Looping

Objectives

- Declare list variables
- Access an entry in a list using an index value
- Perform list concatenation using +
- Append new entries into a list
- Use the `range` function to create a list
- Obtain the length of a list using the `len` function.
- Use the slice operator to create a sublist of a list
- Write `for`-loops
- Use the `clock` function the `time` module to measure time taken to execute a block of statements
- Use floating point values and their operators
- Use tuples
- Understand the difference between tuples and lists
- Use Python to check if an integer is prime
- Use Python to generate a list of primes

Review and the Big Picture (DIY)

You notice that I've already mentioned some programming jargon. Sometimes as a programmer you might think that the only thing you need to know is how to write programs.

It's certainly true that the true test of a software engineer is in the quality of the work he/she produces. But on the other hand jargons are important because they represent important concepts that Computer Science people use in order to communicate with each other.

Learning is extremely important for anyone who wants to be the best in his/her area – whatever area that may be. If you want to be one of the best software engineers or software architects, you will need to be able to learn new things rapidly to stay ahead. Jargon and general concepts are crucial to rapid learning.

And of course, if you are not familiar with the jargon, besides the fact that you can't communicate with others who use a particular jargon, you will probably come up with your own jargon and therefore you will in fact confuse others.

And even worse ... you would feel really bad when the "others", after a long and confusing conversation with you, tell you that you are using the wrong terms.

Let me give you an example. Take for instance you now know that in Python, to print something you use `print` and not `Print` or `PRINT`. We say that Python is **case-sensitive**. If you know that the programming language Java is case-sensitive, and printing involves something call `System.out.println`, then you know immediately when you read a program with `System.Out.Println`, that it is very likely an error.

Exercise. What about variables? Are variables in Python case sensitive too? Write a program that creates a variable `abc` giving it `42`. Print `ABC`.

So you see, general concepts (or jargon) create a language for us to learn rapidly. Now I will quickly summarize a few

important concepts, some of which you have already seen. Most of these concepts are common to all programming languages.

A **reserved word** is a word that has a special meaning to Python. For instance `print` is a Python reserved word.

Exercise. What other Python reserved words have you seen so far?

An **identifier** is just a name. For instance a variable has a name. A function has a name. A module has a name. An object has a name. Etc. In Python an identifier is made up of the underscore and **alphanumerics** (i.e. letters from the alphabet or digits) however the first character cannot be a digit. Note that an **identifier cannot be a reserved word**.

These rules for creating identifiers are pretty general and apply to most programming Languages. There are some exceptions. For instance some languages allow the $ sign.

Exercise. Remember what I said about verifying everything I tell you? What do you think you need to verify right now?

Variables are assigned values using the **assignment operator =**.

The jargon for "creating" a variable is **"declaring"** a variable.

Values supported by Python include strings, integers and boolean values. Integer operators include +, -, *, /, %, **. Comparison operators are ==, !=, <, <=, >, >=. The boolean values are `True` and `False`. Boolean operators include `and`, `or`, `not`.

Many other programming languages do not have exponentiation operators. Also `True`, `False`, `and`, `or`, and `not` are written in different ways.

Python is a **dynamically typed** programming language. This means that the value that a Python variable refers to can be any type of value. Try this:

```
x = 0
x = "hello world"
```

Do you get an error? This shows that x can refer to an integer at one time and then to a string at another time.

(This is not possible for a statically typed languages such as C, C++, and Java.)

A **block** of statements is a group of consecutive statements with the same indentation.

An **arithmetic expression** contains integer values, integer variables and integer operators that can be evaluated to give an integer value. A **boolean expression** on the other hand contains boolean values (`True`, `False`), boolean variables and boolean operators (`and`, `or`, `not`) that can be evaluated to give a boolean value.

Note that the if-statement looks like this:

```
if [condition]:
    [statement]
```

But since the `if`-statement is a statement this implies that you can also so this:

```
if [condition]:
    if [condition]:
        [statement]
```

Likewise for the if-else statement. For instance you can have

```
x = input()
if x < 0:
    if x == -1:
        print "-1"
    else:
        print "..., -3, -2"
else:
    if x == 0:
        print "0"
    else:
        print "1, 2, ..."
```

(Make sure you run this program with different values for x. Make sure you understand what's happening.)

The `if`-statement is a compound statement: it's made up of two parts. There's the **header** and the **body**. For instance, for the `if`-statement:

```
if x == 0:     # this is the header
    a = 42     # this is the body
```

The body can be a block of statements.

```
if x == 0:     # this is the header of it
    a = 42     # this is the body
    b = 24     #     of the if
else:          # this is the header for else
    a = 41     # this is the body
    b = 22     #     of the else
```

What is the best way to learn? The best way to learn is to discover things for yourself. So instead of telling you everything I will try to give you examples and see if you can discover things for yourself.

Let's move on to the next major tool in programming languages.

Why Loops?

What are loops? Well, a loop is just something that will **repeat** a statement or a bunch of statements.

Why do we need loops? Because we are too lazy and way too smart to do redundant work.

For instance look at the multiplication game. Instead of asking a single multiplication question:

```
import random
random.seed()

x = random.randrange(90, 101)
y = random.randrange(90, 101)
print "What is", 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
```

what if you want to ask two questions? You can copy and paste your code:

```
import random
random.seed()

x = random.randrange(90, 101)
y = random.randrange(90, 101)
print "What is", 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

x = random.randrange(90, 101)
y = random.randrange(90, 101)
print "What is", 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
```

Exercise. Make sure you run the above program. Why did I ***not*** copy and paste the statements `import random` and `random.seed()`?

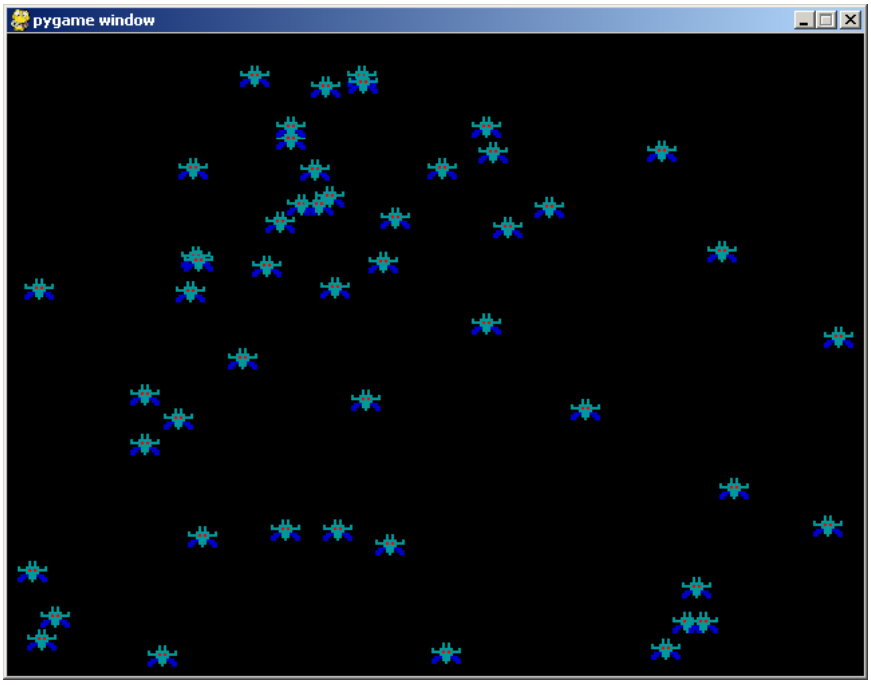
Great ...

But ... what if you want to ask 10 questions instead? Or, what if you allow the user to specify how many questions he/she wants to answer?

That's when you need to know how to write loops.

Or take for instance a computer game. Most game with graphics involves continually moving an image by a small amount to simulate smooth motion. In this case you want to repeat each small change until the user stops the game or when the game ends.

Or take our bouncing_alien.py program. What if you want to have 50 aliens:



If you know how to write loops the modification to bouncing_alien.py is minimal. This will be one of the projects in this set of notes.

In a business application, let's say a payroll program, the program will process the pay for each employee kept in a file until all records are read. (Never mind if you don't understand what a file is. The point is that you need repetitions.)

Anyway, you will need a way to tell Python to continually execute a statement or a block of statements until a condition is reached.

Python understands two different types of loops: the `for`-loop and the `while`-loop. We'll talk about the `for`-loop first. But before we talk about the `for`-loop we will need to talk about lists.

Lists

An integer value such as 42 is sort of like an atom. You can build a bunch (molecule?) of integers like this:

```
>>> 42
42
>>> [1, 2, 42]
[1, 2, 42]
>>>
```

`[1, 2, 42]` is a **list**. In particular it's a list of integers. Just like an integer, you can give a list to a variable:

Exercise. Verify with Python! Declare a variable and assign it `[1, 2, 42]`. Print that variable. Does it work?

You can also put strings into a list. And you can also mix integers and strings:

```
>>> [1, 2, "buckle my", "shoe"]
[1, 2, 'buckle my', 'shoe']
>>>
```

Besides putting a bunch of things together to create a list, you can pull a value **out** of a list:

```
>>> x = [1, 2, 42]
>>> print x[0]
1
>>> print x[1]
2
>>> print x[2]
42
>>>
```

So if `x` is a list `x[0]` refers to the first thing in the list, `x[1]` refers to the second, etc. The important thing to remember is that **the first thing in the list is `x[0]`** and **not `x[1]`**.

Exercise. Suppose `x` is the list `[1, 2, "buckle my", "shoe"]`, what is `x[1]`? `x[3]`? Verify using Python!

You can think of a list `x` as a building with rooms and the value you put into the [ ] as a room number within the building. Remember the first entry in the list is room number 0 (not 1!!!). The value you put into the [ ] is called an **index value**. So for the list `[1, 2, 42]`, I would say `42` is at index `2` or index position `2`.

Exercise. Can a value occur at two different index positions? Verify with Python.

In fact you don't need to restrict yourself to integer values as index values. You can use expressions too, as long as the expression evaluates to an integer value.

Exercise. Guess the output of the program:

```
x = [1, 2, 42]
i = 1
print x[i]
print x[i - 1]
print x[i*i + 1]
```

Verify using Python.

Not only can you use values from the list as above, you can modify the value at any index position:

Exercise. First guess the output of this program:

```
x = [1, 2, 42]
x[0] = 3
print x
a = 41
i = 2
x[i] = 2 * a + i
print x
```

Next verify with Python.

You can also create a list with nothing:

```
zippo = []
print zippo
```

`[]` called an **empty list**.

Exercise. What if you go beyond the list? For instance create a variable and assign it a list with 3 things. Try to print the value at index 3 of the list.

Exercise. What about the other direction? In other words, create a variable and assign it a list with 3 things. Try to print the value at index -1. What about -2? Can you guess what's happening?

`for`-loop

I want you to try the following programs and see if you can figure out how the `for`-loop works:

Try this:

```
for x in [2, 3, 5, 7]:
    print "x =", x
print "done!"
```

Exercise. For the above program, which statement is repeated? The statement

```
print "x =", x
```

or the statement

```
print "done!"
```
?

Exercise. For the above program, the statement which is repeated picks up a different value for x. Where do these values come from?

Try this:

```
for y in [2, 3, 5, 7]:
    print "y + 1 =", y + 1
print "done!"
```

And this:

```
for i in [2, 3, 5, 7]:
    print "hello world"
print "done!"
```

And this:

```
for x in [1, 2, "buckle", "my", "shoes"]:
    print x
print "done!"
```

And one more:

```
for x in [1, 2, "buckle", "my", "shoes"]:
    print x,
print "done!"
```

(Remember what happens when you have a print statement

ending with a comma?)

If you see what's happening skip ahead to the next section. If not try the above examples again before reading the explanation below.

The `for`-loop looks like this:

```
for [variable] in [list]:
     [statement]
```

The words `for` and `in` are reserved words in Python; they are not optional – you **must** include them. The variable `[variable]` will run across the values in `[list]` from left to right. For each value `[variable]` picks up from `[list]`, the `[statement]` will run once.

So something like

```
for x in [1, 2, 3]:
     print x
```

has the same effect as

```
x = 1
print x
x = 2
print x
x = 3
print x
```

Some Exercises to Flex Your Brain

To test your understanding of the `for`-loop try the next few exercises.

Exercise. Complete the following program:

```
for x in [1, 2, 3, 4, 5]:

print "done!"
```

so that when you run the program you get this output:

```
>>>
2
4
6
8
10
done!
>>>
```

Exercise. Complete the following program that prints the squares of 1, 2, 3, 4, 5.

```
for x in [1, 2, 3, 4, 5]:

print "done!"
```

Exercise. Complete the following program that prints `x` when `x` is less than 3.

```
for x in [1, 2, 3, 4, 5]:


print "done!"
```

Exercise (DIY). Using Python find the largest integer whose 5-th power is less than 100000. This requires some thinking. You might want to do this exercise when you read this set of notes the second time.

`for`-loop with block

Now the following should not be too surprising. You can execute a **block** of statements under the for-loop. Make sure you run it:

```
for x in [1, 2, 3, 4, 5]:
    print "x is ...[drumroll]..."
    print x
print "done!"
```

Suppose you have a list and you want to scan across it (to print it, to compute the sum of its terms, etc.) There are two different ways of doing it. You have already seen the first method:

```
xs =  [100, 300, 500, 200, 400]
for x in xs:
    print x
```

Now here's another:

Exercise. Complete the following so that all the values in `xs` are printed.

```
xs =  [100, 300, 500, 200, 400]
for i in                :
    print xs[i]
```

Read and understand the difference between the above programs. In the first program, your `for`-loop variable `x` picks up the values in the list. In the second, the variable `i` in the `for`-loop picks up the index values of the list `xs` so that the `xs[i]` runs across all the values in the list `xs`.

The `range` function

The following exercises are very helpful:

Exercise. Run this program:

```
print [0, 1, 2]
print range(3)
```

What is `range(3)`? Change the 3 to 10 and run the program again. Try a few more numbers. Do you get it? `range` is a function. If `n` is an integer what do you get from `range(n)`?

Exercise. Print the list of all the integers from 0 to 999, one on each line. Of course you can try this

```
print 0
print 1
print 2
print 3
```

at which point you would realize that this is crazy. Is there a better way? [Hint: Use a `for`-loop and the `range` function].

Exercise. In the first example of this section, an integer value was used as an input into the `range` function. Can you use a variable instead? Write a program that prompts the user for an integer and assign the value to variable `n`. Try to print `range(n)`. Does it work? Does it work if you use an expression (example `n * n + 5` instead of `n`) as an input?

Exercise. Execute this:

```
print range(10, 100)
```

What is `range(10, 100)`? Try different values in place of `10` and `100` including negative values. If m and n are integer values what do you get from `range(m,n)`?

Exercise. Execute this:

```
print range(100, 10)
```

What do you get? Why?

Exercise. Write a program that prompts the user for two integer values. If the first integer is less than the second, print all the integers from the first to one less than the second. Otherwise print `"Hey! The second has to be`

```
larger than the first!"
```

Exercise. Run this program

```
print range(10, 20, 2)
```

Now run this

```
print range(10, 20, 3)
```

Get it? If not try a few more values.

Exercise. Print the squares of integers of all the odd integers from 1 to 99 (inclusive).

Exercise. Run this program

```
print range(20, 0, -1)
```

Now run this

```
print range(20, 0, -2)
```

I hope it's clear that the `range` function gives you a list. Now in math the number of inputs for a function is always the same. You see from the above that in Python a function can have many **forms**. In fact there are three forms for `range`:

- `range(a)`
- `range(a, b)`
- `range(a, b, c)`

Exercise. Write a program that prompts the user for a positive integer n and prints n “hello world”s.

Exercise. Johnny was a bad boy. So mom told him to write “I will not put peanut butter into dad's shoes anymore” 1000 times. Johnny decided to get Python to do it for him. Write a program that does just that.

Exercise. Modify your multiplication guessing program so that instead of asking one question, the program asks n questions where n is an input from the user.

Recall the following example:

```
xs =  [100, 300, 500, 200, 400]
for i in [0, 1, 2, 3, 4]:
    print xs[i]
```

You can write this as:

```
xs =  [100, 300, 500, 200, 400]
for i in range(5):
    print xs[i]
```

There's a better way to do it. Note that this method requires knowing `range(5)`, i.e., that there are 5 things in `xs`. Actually you can find out how many things are in a list. Try this:

```
xs = ["foo", "bar", "baz", "boo", 42]
print len(xs)
ys = [2, 3, 5, 7, 11]
z = len(ys)
print z
```

`len` is the **length function**.

With the length function the program above can be rewritten as:

```
xs =  [100, 300, 500, 200, 400]
for i in range(len(xs)):
    print xs[i]
```

Randomizing

This is the goal for this section. You notice that in the bouncing_alien.py program, the alien always begin at the same position (top-left corner) and the speed is the same. We want to now randomize the initial position and the speed of the alien.

Later we will use this to create 50 aliens. Without these randomizations, we will have 50 alien bugs starting at the same position and moving in the same way – we have 50 bugs on top of each other. Not too exciting is it?

First make a copy of bouncing_alien.py, call the new program bouncing_alien_loop.py.

Exercise. Open bouncing_alien_loop.py in IDLE. Look at the code. There is one list variable. Find it. Try to understand what it does.

Recall that `alienrect` is a rectangular region made up of 4 integer values x,y,w,h. (x,y) is the top-left corner of the region in the main surface. w and h are the width and height of the rectangular region.

Now look at these crucial statements in the bouncing_alien_loop.py (I've added a print statement):

```
# Move alien
alienrect = alienrect.move(speed)
print alienrect
```

Change the values in `speed` and try to understand what's happening.

`alienrect` was declared here in bouncing_alien_loop.py:

```
alienrect = alien.get_rect()
print alienrect
```

And this is the first value of `alienrect`.

Exercise. Just below the declaration of speed, declare another variable `initial_pos` (i.e. initial position) that looks like `speed`. But give it the value `[100, 100]`. Now we move the initial value of `alienrect` by `initial_pos`.

```
alienrect = alien.get_rect()
alienrect = alienrect.move(initial_pos)
print alienrect
```

Now run the program. Observe the print output. Do you understand what's happening?

Exercise. Instead of giving `initial_pos` the value `[100, 100]` give it a list of two “reasonable” random values. (What's reasonable???) Run the program. Make sure the alien begins at random initial positions.

Exercise. Now do the same with `speed`: give `speed` a list of two reasonable random values. Run your program and check your work.

If you manage to finish your work, you will get the alien to start at a randomly chosen initial position and move at a randomly chosen speed.

(For the Physics buffs, technically “speed” should be called “velocity”.)

Computing Sum and Products in a Loop

Try this program.

```
sum = 0
for i in range(10):
    sum = sum + i
    print "i:", i, "sum:", sum
print "final sum: ", sum
```

Make sure you trace the execution of this program step by step by hand using the model of Python seen on the set of notes on Branching. Make sure your execution of the model gives the same results as the actual Python.

Now do the same for this example:

```
product = 1
for i in range(1, 10):
    product = product * i
    print "i:", i, "product:", product
print "final product: ", product
```

The following program will allow you to add numbers based on user input:

```
n = input("How many terms to add? ")
sum = 0
for i in range(n):
    term = input("Enter term: ")
    product =  + term
    print "term:", term, "sum:", sum
print "final sum: ", sum
```

Exercise. Write a program that prompts the user for m and n and prints the sum of all the integers from m to n (inclusive).

Exercise. Run this program.

```
product = 0
for i in range(1, 10):
    product = product * i
print product
```

Why?

`time` and Floating Point Numbers

Let's improve on the multiplication program. We now want to tell the user how much time he/she took to do the problems. Try this program:

```
import time
start = time.clock()
for i in range(1000):
    print "blah"
end = time.clock()
print "that took", start - end, "seconds"
```

AHA! So Python can do decimals too. (Not too surprising right?)

`clock` is a function in the `time` module that returns the number of seconds since the `time` module was imported. So to measure the amount of time needed to execute a block of statements (including a single statement), you just need to record the time before and after the block. When you take the difference, you get the time taken to execute the block.

Exercise. How much time does it take Python on your PC to print 100 `"hello world"`s.

Exercise. How much time does it take Python to add 1000 pairs of integers? Do this:

- Create a list `x` of 1000 random integer from 0 to 1000000
- Create another list `y` likewise.
- In a `for`-loop assign a dummy variable `z` the sum of `x[i]` and `y[i]` as `i` runs across all the index values of `x` and `y`.
- Finally time the `for`-loop.

(Note that the time is slightly smaller since the time the above program measures includes the assignment of the product to `z` and it also includes the time for `i` to run across the index values.

Exercise. Now repeat the above program for multiplication. Which is more expensive to the computer (i.e. takes more

time), addition or multiplication?

What about numbers with decimals? The technical term for such numbers are **floating point numbers** or just **floats**. I don't have much to tell you. Almost everything you know about integers apply. For instance you can add using +, etc. You can compare using <, etc. The only thing you have to careful about is that **they are not always exact; they are approximations, very good approximations.**

Exercise. In the Python shell, print 0.9. Is it 0.9?

Now when you add integers, you get an integer. If you add two floating numbers you get a floating numbers. But **when you add an integer and a floating point number, you get floating point number.**

```
print 1 + 2
print 1.0 + 2.0
print 1 + 2.0
print 1.0 + 2
```

That's the same for `-` , `*`, etc. In general when there is an operator requiring two numbers where one is an integer and the other is a float, the integer is changed into a float before the operator is used. The change from an integer to a float like this is called **type promotion**.

Two other things about floats before we move on. First you can convert an integer value to a floating point number easily:

```
x = 42
y = float(x)
print x, y
y = y + 0.1
z = int(y)
print y, z
```

The important thing to remember is that `int(y)` will give an

integer value from the value `y` with the decimal part chopped off. This “chopping off” business is called truncation.

The second thing is that you can prompt the user for a floating point number using the same `input` function:

```
x = input("gimme an integer: ")
y = input("gimme a float: ")
print x, y
```

Run this program entering `42` for `x` and `4.2` for `y`.

Exercise. Write a program that prompts the user for two integers and prints the average. Run the program and enter two integers such as 1 and 2. You should get 1.5.

Exercise. Modify your multiplication program so that it displays the total amount of time to answer all the questions.

Exercise. Modify your multiplication program so that it displays the total and average amount of time to answer all the questions.

Assignment and Lists

Lists and integers are values. You can give them to variables:

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

There is however a **big difference** between them when it comes to assignment (i.e. the = sign).

First try this:

```
x = 42
y = x
x = -3
print x, y
```

Now try this:

```
x = [1, 2, 3]
y = x
x[0] = 42
print x, y
```

So what do you think is the point of the above two programs?

If you don't see it, here's the point. Suppose you look at this assignment:

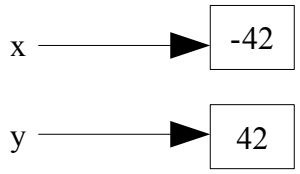
```
y = x
```

If x is an integer, x and y have their own “boxes” for values so that changing one does not affect the other. However if `x` is a list, then after the assignment, `x` and `y` **share the same box**.

So the picture in Python's mind after this chunk of code

```
x = 42
y = x
x = -42
print x, y
```

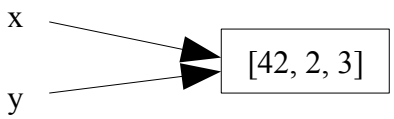
is:



But for

```
x = [1, 2, 3]
y = x
x[0] = 42
print x, y
```

the picture is



In other words for the above code, `x[0]` is the **same as** `y[0]`, `x[1]` is the same as `y[1]`, `x[2]` is the same as `y[2]`.

Very frequently a program requires you to copy the value from a variable say x to another say y. You then want to work on y such as changing its value but you don't want the value of x to change. For an integer variable x, copy can be achieved by assignment, i.e. =. But not for lists. I'll show you how to do that later.

Right now, just remember this **important** fact: = has two different “interpretations”. It can mean “**copy** the value from one variable to another” or it can mean “make two variables **share** the same value space”.

Slices and Copying Lists (DIY)

If `x` is a list, you have already seen the notation `x[i]`. The `[]` is called the **bracket operator**. There are two other operators that look like it: the `[:]` and the `[::]`.

Run this program:

```
x = [10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
print x[2:7]
print x[:7]
print x[2:]
y = x[2:]
print y
print x[:]

print x[1:9:2]
print x[9:1:-1]
print x[9:1:-2]
print x[::-1]
```

By modifying the values in `[:]` or `[::]` and observing the outputs, see if you can figure out what they do.

Now you notice in the above program that printing out `x` and printing out `x[:]` give you the same output. So it seems like `x` and `x[:]` are the same. But there's a difference. Try this:

```
x = [1, 2, 3, 4, 5]
y = x
z = x[:]
print x, y, z

y[0] = 42
print x, y, z

z[1] = 12345
print x, y, z
```

You already know that for lists x and y, the statement `x = y` does not mean "**copy** the value from `y` to `x`"; it means "make `x` and `y` **share** the same value space. This means that changing the value of `y`, changes the value of `x`. But what happens when you change `z`? Does it change `x`?

NO!

So if you really want to copy the value from a list variable to another (and not make them share the same value space), you should use the above trick.

Lists: + and `append` (DIY)

Here's another operator for lists. Run this program:

```
x = [1, 2]
y = ["buckle", "my", "shoe"]
z = x + y
print x, y, z

z = z + [3, 4]
print z
```

This example tells you that you can perform the + operation on lists. The + operation joins or **concatenates** the two lists.

Another thing we frequently do is add a thing into a list. Try this:

```
names = ["john", "mary"]
name = "harry"
names.append(name)
print names
names.append("susan")
print names
```

(AHA! Now you know that lists are objects, right? Why?)

Make sure you see the difference between + and `append`:

```
x = [1, 2]
y = [3, 4, 5]

z = x + y
print z

x.append(y)
print x
```

Tuples

This is easy. I'll just say that tuples are like lists in many ways. Try this program:

```
a = (1, 2, 3)
print a
print a[0]
print a[1]
print a[2]
```

Exercise. Write a for-loop with a variable i that runs across not a list but a tuple of 1, 2, 3. Does it work?

The only difference is that you cannot change the value of a tuple. In other words you cannot change the value of an entry in a tuple and you cannot append to a tuple.

Exercise. What must you verify right away?

So as long as you're not changing the value of a tuple, it's really nothing more than a list.

Exercise. Open up your first bouncing_alien.py. Look for all the tuples.

50 Aliens

This is it. It's time to catch 50 alien bugs.

Let's look at bouncing_alien_loop.py again.

We want to have 50 aliens. (Don't worry about collision between aliens.)

Let's analyze the problem. Look at the program. There are three variables: `alien`, `alienrect`, and `speed`. Alien is the image loaded from an image file. `alienrect` is the region where the alien image is to be blitted. And `speed` is used to change/move `alienrect`.

To the viewer of the program, he/she sees 50 aliens moving around. To simulate this what do we need?

Well, we need to blit the image to 50 rectangular regions.

Note what I just said: We need 50 rectangular regions. But we only need one image variable.

The 50 rectangular regions must change/move. Of course each rectangular region must move in its own way. Therefore, each rectangular region has its own speed. Therefore we need 50 speed variables.

We are about to modify bouncing_alien_loop.py. Here's my advice: make a backup copy of the program in case things go wrong.

Exercise. Here's a practice problem on creating a list of 10 random values. Here are some steps that might help:

- Declare a variable, call it `xs`, initializing it to the empty list.
- In a for-loop that executes it's statement 10 times, append `xs` with a random number between 0 and 5 (inclusive).
- Print `xs`.

You should see a list of 10 numbers from 0 to 5.

Now complete bouncing_alien_loop.py! Have fun!

Numeric Crunching Exercises (DIY)

The next few exercises (projects?) are numeric crunching programs. You should try them. But be very patient. If you're lost just ask for help. Actually for the first part, the solution is given. In fact the most important thing about the first part is the steps I take to solve the problem. So please read it carefully and thoughtfully.

The exercises are all related to computing prime numbers which are crucial for cryptography. Which means that whenever you buy something online you're unknowingly using prime numbers (well ... I hope you bought your goodies at a *secure* web site ... did you check???) For online games, in order to make sure no one is cheating, cryptography is also used. And of course to make sure that unauthorized people cannot hack into a game server easily, cryptography is again used.

Prime Numbers (DIY)

An integer is always divisible by 1 and itself.

A **prime number** is a positive integer which is divisible only by 1 and itself. 1 is not considered a prime (remember that).

For instance 6 is not a prime number because 6 is divisible by 2. On the other hand 7 is prime.

Now how do I know 7 is a prime? Because 2, 3, 4, 5, and 6 cannot divide 7.

To say that 2, 3, 4, 5, 6 cannot divide 7 is the same as saying that the remainders after 7 is divided by 2, 3, 4, 5, 6 are all non-zero. Correct?

Recall that in Python the remainder after 7 is divided by 2 is 7 % 2. (Check your notes!)

So to show that 7 is a prime, I just need to use Python to check

```
print 7 % 2
print 7 % 3
print 7 % 4
print 7 % 5
print 7 % 6
```

The values are all non-zero, so 7 is a prime.

The following example (sequence of exercises) is important because this is the first non-trivial example. Although primes are important, the more important thing about this section is the problem-solving aspect.

The final program prompts the user for a positive integer and prints “prime” if the integer is a prime, and “not prime” otherwise.

You are strongly encouraged to give it a shot on your own. If you can write your own program ... great work! If not, then do not be discouraged. Just read on and follow the hints ...

Exercise. Write a program that prompts the user for a positive integer; give variable n that value. Write a `for`-loop that prints n % i where i runs from 2 to n – 1 (inclusive).

Exercise. Modify the above program by changing the print statement so that if n % d is not zero you print “not divisible” and if n % d is zero you print “AHA! Not a prime!”.

Exercise. (This is a long one). Modify the above program. Instead of printing out the result of each test in the `for`-loop, we'll remember the result. Now look at the print out from the previous exercise. Think about what you want the program to do. In fact think about how **you** would do it by hand. To be good programmer, you have to be able to execute the steps on a specific example and then translate each step to statements.

Let's use a previous example where we printed out the remainders. Let's include some comments based on the output:

```
print 7 % 3 <--- Possibly a prime. Go on.
print 7 % 2 <--- Possibly a prime. Go on.
print 7 % 4 <--- Possibly a prime. Go on.
print 7 % 5 <--- Possibly a prime. Go on.
print 7 % 6 <--- Possibly a prime. Go on.
                 No more cases. A prime.
```

What about 9?

```
print 9 % 2 <--- Possibly a prime. Go on.
print 9 % 3 <--- AHA! Not a prime.
print 9 % 4
print 9 % 5
print 9 % 6
print 9 % 7
print 9 % 8
```

This hints that for each execute of the statements of the for-loop, there are two possible outcomes: Either “Possibly a prime. Go on” or “AHA! Not a prime”.

Suppose we have a variable to remember the outcome. Let's call the variable x. For the case “Possible a prime. Go on”, we set x to 0. For the other case we set x to 1. Let's go back to the examples and change the comments to something

involving variable x.

```
print 7 % 3 <--- Set x = 0.
print 7 % 2 <--- Set x = 0.
print 7 % 4 <--- Set x = 0.
print 7 % 5 <--- Set x = 0.
print 7 % 6 <--- Set x = 0.
                 No more cases. If x is 0, 7
                 is a prime.
```

You see the general pattern is

```
if n % i is not 0, set x to 0
```

```
print 9 % 2 <--- Set x = 0.
Print 9 % 3 <--- Set x = 1.
print 9 % 4
print 9 % 5
print 9 % 6
print 9 % 7
print 9 % 8
```

Since 9 % 3 is 0, and x is set to 1, we know that 9 is not a prime. The general statement for the second comment seems to be:

```
if n % i is 0, set x to 1
```

When we do the computation by hand, we know that we can stop now. But for Python, the `for`-loop has to go on!

Anyway the `for`-loop looks like this: Given integer n,

```
for i in range(2, n):
    if n % i is not 0, set x to 0
    otherwise set x to 1.
```

But here you have a problem. If the `for`-loop continues to run we have this:

```
print 9 % 2 <--- Set x = 0.
print 9 % 3 <--- Set x = 1.
print 9 % 4 <--- Set x = 0.
print 9 % 5
print 9 % 6
print 9 % 7
print 9 % 8
```

In other words we lose the x = 1. Now think about it. Think about what you would do in real life. If `n % i` is zero, you set x to 1 and you don't really need to do anything more. In other words don't change the value of x anymore. Right? See that!

Say that again, more precisely: When we've found an i that divides n, we set x to 1 and don't change it anymore.

Not changing x “anymore” mean the next time you run the statement in the for-loop, don't do anymore checking or changing x anymore. That means only do the check when x is 0:

```
for i in range(2, n):
    if x is 0:
        if n % i is not 0, set x to 0
        otherwise set x to 1.
```

Of course this means that x has to have a value to begin with. What does it mean when x is 0? It means we have to “go on”. So the initial value of x should be 0:

```
x = 0
for i in range(2, n):
    if x is 0:
        if n % i is not 0, set x to 0
        otherwise set x to 1.
```

In proper Python:

```
x = 0
for i in range(2, n):
    if x == 0:
        if n % i != 0:
            x = 0
        else:
            x = 1.
```

Do you notice something fishy about the program? x starts off as 0. x gets set to 0 again and again (look at the examples above) until it gets set to 1. At which point x is not changed anymore. Is something redundant here? Why do we need to set x to 0 again and again?? We don't need to do that. So we only need to worry about the case where we set

`x` to 1. We need to do this when `n % i` is 0 (the `else` part of the `if-else` statement). Let's do that instead. Make sure you read the following and understand it:

```
x = 0
for i in range(2, n):
    if x == 0:
        if n % i == 0:
            x = 1
```

There's one final improvement: Remember choosing good variable names? What does x represent? x is 0 when n, so far, seems like a prime. In other words, no divisor has been found, yet. Notice also that x takes two values? There is a type of variable that takes two values: Boolean variables. So I'm going to use the variable `isprime` to tell me that “the number is possibly a prime because a divisor is not yet found”.

```
isprime = True
for i in range(2, n):
    if isprime:
        if n % i == 0:
            isprime = False
```

Now you should see that

```
if [condition1]:
    if [condition2]:
        [block]
```

if the same as

```
if [condition1] and [condition2]:
    [block]
```

Pause and think about why they are the same. See it?

And so we have

```
n = input("Enter positive integer: ")

isprime = True
for i in range(2, n):
    if isprime and n % i == 0:
```

```
        isprime = False

if isprime:
    print "Not a prime"
else:
    print "Prime!"
```

You're strongly encouraged to read the above again. The problem solving process is more important than the program. When you think you've understood everything, close your notes and write your own version of the program. If you can't get yours to work, check with my solution, especially the reasoning behind the steps that you miss. Ask questions!

Breaking out of a loop (DIY)

Now look at our program:

```
n = input("Enter positive integer: ")

isprime = True
for i in range(2, n):
    if isprime and n % i == 0:
        isprime == False

if isprime:
    print "Not a prime"
else:
    print "Prime!"
```

You know that when `isprime` is `False`, there is really no point in continuing the `for`-loop. Actually Python does know how to stop a `for`-loop early. You use the `break` statement. While in a loop, if Python executes the `break` statement, it immediately goes to the statement just after the loop. Try this:

```
for i in [1, 2, 3, 4, 5]:
    print i
    if i == 4:
        break
print "done"
```

Now try this. I've included a print statement to show the number of values `i` picks up from the list. Enter 25 for n.

```
n = input("Enter positive integer: ")

isprime = True
for i in range(2, n):
    print "i =", i
    if isprime and n % i == 0:
        isprime = False
        break

if isprime:
    print "Not a prime"
else:
    print "Prime!"
```

An Optimization Step (DIY)

Actually for our prime checking program:

```
n = input("Enter positive integer: ")

isprime = True
for i in range(2, n):
    print "i =", i
    if isprime and n % i == 0:
        isprime = False
        break

if isprime:
    print "Not a prime"
else:
    print "Prime!"
```

you do not need to check `n % i` for `i` from 2 to n-1. You can actually stop much earlier. You only need to run i from 2 to the square root of n. So if you want to check if a number close to 10000 is a prime you need to check remainders using i from 2 to roughly 100 instead of 2 to roughly 10000. That's a lot of savings.

The square root function in Python is `sqrt`. `sqrt` is in the `math` module.

Exercise. Using Python, compute the square root of 4. Note that you get a floating point number.

Exercise. Try to print `range(2, 5.1)`. In other words can you use floating point numbers in the `range` function?

Exercise. Modify the prime number program by checking for divisors from 2 to the square root of n (instead of up to n-1).

Computing a List of Primes (DIY)

Exercise. First create a variable `primes` and set it to an empty list. Next use the above program, and instead of the print statement, if `n` is a prime, append `n` to `primes`. Print `primes`. Now instead of prompting the user for `n`, "wrap" the code inside a for-loop that runs n from 2 to 100. Your output is the list of primes from 2 to 100.

```
# create variable primes here

for n in range(2,101):

    # check if n is a prime







    # put n into primes if n is a prime




print primes
```

Exercise. Time the computation of all the primes from 2 to 1000. What about up to 10000? What about up to 100000?

Exercise. Remember that we made an optimization so that prime checking goes from 2 to the square root of n instead of n-1? Change the program so that it checks up to n-1. Time the computation of all the primes from 2 to 1000 and compare with the previous exercise. How much time do you save? What about up to 10000?

This is my answer:

- For primes from 2 to 10000, without the optimization it took 3.91 seconds. With the optimization, it took 0.393 seconds. 10 times difference.
- Now for 2 to 100000 primes, without optimization, it took 344 seconds and with optimization it took 3.34 seconds.

Do you see the gap widening?

Now consider this: For cryptography the primes used are extremely huge. There are usually more than a hundred digits in length.

Exercise. (Challenging!) Now improve the program. Suppose you already have the primes from 2 to n and you want to check if n+1 is a prime. Do you really need to check if n+1 is divisible by all the integer from 2 to n? For instance suppose you want to check if 21 is a prime and you already have the primes up to 20, i.e., `[2, 3, 5, 7, 11, 13, 17, 19]`. Wouldn't it be enough to try to check if the numbers in `primes` divide 21? Yes? When you're done with your program, time the computation of primes from 2 to 10000 and then up to 100000.