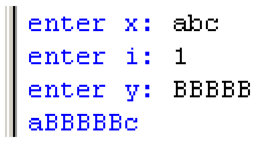
At the end of the second statement this is the memory of Python:



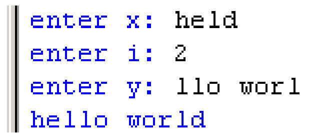
Exercise. Using the above program as an example, write a program that prompts the user for a string `x`, an integer `i`, another string `y` and prints `x` with the character at index position `i` replaced by `y`. Here are a couple of executions of my program:



```
enter x: abc
enter i: 1
enter y: B
aBc
```



```
enter x: abc
enter i: 1
enter y: BBBBB
aBBBBBc
```



```
enter x: held
enter i: 2
enter y: llo worl
hello world
```

Comparing Strings and Lists

You can compare integers and floats using ==, !=, <, <=, >, and >=. It shouldn't be surprising that you can compare strings and lists.

But before I go on, here's a warning for those of you learning other programming languages: For many programming languages the following might not work and in fact is the reason for many difficult to find bugs (errors).

OK. It's not too surprising that string comparison uses == just like integers:

```
x = raw_input("gimme your name: ")
y = raw_input("can you give that to me
again? ")

if x == y:
    print "ok ... just checking ..."
else:
    print "you changed your name in 5 secs?"
```

And you should try this:

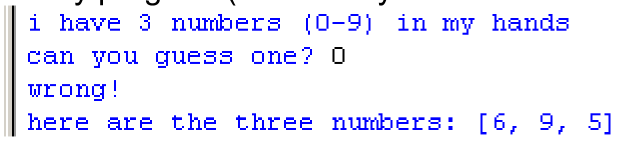
```
print [1,2,3] == [1,2,3]
print [1,2,3] == [1,3,2]
print [1,2,3] == [1,2,3,3]
print [1,2,3] != [1,2,3,3]
```

Not too surprising is it?

You can also check if a **value** is **in** a **list**:

```
print 42 in [1, 2, 42, 3]
```

Exercise. Create a list with three random integers from 0 to 9. Prompt the user to guess one of the numbers. Here are two runs of my program (I was lucky the second time):



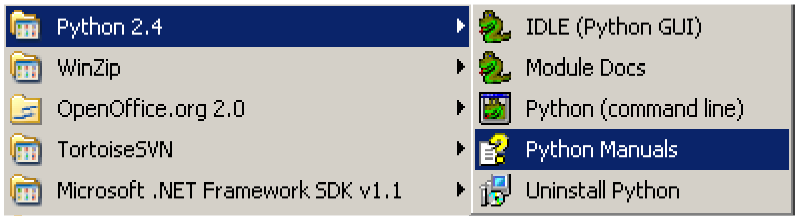
```
i have 3 numbers (0-9) in my hands
can you guess one? 0
wrong!
here are the three numbers: [6, 9, 5]
```

```
i have 3 numbers (0-9) in my hands
can you guess one? 1
you read my mind!
here are the three numbers: [4, 9, 1]
```

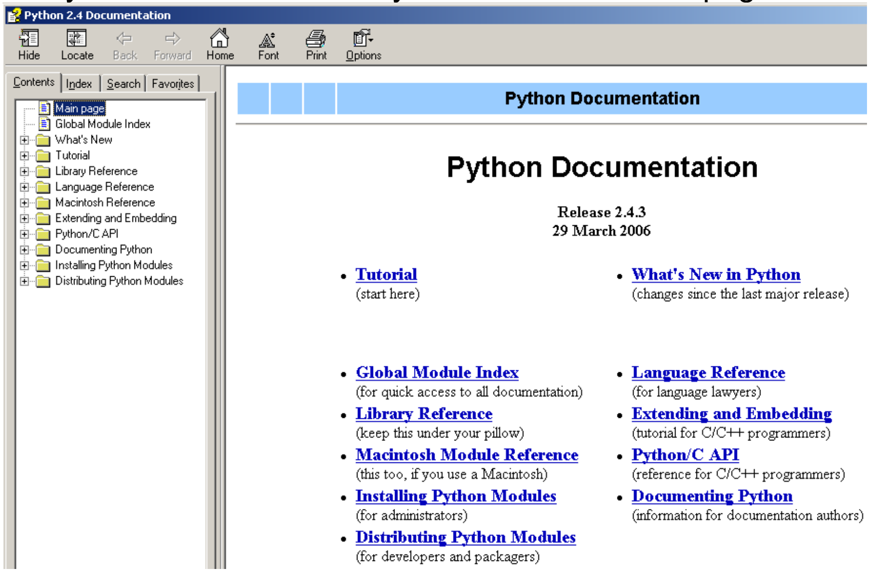
Documentation

Lists and strings have many “features”. I can't possibly tell you all. The best way for me to teach you to be a good programmer is to tell you how to obtain information on your own. The best Python information is already available on your PC!!! When you installed Python, you have also installed some documentation.

Let's bring up the documentation: Do Start > All Programs > Python 2.4 > Python Manuals:



and you will see the main Python Documentation page:



Python Documentation

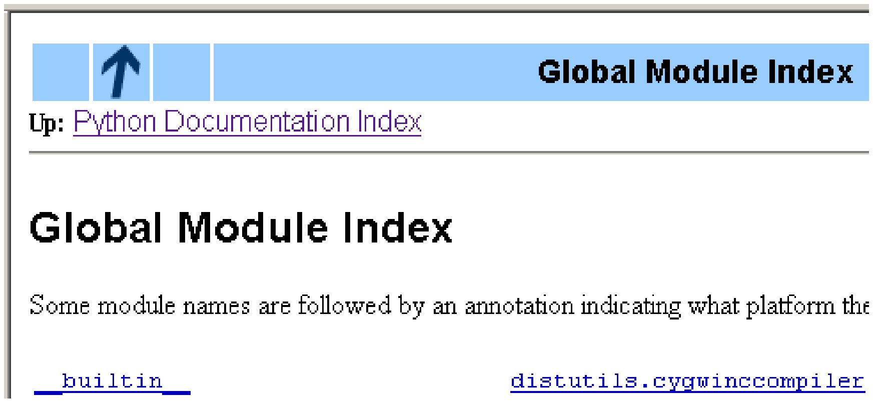
Release 2.4.3
29 March 2006

- Tutorial
  (start here)
- What's New in Python
  (changes since the last major release)
- Global Module Index
  (for quick access to all documentation)
- Library Reference
  (keep this under your pillow)
- Macintosh Module Reference
  (this too, if you use a Macintosh)
- Installing Python Modules
  (for administrators)
- Distributing Python Modules
  (for developers and packagers)
- Language Reference
  (for language lawyers)
- Extending and Embedding
  (tutorial for C/C++ programmers)
- Python/C API
  (reference for C/C++ programmers)
- Documenting Python
  (information for documentation authors)

The left side bar includes an “Index” and a “Search” feature.

You are very strongly encouraged to spend a couple of minutes browsing through the documentation when you have the time. Don't worry if you don't understand everything you read. The best place to start is the first item on the page “Tutorial”.

Let's try to find out more about one of the first modules you used: `random`. On the main Python Documentation, click on "Global Module Index" and you should see this:



Global Module Index

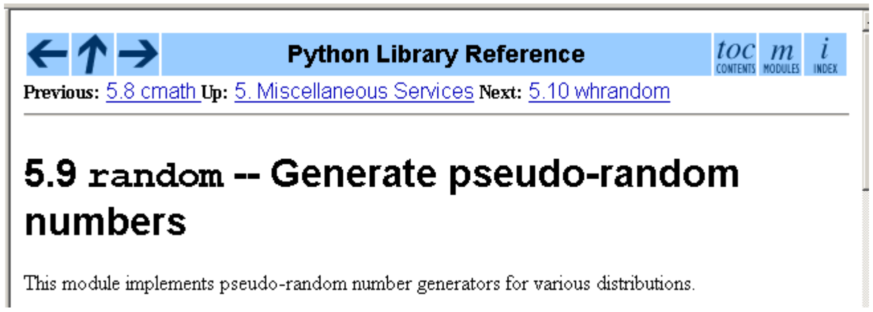
Up: Python Documentation Index

Global Module Index

Some module names are followed by an annotation indicating what platform the

__builtin__ distutils.cygwinccompiler

Now look for and click on `random`. You will see:



Python Library Reference

Previous: 5.8 cmath Up: 5. Miscellaneous Services Next: 5.10 whrandom

5.9 random -- Generate pseudo-random numbers

This module implements pseudo-random number generators for various distributions.

Now you can read all about the `random` module. Remember `random.seed` and `random.randrange`? Look for them now and quickly read it ...

Now here's an exercise for you ...

Exercise. Recall a previous exercise. I need to select a random name from a list? Here's the answer:

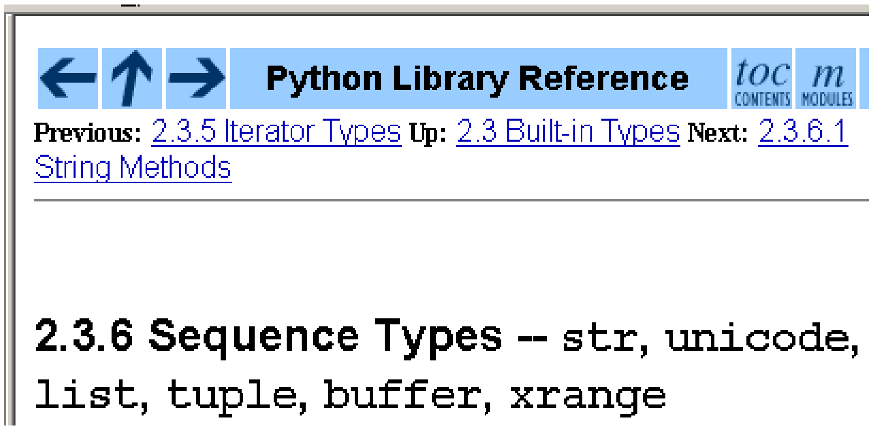
```
import random
random.seed()

names = ["andrew","brad","daniel","josh", \
         "nick","rosalie","thureen"]
i = random.randrange(len(names))
name = names[i]
print name
```

Using the documentation for `random` (see above), look for the `choice` function and redo the above problem.

But what if the thing you want to know more about is not in a module. For instance you do not need to import anything to create a list, right? Where do you go to learn more about lists?

You go to the main page of the documentation and click on “Library Reference”.

Exercise. Using Python's documentation, look for the “Library Reference” and then look for the “Sequence Types”.



Previous: 2.3.5 Iterator Types **Up:** 2.3 Built-in Types **Next:** 2.3.6.1 String Methods

2.3.6 Sequence Types -- str, unicode, list, tuple, buffer, xrange

On the page, there is a list of 10 operations you can carry out on objects of sequence types including lists and strings. Using one of the operations, complete the following program that prints the ***largest*** value in a list of integers:

```
x = [3, 1, 5, 2, 42, 6]
y = ___________
print y
```

Sorting a List

I've told you that a list is mutable sequence. Go to the main documentation page again. Once again go to the “Library Reference”, look for “Sequence Types” and then look for mutable sequence types. Look for the `sort` function. On the page, you see this description of sort:

```
     s.sort([cmp[, key[, reverse]]])
```

with notes.

What are those brackets? When you see [ ] in the documentation it means that the stuff in the [ ] is optional. So the innermost `[, reverse]` is optional. This means that you can execute one of the following:

```
     s.sort([cmp[, key, reverse]])
     s.sort([cmp[, key]])
```

Applying the same argument again and again, you see that there are the following ways to sort:

```
     s.sort(cmp, key, reverse)
     s.sort(cmp, key)
     s.sort(cmp)
     s.sort()
```

I'll only talk about the last one.

Try this:

```
x = [1, 3, 5, 2, 1]
print x
x.sort()
print x
```

Exercise. Is `x` sorted in ascending or descending order?

You can actually also sort on a list of lists. A list of lists is just a list where the values are lists (duh).

Try this:

```
x = [[1,2], [3,2], [-2,3,3], [42,1,2]]
```

```
print x
x.sort()
print x
```

You will see that the resulting `x` is sorted by the ***first*** entry of the lists in x.

Exercise. But what if there are two entries in the list with the same first entries? For instance

```
x = [[1,2], [1,1], [1,3,3], [1,1,2]]
```

Notice that all lists in the list have the same first entry, i.e. 1. What will happen when you sort x?

Exercise. Quickly review your notes on tuples. A tuple is like a list. For instance while `[1,2]` is a list, `(1,2)` is a tuple. Can you sort a list of tuples? (First make a guess. Second use the above program, change all the lists in `x` to tuples, and run the program.)

Exercise. Can you sort a string? (First make a guess. Second verify with Python).

while-loop

The while-loop is very similar to the for-loop. Just to refresh your memory, here's an example of a for-loop:

```
print "about to execute for-loop ..."
for i in [1, 3, 5]:
    print i
print "done with for-loop"
```

In the case of a **for-loop**, you have a **control variable** that runs across a **list** of values. Each time the control variable picks up a value from the list, you execute the body of the for-loop. When there are no more values, Python stops executing the for-loop and begin executing the statement immediately after the for-loop.

Here's an example of a while-loop. Make sure you run it!

```
i = 0
while i < 5:
    i = i + 1
    print "i =", i

print "\ndone with while-loop ... i =", i
```

Study it and guess what's happening.

Here's another

```
i = 5
while i >= 0:
    i = i - 1
    print "i =", i

print "\ndone with while-loop ... i =", i
```

Do the same.

And another

```
score = 0
points = input("enter points: ")
while points != 0:
    score = score + points
    print "score: ", score
    points = input("enter points: ")
```

```

print "\ndone with while-loop ... "
print "score: ", score
```

OK let me explain what's happening. Here's the first program with the `while`-loop:

```
i = 0
while i < 5:
    i = i + 1
    print "i =", i

print "\ndone with while-loop ... i =", i
```

Adding a few print statements to make Python tell you what it does while running the program we have this:

```
i = 0
print "about to check condition ..."
while i < 5:
    print "\ncondition True, execute body"
    print "i =", i
    i = i + 1
    print "i =", i
    print "about to check condition ..."

print "\ndone with while-loop ... i =", i
```

Run it.

Now use the following explanation of the `while`-loop to study this program and understand why you get the output ...

For a **while-loop** there is a **control boolean expression**. When Python first enters the `while`-loop, it evaluates the boolean expression. If the boolean expression evaluates to `True`, Python will execute the body of the `while`-loop. At the **end** of executing the body, Python will go to the top of the `while`-loop and check the value of the boolean expression again. If it's `True`, it will execute the body. This goes on until the boolean condition is `False`. Once it is `False`, Python will exit the `while`-loop and execute the rest of the program starting with the statement right after the `while`-loop.

Try this out:

```
while False:
    print "do you see me?",
print "done"
```

This shows you that the condition is checked before the body is executed.

And this is a common trick:

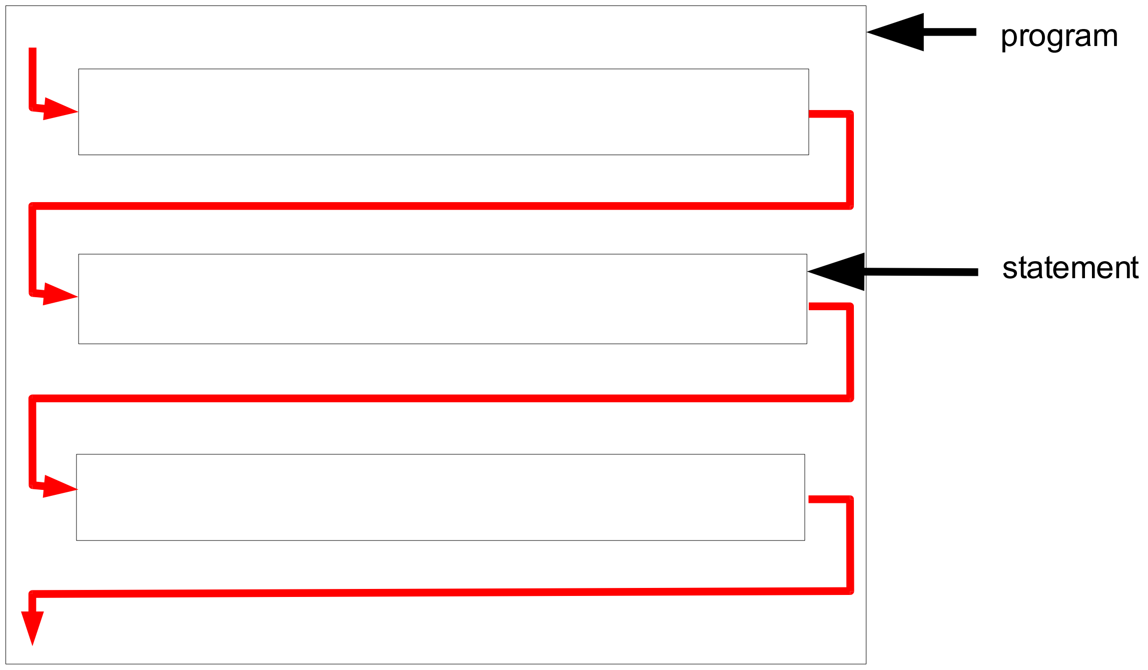
```
while True:
    print "do you see me?",
print "done"
```

Of course the second program above tells you that there's no way for the program to stop. If you're ever in this situation, do a Ctrl-c (hold down the Ctrl key and press c) to stop a “runaway” program.

Flow of Execution

Here's the big picture when you run a program and I'm only focusing on how Python picks which statement to execute. Right now I'm not interested in Python's memory, etc. I'm interested in the flow of execution. You can think of Python as having a finger that points to the next statement it should run.

First of all Python will execute one statement at a time from the first to the last:



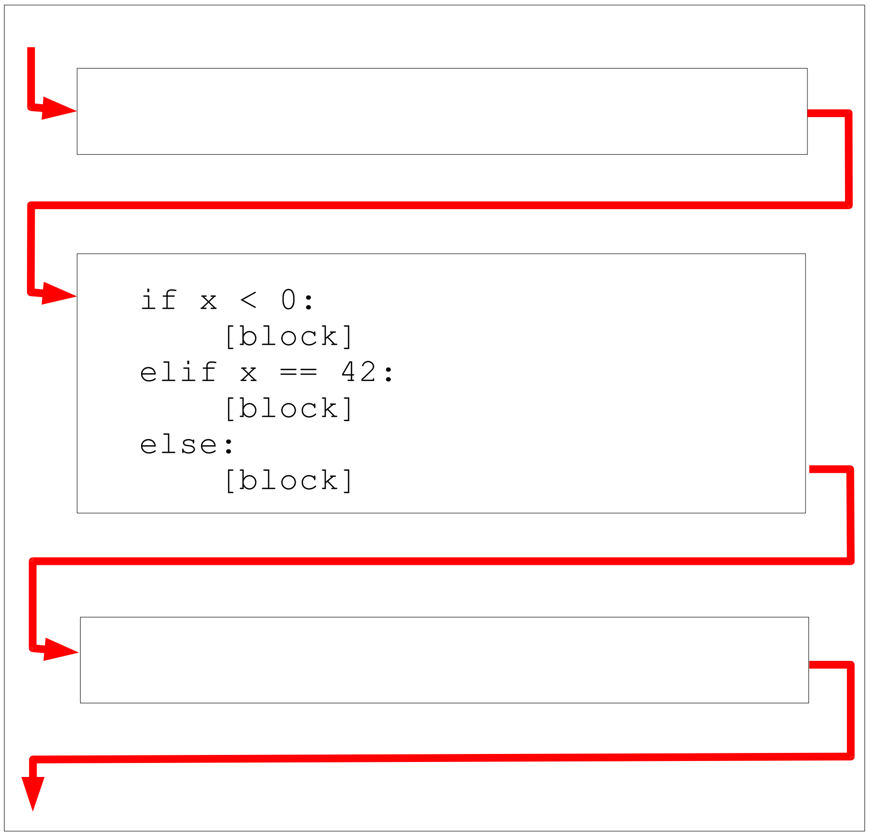
Note that although you can think of the above bunch of statements as a program, it's actually better to think of it as a block of statement.

So ... one of the first programs you wrote was the hello world program which prints hello world. The program has only one statement. You can think of that program as a block with one statement in the it.

Now if the second statement is an if-elif-else statement such as:

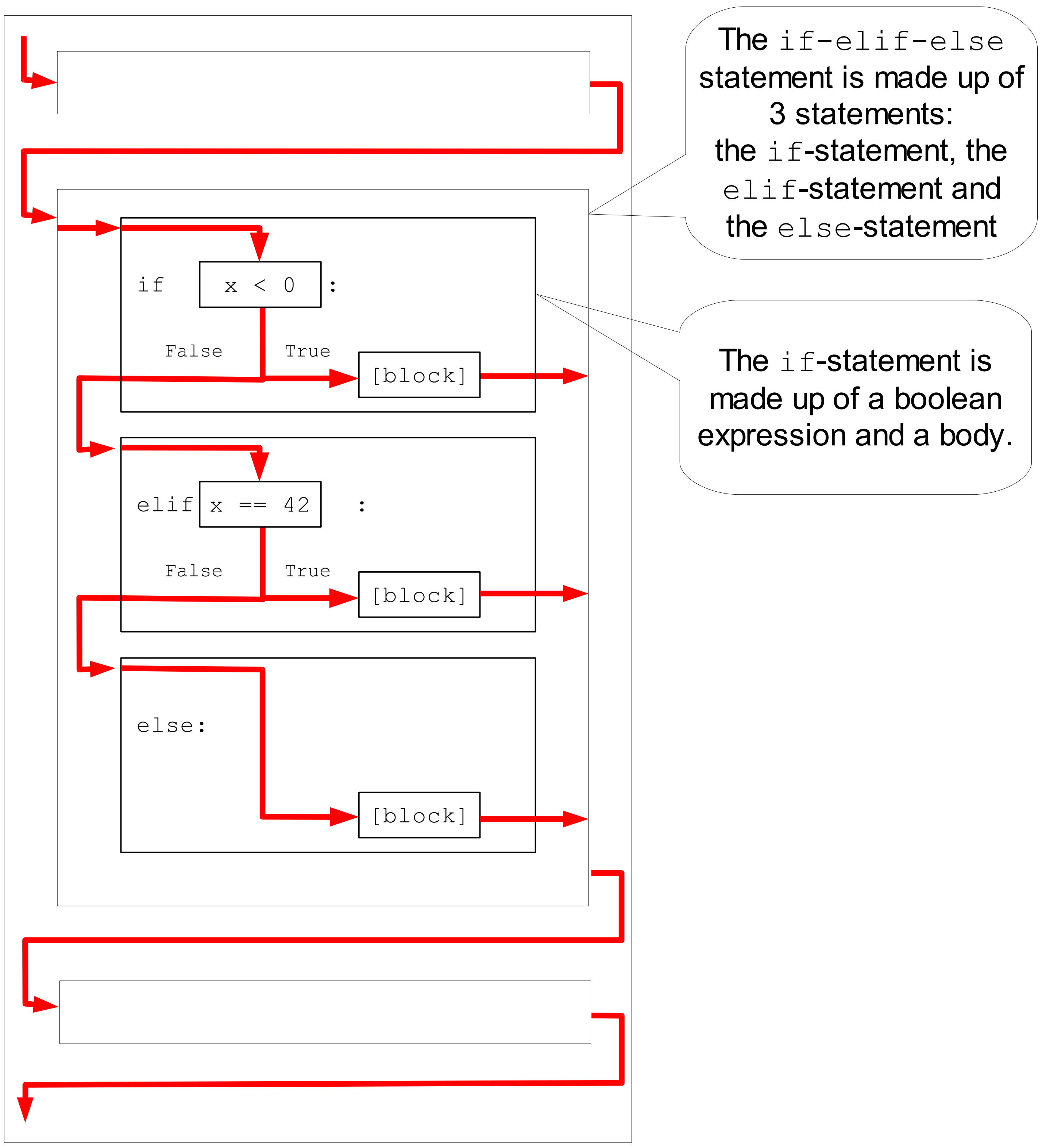
```
if x < 0:
    [block]
elif x == 42:
    [block]
else:
    [block]
```

you will have the following structure for the flow of execution in your program:



```
if x < 0:
    [block]
elif x == 42:
    [block]
else:
    [block]
```

But once you enter the second statement, what is the flow of execution? It looks like this:


if
x < 0
:
False
True
[block]
elif
x == 42
:
False
True
[block]
else:
[block]
The if-elif-else statement is made up of 3 statements: the if-statement, the elif-statement and the else-statement
The if-statement is made up of a boolean expression and a body.

Exercise. Modify the program

```
    if x < 0:
        [block]
    elif x == 42:
        [block]
    else:
        [block]
to get this:
```

```
x = -100

if x < 0:
    print "block 1"
elif x == 42:
    print "block 2"
else:
    print "block 3"

print "out of if-elif-else"
```

First guess what output you will get with this program using the above picture of the flow of execution if necessary. Second run the program and verify your guess. Repeat this exercise with `x` set to 0, 5, 42, and 50.

Exercise. What is the output of the following program?

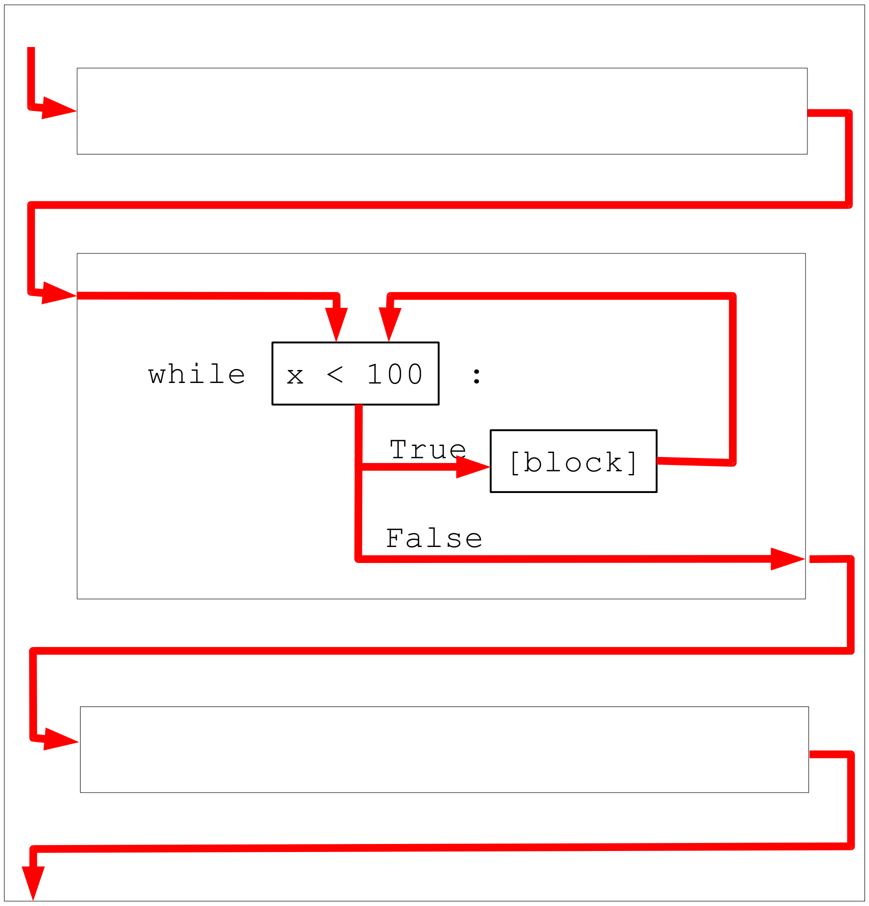
```
x = 42

if x < 0:
    print "a"
elif x == 42:
    print "b"
elif x == 43:
    print "c"
elif x == 42:
    print "b"
else:
    print "d"
```

Do you have something to complain about this program?

I'll skip the picture for the `for`-loop. The picture of the `while`-loop looks like this. Suppose the second statement is a `while`-loop of the form:

```
    while x < 100:
        [block]
```

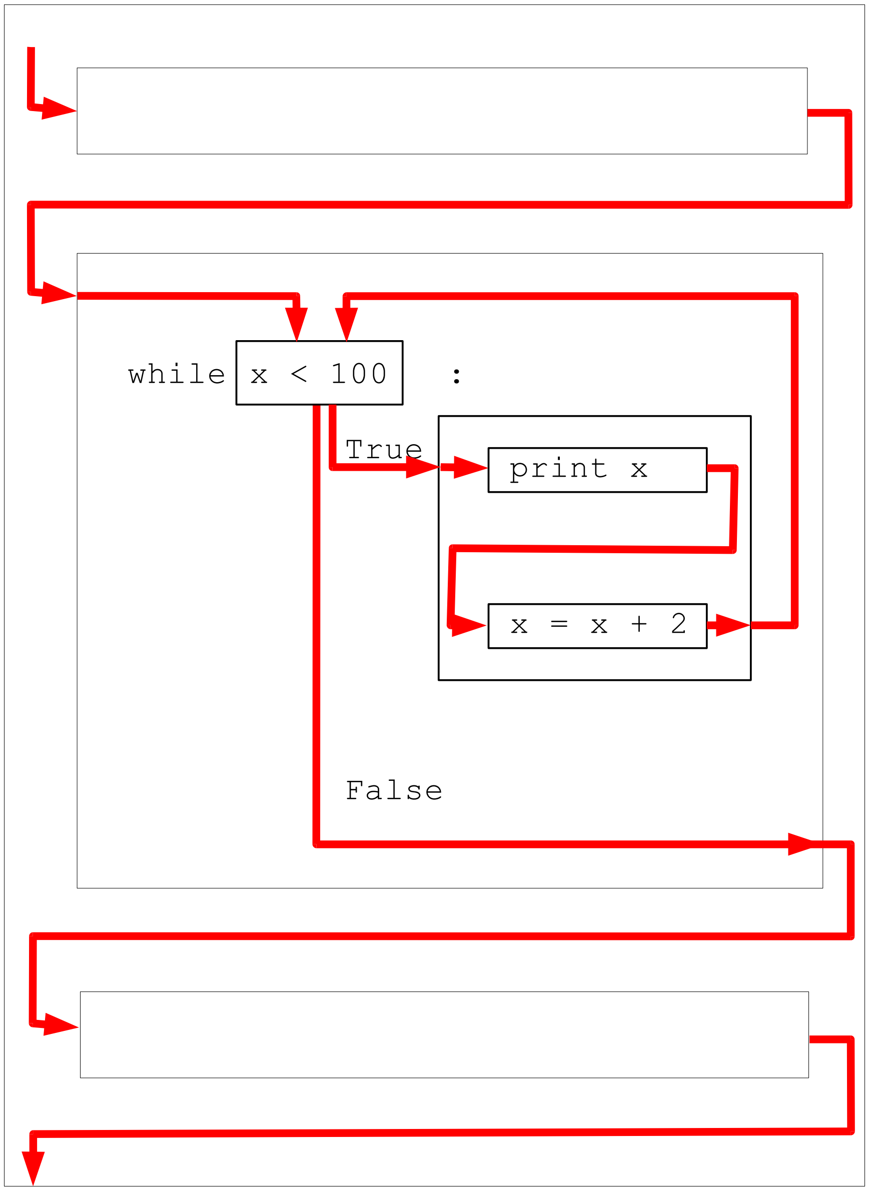
then we have this picture:



And of course [block] can be any block of statements. So for instance if the block is made up of the following two statement:

```
    print x
    x = x + 2
```

we have the following picture:



Note in particular when Python enters the body of the `while`-loop it executes the statement one at a time.

Exercise. You are given this program:

```
x = 95

while x < 100:
    print x
    x = x + 2

print "outside while-loop"
```

First using the above diagram, figure out the output. Next verify with Python by running the program. Repeat the exercise by giving `x` an initial value of 100.

Don't worry if the diagram looks overwhelming. After a while you will be able to read programs without even thing about them. It simply takes time.

A very Common Gotcha

Here's a common gotcha for new programmers.

Note that in a `while`-loop the control condition is checked only at the beginning and at the end of executing the body of the `while`-loop. It is NOT checked while Python is executing the body of the `while`-loop.

For instance look at this program. The boolean condition “x is not zero”. But notice that x is temporarily set to 0 in the body. Does Python exit the `while`-loop?

```
x = 1

while x != 0:

    print "setting x to 0"
    x = 0 # Does Python exit the while-loop
          # here?

    print "do you see this line?"
    x = 1

print "this is after the while-loop"
```

Sentinel Values

Don't rush through this section.

Exercise. Read the following program:

```
x = input("Gimme a number: ")
while x != 0:
    x = input("Gimme a number: ")

print "phew!"
```

Try to guess how Python will execute the program. Now run the program. Enter 42. Read the output. Try a few more numbers. What must you enter to exit the `while`-loop? (And I'm not talking about closing the program or doing Ctrl-c!)

Let's have another example. Run this program:

```
xs = []

x = input("enter integer (0 to stop): ")
while x != 0:
    xs.append(x)
    print "xs: ", xs
    x = input("enter integer (0 to stop): ")

print "\nfinal xs: ", xs
```

Now analyze the program carefully. Run the program as many times as you need. Do not rush. Ask questions if you need help.

In general if you continually prompt the user for a value to process and you want to process the input when the input from the user satisfies some condition (in the above case the condition is the input is non-zero), otherwise you stop the loop, then the format of your program will look something like this:

```
some optional initialization

prompt the user for value
while condition on value is true:
    process the value
    prompt the user for value
```

Note in particular you should see two places where the program prompts the user for a value.

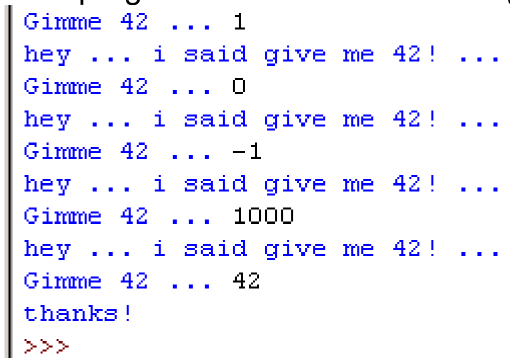
The special value that controls when the `while`-loop should stop is called a **sentinel value** (i.e. a guard). In the above examples the sentinel value is 0.

Exercise. Using the above format, analyze the previous program

```
xs = []

x = input("enter integer (0 to stop): ")
while x != 0:
    xs.append(x)
    print "xs: ", xs
    x = input("enter integer (0 to stop): ")

print "\nfinal xs: ", xs
```

Do you see the initialization part? The condition? The processing part? Do you see the prompt for value twice?

Exercise. Write a program that does the following



```
Gimme 42 ... 1
hey ... i said give me 42! ...
Gimme 42 ... 0
hey ... i said give me 42! ...
Gimme 42 ... -1
hey ... i said give me 42! ...
Gimme 42 ... 1000
hey ... i said give me 42! ...
Gimme 42 ... 42
thanks!
>>>
```

Exercise. Recall the following multiplication game:

```
import random
random.seed()

x = random.randrange(90, 101)
y = random.randrange(90, 101)
print "What is the", x, "*", y, "?"
guess = input("Answer: ")
product = x * y
if guess < product:
    print "Incorrect! Too low!"
    print "The answer is", product
```

```
elif guess == product:
    print "Correct!"
else:
    print "Incorrect! Too high!"
    print "The answer is", product
```

Rewrite this so that the game continually generates a question for the user and prompts him/her for a guess until the user answers the question correctly.

Breaking out of the `while`-loop

Just like in the case of the `for`-loop, if Python executes a `break` statement in a `while`-loop, it will immediately exit that `while`-loop. Try this:

```
i = 0
while True:

    if i == 5:
        print "about to execute break ..."
        break

    i = i + 1
    print "i =", i

print "done!"
```

Observe your output on your screen. Read the program carefully and understand why you get the output. Note in particular the `break` statement.

So there are two different ways to exit a `while`-loop: either the control boolean expression evaluates to `False` or Python executes a `break` statement.

Bouncing Alien: `while`-loop and events

Now you're ready to see how the while-loop, for-loop and lists are used in `bouncing_alien.py`. I will also talk about events.

Go ahead and open `bouncing_alien.py`.

You notice that the `while`-loop in the program looks like this:

```
while 1:
    [body stuff]
```

Why is there an integer in the control boolean expression?!? Shouldn't the control be a boolean expression?

Well the reason is that `1` is converted to `True`. In fact any value passed into the `while`-loop as the control boolean expression will be converted to a boolean value. In particular the value 0 is converted to `False`; any other value is converted to `True`.

So you now know that the control boolean expression does not exit the `while`-loop. So how is the program stopped?

Look for this in your program:

```
    # Exit if window is closed
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
```

Let's analyze this code segment.

First from

```
pygame.event.get()
```

you can tell that there is a module called `event` in the `pygame` package. In `event` there is a function called `get`. The output of this function just be a list since it's used in this manner:

```
for event in pygame.event.get():
```

The list contains events. Now what are events?

You can think of **events** as system input signals. When you move the mouse, an event is created. When you close a window (by clicking on the top-right X button of the window) an event is created. Events are temporarily stored somewhere (never mind where ... it's not important to us).

Now `pygame.event.get()` will remove the stored events and put them into a list. `for`-loop let the variable `event` run over the events of this list. So the control variable of the `for`-loop is `event`. In the body of the `for`-loop, Python executes this:

```
        if event.type == pygame.QUIT:
            sys.exit()
```

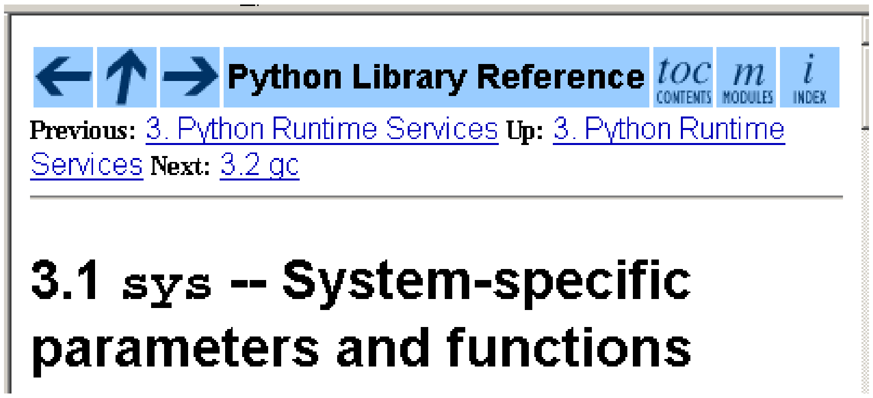
This checks if the event is the one that closes it the game window. If that's the case, then a function is called:

```
sys.exit()
```

The function is `exit` and it's in the `sys` module.

Exercise. Whoa ... wait ... has `sys` been imported in the program?

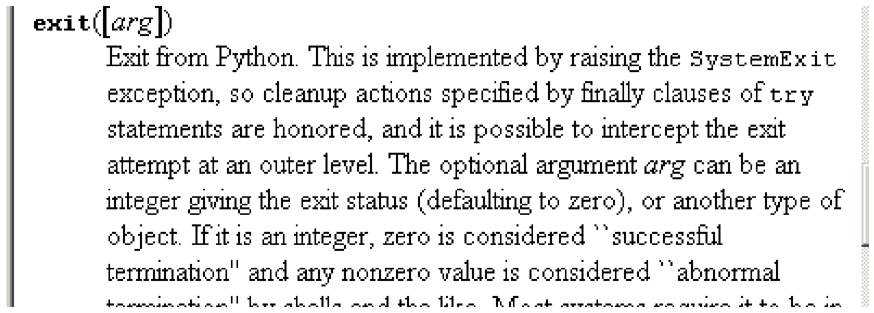
Exercise. Look for more information on the `sys` module. Make sure you can at least get to this point in the documentation:



Previous: 3. Python Runtime Services **Up:** 3. Python Runtime Services **Next:** 3.2 gc

3.1 sys -- System-specific parameters and functions

and make sure you can find the documentation of the `exit` function:



exit([*arg*])

Exit from Python. This is implemented by raising the `SystemExit` exception, so cleanup actions specified by finally clauses of `try` statements are honored, and it is possible to intercept the exit attempt at an outer level. The optional argument *arg* can be an integer giving the exit status (defaulting to zero), or another type of object. If it is an integer, zero is considered ``successful termination'' and any nonzero value is considered ``abnormal termination'' by shells and the like. Most systems require it to be in

The `sys` module is extremely low level and can be hard to understand. So let me explain ...

The `exit` function is actually pretty easy. It stops the program. It doesn't even bother exiting the `while`-loop!!! Whenever Python executes `sys.exit()`, in an `if`-statement or `for`-loop or what-have-you, the program simply stops.

This will explain everything:

```
import sys

i = 0
print "going into while-loop ..."
while 1:
    print i
    if i == 42:
        sys.exit()
    i = i + 1

print "do you see this line?"
```

Run this program. Analyze the program.

There are other types of events including those related to the mouse and joystick. We don't need them right now.

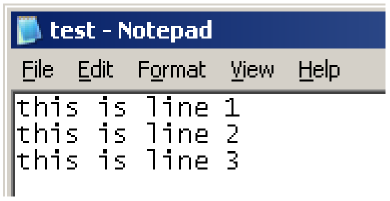
Files

There is one other type of sequence objects I want to talk about: files. What is a file a sequence of? It's a sequence of the lines in the file.

If you want to write a game that records high scores with player names, then you will need to know how to read and write to files on your hard drive.

Let's write a program that reads a file. First use the Notepad to create a text file. (Notepad is a text editor. It allows you to create text files. You should be able to find it from the Start menu. You can usually find it at Start > Programs > Accessories > Notepad.)

Enter the following data into the file and save it as test.txt on the Desktop:



```
this is line 1
this is line 2
this is line 3
```

Make sure you hit the enter key at the end of line 3. Check your Desktop (after saving the above file) to make sure it's there.

Now save the following program on your Desktop.

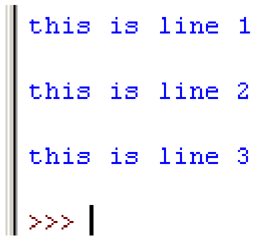
```
f = file("test.txt")

for line in f:
    print line
```

Let's read this program.

The first statement opens a file called test.txt and associates `f` with this file. `f` becomes a file variable. That is how you open a file.

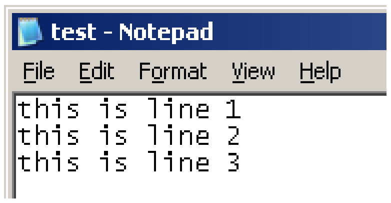
The variable `line` runs over the file `f` in the `for`-loop. If `line` has length 0, Python exits the `for`-look because of the `break` statement. Otherwise the `line` is printed.

Here's the output:



```
this is line 1

this is line 2

this is line 3

>>> 
```

Why are there blank lines between each line? It's because Notepad records newlines. When you entered the following:



```
this is line 1
this is line 2
this is line 3
```

there are line skips at the end of each line. Notepad records these line skips as newlines. In other words the first line when read by Python is actually

```
"this is line 1\n"
```

Let's get rid of the last character when we read the text file. Modify the program as follows:

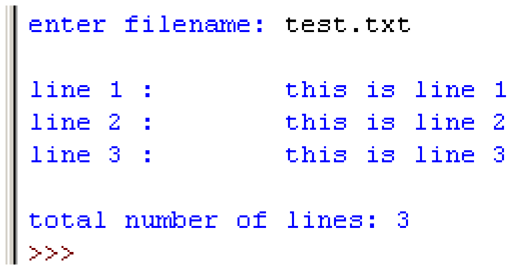
```
f = file("test.txt")

for line in f:
    line = line[:-1]
    print line
```

It is clear from the program that one can view `f` as a sequence – it's a sequence of the lines in the file.

Also, the name of the file can be from a string variable. For instance the first line of the above program can be replaced by the following two line:

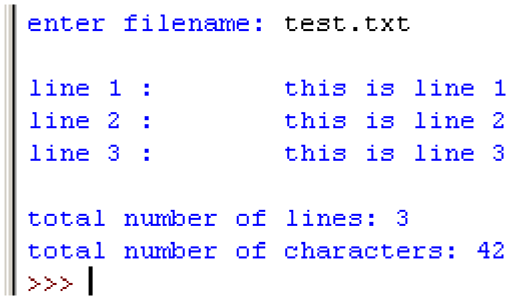
```
filename = "test.txt"
f = file(filename)
```

Exercise. Write a program that prompts the user for the name of a file, opens the file, and prints the lines with their line number. After print all the lines in the file, the program prints the total number of lines in the file. Here's an execution of my program:



```
enter filename: test.txt

line 1 :         this is line 1
line 2 :         this is line 2
line 3 :         this is line 3

total number of lines: 3
>>>
```

Exercise. Improve the above by printing the number of characters in the file.



```
enter filename: test.txt

line 1 :         this is line 1
line 2 :         this is line 2
line 3 :         this is line 3

total number of lines: 3
total number of characters: 42
>>>
```

(The number of characters does not include the newline characters.)

Finally this is how you ***write*** data to a file. Run this program:

```
f = file("test.txt", "w")

f.write("abc")
f.write("def")
f.close()
```

and then open the file test.txt and see what data is written to it.

Be careful! There is a statement to close the file:

```
f.close()
```

Exercise. What if you run the program without the statement to close the file?

What if you want to ***append*** to a file? In other words you want to retain the contents and whatever you write to the file is added to the end of the file. You do the same as above except that you open the file this way:

```
f = file("test.txt", "a")
```