

Branching

Objectives

- Write if statements
- Write if-else statements
- Write if-elif and if-elif-else statements
- Import modules
- Use names in a module

There are really two main sections in this set of notes. First we will look at how Python **make decisions**.

Without the ability to decide, Python won't be smart. You'll see that decision making statements are already found in the `bouncing_alien.py`.

The second main section is an introduction to pygame, a Python game development tool. We will also begin explaining more about `bouncing_alien.py`.

Along the way we will look at **generating random numbers**. Generating random numbers is important because also all games have some randomness in them. (Example: Random placement of objects in a role-playing game, random mazes for dungeon type games, randomness in an AI bot while making decisions, etc.)

Two other things we will look at are **modules** and **functions**. We will need them to understand random number generation and `bouncing_alien.py`.

To make sure you understand the new ideas, you will

- Write a simple multiplication game
- Modify the `bouncing_alien.py` program

Making Decisions

So far you know that Python can do arithmetic, print integer values, print string values, accept integer values from the keyboards, create and update variables.

You should expect more!

For instance you would expect programs to make decisions. Now wouldn't it be nice if Python can do this:

```
if alien is killed:
    score = score + 10
```

or

```
if my ship's energy is 0:
    end the game
```

Note that for the first case, if the alien is not killed, you do *not* want to execute the statement `score = score + 10`. Otherwise everyone would keep getting the same score! What kind of dumb game is that?

In fact Python *does* understand the `if` statement.

Try this program:

```
x = input("Enter an integer: ")
if x > 0:
    print "greater than 0"
print "tada!"
```

IDLE will provide an indentation for you.

IDLE will provide another indentation. Use the backspace to remove it.

You read `x > 0` as “x is greater than 0”.

The spacing for the statement under the `if` statement is called an **indentation**:

```
if x > 0:
    print "greater than 0"
```

Indentation

It's **not** optional if you want that statement to execute only when `x > 0` is true.

Note that the statement `print "tada!"` is **not** controlled by the `if`.

Run the program several times, entering different values for `x` each time. Make sure you try positive and negative values for `x`.

Exercise. What if you remove the colon (i.e., `:`) after `x > 0`. Does the program still work?

Exercise. What if you wrote this instead:

```
if x > 0:
    print "greater than 0"
```

(i.e. have a bigger indentation) by putting your cursor at the beginning of the line and pressing the tab key once.

Now try this:

```
x = input("Enter an integer: ")
if x < 0:
    print "boo"
print "tada!"
```

`x < 0` is read “`x` is less than 0”.

What if you want to check if `x` has value 0? Try this

```
x = input("Enter an integer: ")
if x == 0:
    print "bar"
print "tada!"
```

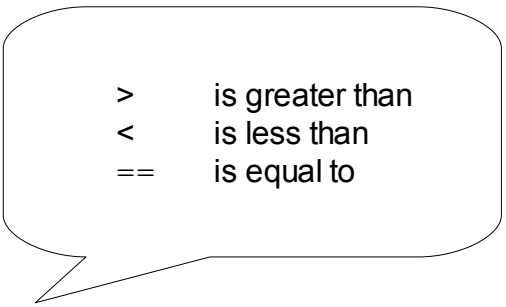
Run this several times entering 0 and any non-zero value.

`==` is read “**is equal to**”.

WATCH OUT! There are **two** `=` signs, i.e., `==`. A single `=` sign is for assignment. Remember?

Exercise. If you put a space between the two equal signs does the program still work?

```
x = input("Enter an integer: ")
if x = 0:
    print "boo"
print "tada!"
```



<code>></code>	is greater than
<code><</code>	is less than
<code>==</code>	is equal to

Exercise. What about this?

```
x = input("Enter an integer: ")
if x = 0:
    print "boo"
print "tada!"
```

(ASIDE: In some programming languages this is actually allowed, causing many hard-to-find programming errors.)

Summary

As a quick summary the format of the `if` statement looks like this:

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

where `[condition]` looks like this:

```
[variable] [op] [value]
```

where `[variable]` is a variable and `[value]` is an integer value. So far the `[op]` is `<`, `>`, or `==`. There are others:

<code>==</code>	is equal to
<code>!=</code>	is not equal to
<code><</code>	is less than
<code><=</code>	is less than or equal to
<code>></code>	is greater than
<code>>=</code>	is greater than or equal to

Exercise. Write a program that prompts the user for how much he/she has in his/her savings account. If the amount less-than-or-equal-to zero, print “you're broke!”. The “less-than-or-equal-to” in Python is written `<=`. Here's a skeleton that might help:

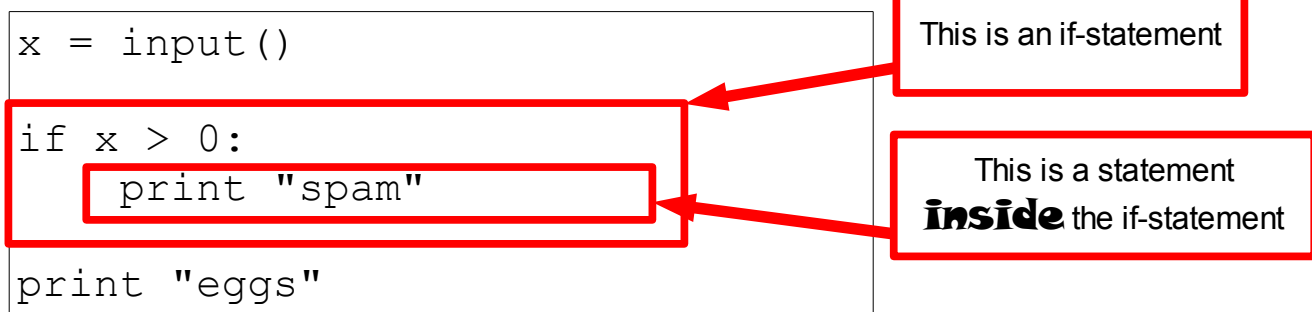
```
x = input(_____  
if _____:  
    print _____
```

Mental Picture: Flow of Execution for `if`

The `if` statement is difficult for a beginning programmer because it alters the flow of execution of a program.

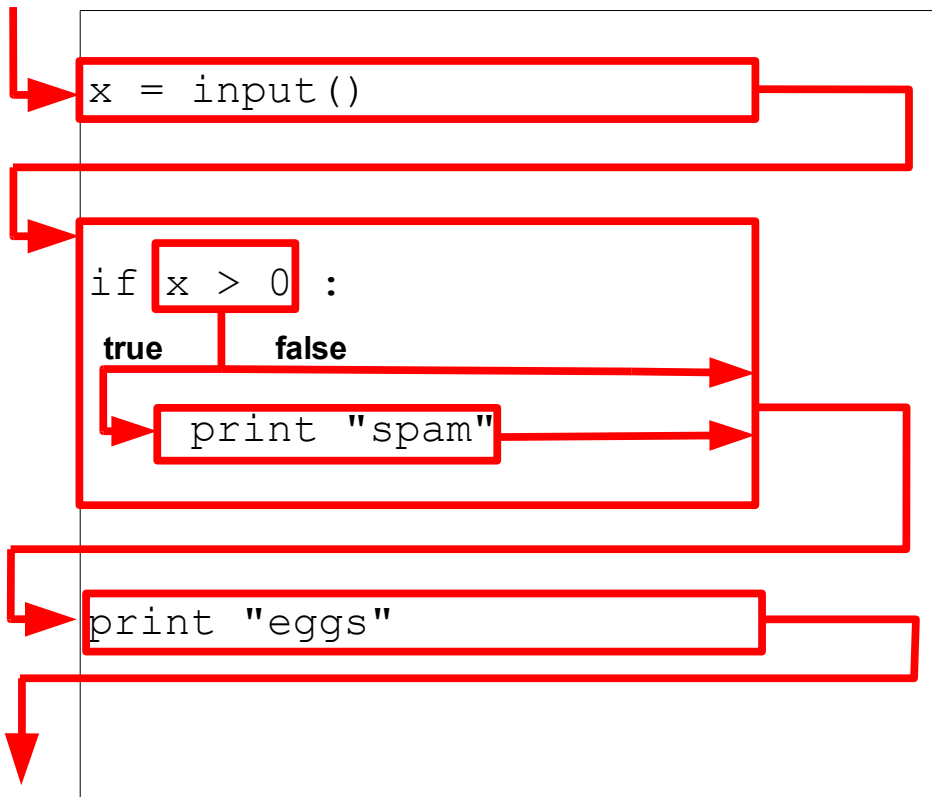
Before this set of notes, Python reads and execute one statement at a time from top to bottom; it executes every statement.

The `if` statement is different because it has a statement **inside** the `if` statement.



The statement in the `if` statement is executed only when the `x > 0` is true. If the condition is not true, the statement in the `if` statement is not executed.

In the following diagram, you can follow the arrows to get a sense of the flow of execution:



Multiplication Game

Exercise. Write a program that tests if a person can do two digit multiplications. Specifically the program

- Prints a string prompting the user for the product of 97 and 94.
- Prompts the user for an integer. Give the value to the variable `guess`.
- If value of `guess` is the same 9118, prints `Correct!`

I'm going to run mine, first with a wrong answer:

```
>>>
What is the product of 97 and 94 ?
Answer: 1
>>>
```

The second time I run it, I enter the correct answer:

```
>>>
What is the product of 97 and 94 ?
Answer: 9118
Correct!
>>> |
```

This is kind of a dumb program because it asks the same question again and again.

Random Integers

Generating a random integer occurs commonly in computer programs. For instance if you want to write a strategy role-playing game, you might have different rooms in a mansion and you want to put different things in different rooms. Suppose you have a magic potion in a bottle. Now suppose the rooms are numbered from 1 to 1000. You don't want your magic potion bottle to be in room 5 whenever you start the game. You want each game to be different.

At a more complex level you might want to randomly make doors between two rooms. This creates a random maze.

So you'd better learn how to generate random things.

Try this:

```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

Run this program 10 times. Notice that the room number is different each time you run the program.

We'll have a short break from random numbers to talk about modules and functions ...

A Module Quickie

Now I need to explain the statement `import random`.

First I want you to create the following programs `test.py` and `usetest.py` on your Desktop:

```
# test.py
foo = 42
print "in test ...", foo
```

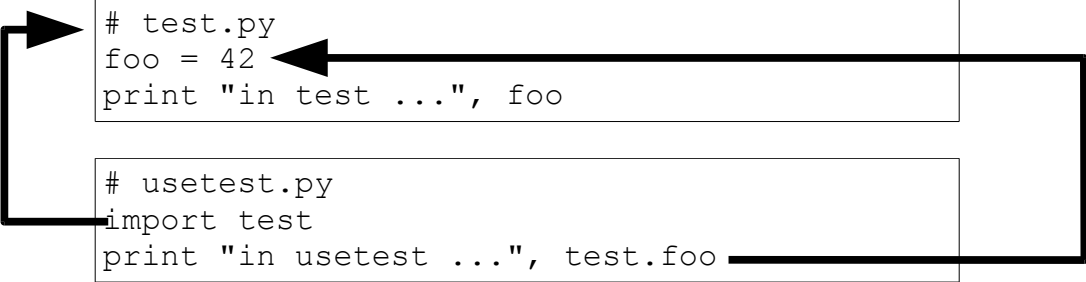
```
# usetest.py
import test
print "in usetest ...", test.foo
```

Next run `test.py` and then `usetest.py`. Look at the output and see if you can figure out what's happening ...

Let me explain what's happening.

The statement `import test` makes the **names in `test.py` available to `usetest.py`**, this includes the variable `foo`. After importing `test.py`, the **`foo` variable from `test.py` is referred to as `test.foo` in `usetest.py` and not `foo`.**

The `test` in `import test` is called a **module**. For the time being you can think of a module as a file, just like the module `test` is like the file `test.py`. (The two are actually different.)



```
# test.py
foo = 42
print "in test ...", foo
```

```
# usetest.py
import test
print "in usetest ...", test.foo
```

Exercise. This is a **common gotcha** for Python beginners. In `usetest.py`, change `import test` to `import test.py` like this:

```
# usetest.py
import test.py
print "in usetest ...", test.foo
```

Run the program. Does it work?

Exercise. In `usetest.py`, remove `import test` like this

```
# usetest.py
print "in usetest ...", test.foo
```

and re-run your program. Does it work?

Exercise. This is an **important exercise**.

- In `usetest.py`, add a statement after the `import` statement to create a variable called `foo` and give it the value 41.
- After the statement to print `test.foo`, add a similar print statement to print `foo` (not `test.foo`).

Note that there are now two `foo` variables: one in `test.py` and one in `usetest.py`. Run the program. Do the two `foos` have the same value? Does the program work?

The above is an important exercise and in fact is the reason for the module concept. In large-scale software, you (and your colleagues) usually write a program in many files – sometimes up to thousands. The module concept allows you to **differentiate between variables with the same name in different files**.

Exercise. Create three Python program files. In the first `A.py`, create a variable `a` and give it the value 100 like this:

```
# This is file A.py
a = 100
```

In the second `B.py`, create a variable `b` and give it the value 42. In the third program `C.py`, print the sum of `a` in `A.py` and `b` in `B.py`.

ASIDE FOR C++ and JAVA PEOPLE. The **C++ namespace** and the **Java module** concepts are both similar to the Python module. However the Python module is a lot more flexible and powerful than either.

Exercise. This is a “duh” exercise. Look at a previous program:

```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

What is the module imported?

Exercise. Look at the program `bouncing_alien.py`. What are the two modules imported in the program?

`pygame` is technically speaking not a module – it's a **package**. But the concept is very close to that of the module. A package is a bunch of modules.

This is only a quick introduction to modules. We'll come back to modules and packages again.

A Function Quickie

So what is a function? A **function** in Python is like a function in math but it's more. There is a complete set of notes for functions. Right now I just want to give you enough understanding to move on. Now in math, a function like

$$f(x) = 2x$$

means that

$$f(5) = 2(5) = 10$$

In other words a function in math is something that has an input and an output. For $f(x)$ above, when you put in 5, $f(5)$ gives you 10.



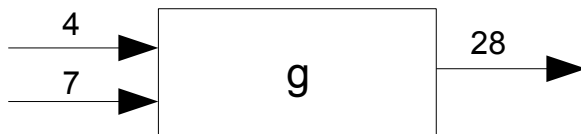
Likewise for the function

$$g(x,y) = xy$$

when you put 4 for x and 7 for y you get

$$g(4,7) = 4(7) = 28$$

Again the 4 and 7 are the inputs and 28 is the output.



Now, back to Python.

Look at a previous program:

```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

Recall that the `import random` makes all the stuff in `random.py` available to the current program. That includes variables **and functions** (and more!). You can tell that there are two things from `random.py`:

- `seed`
- `randrange`

The function **`randrange(x, y)`** has two inputs and the output is a random number between `x` and `y - 1` (inclusive). So the `randrange(x, y)` is very similar to functions from `math`: it has input(s) and an output.



The function `seed()` is also a Python function but it is **very different** from a math function. As you can see it **does not require an input** ... not only that ... `seed()` **does not have an output**. I will only tell you that it makes the random generator in `random.py` "as random as possible".

Exercise. Write a program to simulate the rolling of a die. In other words write a program that will print a random number from 1 to 6.

Right now I only want you to know how to **use functions**. Later I'll show you how to **write your own functions**.

Now back to random numbers ...

Random Numbers (again)

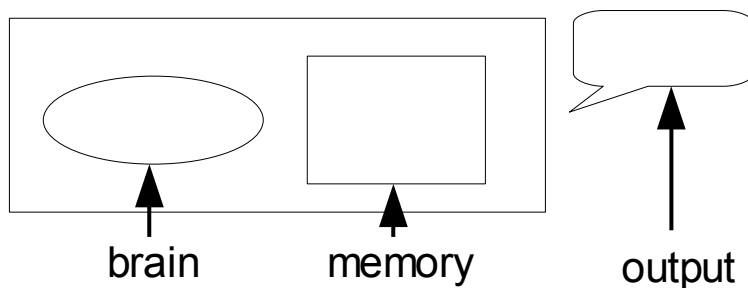
So now you know what's happening in this program:

```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

- `import random` makes everything in `random.py` available to this program.
- `random.seed()` makes the random generator as random as possible.
- `random.randrange(1, 1001)` gives a random number between 1 and 1000.

Now **`random.randrange(1, 1001)` gives** you a random integer value between 1 and 1000 (including 1 and 1000). Now what do I mean by **“gives”**? How does Python **receive** and **use** it? Recall that we have a model of Python:

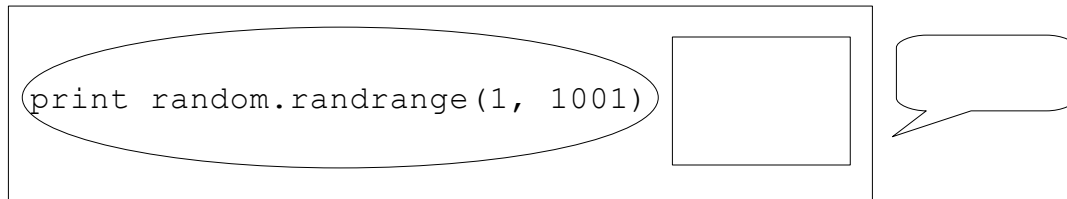


For our model, **in the brain** of Python (but **not in the program file**), `random.randrange(1, 1001)` is **replaced** by the output of the function `random.randrange`. So while executing the 4th statement of

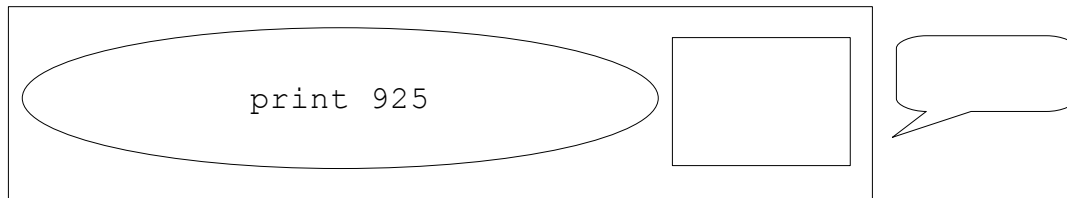
```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

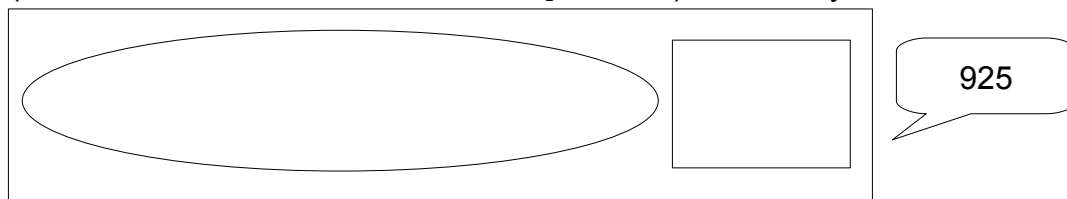
the model starts off as:



and becomes:



(if the random number from `randrange` is 925), and finally:



Again it's important to remember that the change occurs only in Python's brain. At the end of execution the program, the 4th statement of the program is not changed; the program is still

```
import random
random.seed()

print "the potion is in room"
print random.randrange(1, 1001)
```

Exercise. Experiment with the above program by changing the values of `1` and `1001`. Try to figure out how to produce random numbers between (and including) `5` and `10`.

Exercise. Modify the above program so that instead of printing out the random value from

`random.randrange(1, 1001)`, it creates a **variable** for the room of the magic potion and assign it a random number between `1` and `1000` (inclusive). Please use a variable name that's descriptive!!! Print the value of that variable.

Expanding the condition in the `if` Statement

Now we want to modify the multiplication game so that each time you run the program, the problem given is random. But there's a problem ...

Currently our multiplication has only one problem. So there is only one answer which we know is 9118. So we can do something like:

```
if guess == 9118:  
    [statement]
```

More generally, right now the `if` statement looks like this:

```
if [variable] [op] [value]:  
    [statement]
```

Since we want the game to give a different question each time, the answer is different; it need not be 9118. That's the problem.

I'll give a couple of examples as a hint for the next part of our multiplication game.

Exercise. Does the following program work? What is the difference between this form of the `if` statement and the previous?

```
x = 0  
y = 1  
if x < y:  
    print "no kidding"
```

Run the program with different values for `x` and `y`.

Exercise. What about this? What is the difference between this `if` statement and the previous?

```
x = 0  
y = 1  
z = 2  
if x * z + y < y + z:  
    print "no kidding"
```

Again run the program with different values for `x`, `y` and `z`.

Exercise. Help Dr. BadAdviz implement his theory. He believes that a couple is compatible if the product of the man's height (in feet) with the woman's weight (in lbs) is greater than the sum of the number of toes the man has and the woman's waist (in inches). Your program should prompt the user for the man's height (in feet) and number of toes and the woman's weight (in lbs) and waist (in inches). Please use descriptive variable names!!! When the man and woman are compatible, print `Compatible!` Here's a skeleton:

```
man_height = input("Man's height: ")

if _____:
    print _____
```

Summary

The `if` statement looks like this:

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

where `[condition]` looks like this:

```
[expression1] [op] [expression2]
```

where `[expression1]` and `[expression2]` are expressions that can be evaluated into integer values and `[op]` is one of the following:

<code>==</code>	is equal to
<code>!=</code>	is not equal to
<code><</code>	is less than
<code><=</code>	is less than or equal to
<code>></code>	is greater than
<code>>=</code>	is greater than or equal to

Multiplication Game: Random Question

Exercise. Modify your multiplication game as follows:

- Create a variable `x` and give it a random integer value between 90 and 100.
- Create a variable `y` and give it a random integer value between 90 and 100.
- Print a string prompting the user for the product of `x` and `y`.
- Prompt the user for an integer. Give the value to the variable `guess`.
- If `guess` is the same as the product of `x` and `y`, print `Correct!`

I'm going to run my program. For the first run I enter a wrong answer:

```
>>>
What is the product of 97 and 94 ?
Answer: 1
>>>
```

and for the second execution I enter the correct answer:

```
>>>
What is the product of 91 and 99 ?
Answer: 9009
Correct!
>>>
```

Hint: For the input, you know how to do this:

```
guess = input("What is the product?")
```

But for the above question, the problem is that the string changes. For instance in the above example first execution, the string printed when prompting the user for an answer is:

```
"What is the product of 97 and 94?"
```

while for the second execution, the string printed is

```
"What is the product of 91 and 99?"
```

Here's how you do it. Instead of doing this:

```
guess = input("What is the product?")
```

do this

```
print "What is the product?"  
guess = input()
```

And now change the print statement so that it works.
Remember that you can print several things in a print
statement. For instance you can do this:

```
print 1, 2, "buckle my shoe"
```

Blocks of Statements

What if you want to program this:

```
if alien is killed:
    draw explosion on top of alien and
    score = score + 10
```

In other words you want Python to do **two** things when a condition is true. Of course you can do this:

```
if alien is killed:
    draw explosion on top of alien
if alien is killed:
    score = score + 10
```

There are reasons why you do **not** want to do this. It is not efficient and more importantly it tells the readers that you're not programming like a pro!

If you want several statements to execute when a single condition is true, you **bunch** them together like this:

```
x = input()
if x > 0:
    print "spam"
    print "eggs"
print "tada!"
```

So **all** the statements indented after the `if` will execute if the condition `x > 0` is true. These statements form a

block. Note that the **indentations** of **statements** in a **block** must be the **same**.

```
x = input()
if x > 0:
    print "spam"
    print "eggs"
print "tada!"
```

A block of two
statements in the
`if`-statement

Indentation

Exercise. This is a “duh” exercise. Modify the above program

so that when `x` is greater than 0, besides printing `foo` and `bar`, print `baz`.

Exercise. What if you use a different indentation size for the two statements:

```
x = input()
if x > 0:
    print "spam"
    print "eggs"
print "tada!"
```

Does it still work?

Exercise. Prompt the user for the amount in his savings account, his/her monthly salary, and the amount he/she plans to spend this month. If what's left in the savings account at the end of the month is negative, write three print statements. The first prints `Zippo!` The second prints `Nada!` The third prints `Better write home!`

if-else statement

Suppose you want to implement the following in Python:

```
if alien is killed:
    score = score + 10
if alien is not killed:
    move the alien
```

Notice that the condition `alien is killed` and `alien is not killed` is **opposite** to each other.

OK. Great. But Python understand something called the `if-else` statement which does the same thing as the above but in a different way. It's much better than the “two opposite if-statements” method above.

This is an example of an `if-else` statement.

```
x = input()
if x > 42:
    print "spam"
else:
    print "eggs"
```

Read this program entering different values for `x`, and try to guess what it does. Run this program and enter different integer values. You definitely want to try values above 42, 42 and values below 42.

Exercise. Can you leave out the colon `:` after the `else`?

Exercise. You're the hiring tech lead of company Piedon and the interviewee just wrote the following during an interview session:

```
print "Credit card application program"
salary = input("Enter annual salary: ")
if salary >= 20000:
    print "Approved!"
if salary < 20000:
    print "Too bad bro ..."
```

Will you hire him/her? (Duh). And before you call for the next interviewee, correct his/her mistake.

Note that although the indentation size of the statement after

`if` and the indentation size of the statement after `else` can be different (check it out!), you should **not** do so because it makes your program ugly:

```
x = input()
if x > 42:
    print "spam"
else:
    print "eggs" # THIS IS BAD,BAD,BAD!
```

Exercise. After years of research Dr. Bogirs propose that if the product of a person's height (in ft) and weight (in lbs) is less than twice the person's age (in years), then the person IQ must be above 110. Write a program that prompts the user for his height, weight, and age. Print `IQ > 110!!!` if the condition discovered by Dr. Bogirs is true for the inputs. Otherwise print `IQ <= 100 ... eat more fish.`

For the **else** part of the if-else, if you have several statements to execute under the else part, you can also bunch them up into a **block** by giving them the same indentation after the else.

Exercise. Modify the above program

```
x = input()
if x > 0:
    print "spam"
    print "eggs"
print "tada!"
```

so that if `x` is not `> 0`, the program executes two print statements, the first printing `"green"`, the second prints `"ham"`. Run the program with different inputs.

Multiplication Game: `if-else`

Exercise. Modify the Multiplication Game so that if the guess is incorrect the program prints `Incorrect!`. Let me execute my program. First I enter an incorrect answer:

```
What is the product of 97 and 92 ?  
Answer: 1  
Incorrect!  
>>> |
```

and for the second execution, I enter the correct answer:

```
What is the product of 95 and 95 ?  
Answer: 9025  
Correct!  
>>> |
```

Summary

The `if-else` statement look like this:

```
if [condition]:  
    [block of statements]  
else:  
    [block of statements]
```

where `[block of statements]` is a block of one or more statements. `[condition]` is as before.

Some Mental Math

This section has nothing to do with programming. It's just for fun. So you can skip it if you like.

Suppose I told you that it's possible to multiply 98 and 97 very quickly without paper and pencil or calculators or Python. Would you believe me? It does require some practice, but there is a way to multiply two values a little less than 100 very quickly.

Let me show you how with the 98 x 97 example. You subtract the numbers from 100. So $100 - 98 = 2$ and $100 - 97 = 3$. Write this down on a piece of paper:

98	97
2	3

Now look at 98 and 3. Compute $98 - 3$. This is 95. You will need this number. Note that $97 - 2$ is also 95. So there are two different ways of getting 95 with the two diagonals. Write this down:

98	97	95
2	3	

Now look at the 2 and 3. Multiply them together. You get 6. Write this down:

98	97	9506
2	3	

That's it. Here's another example: 96 x 95. First you have:

96	95
4	5

Next you get

96	95	91
4	5	

and finally

96	95	9120
4	5	

The answer is 9120. With some practice you will be able to do the computation in your head. (Using algebra, you can actually prove that the method does work.) So now you can show it off to your math teacher.

if-elif and if-elif-else

Now the `if-else` handles the situation where you want to execute two statements based on two opposite conditions.

```
if my ship collides with alien:
    explode my ship and game ends
else:
    game continues
```

But what if there are more situations to consider? Like so for a two-player game:

```
if my ship collides with partner's ship:
    join ships and increase fire power
if my ship collides with alien:
    explode my ship and game ends
else:
    game continues
```

Of course the above is not a real program. But you get what I'm trying to say. The `if-else` statements specifies one condition and the `if` part executes a statement when the condition is true and the `else` execute a statement when the condition is not true. We want to be able to have **more than one condition**.

Try this program:

```
x = input()
if x < 0:
    print "spam"
elif x < 10:
    print "eggs"
elif x < 42:
    print "green"
else:
    print "ham"
```

Run this program with different input values for `x`. For instance try -1, 2, 20, 100. See if you understand how the program works.

Summary

You now have the following for decision making in Python:

Version 1:

```
if [condition]:  
    [block]
```

Version 2:

```
if [condition]:  
    [block]  
else:  
    [block]
```

Version 2:

```
if [condition]:  
    [block]  
elif [condition]:  
    [block]  
elif [condition]:  
    [block]  
...  
elif [condition]:  
    [block]
```

Version 2:

```
if [condition]:  
    [block]  
elif [condition]:  
    [block]  
elif [condition]:  
    [block]  
...  
elif [condition]:  
    [block]  
else:  
    [block]
```

where `[block]` can be a block of one or more statements
and `[condition]` is of the form

$$[\text{expression}] \text{ [op] } [\text{expression}]$$

where `[expression]` evaluates to an integer value and
`[op]` is one of the following:

==	is equal to
!=	is not equal to
<	is less than
<=	is less than or equal to
>	is greater than
>=	is greater than or equal to

Phew!

Multiplication Game

Exercise. Modify the Multiplication Game so that instead of saying `Incorrect!` when the answer is wrong, the program prints `Incorrect! Too low!` when the guess is less than the product, and it prints `Incorrect! Too high!` when the guess is greater than the product.

Even More??

Believe it or not, the story for the `if` statement and other variations does not end here.

Don't worry. We're not stacking more things onto `if-elif-else`. It's actually the **condition** in the `if` statement that requires a little more work before we're completely done with this set of notes.

Right now the condition in the `if` or `elif` looks like this:

```
[expression] [op] [expression]
```

There are actually situations *not* covered by such a description of a condition.

For instance what if you want to execute a statement (or a block of statements) when **two conditions** are both true?

Here's such an example. In many space shooting games, you can't just fire your laser any time you like. There might be a delay – a power up period. So you might want to program this:

```
if spacebar is pressed and
    laser pods are powered up:
    fire laser
```

As you can see there are **two** conditions. The point is of course: Does Python understand the word **and**?

Before we talk about that ... actually I want to start with “conditions” all over ... from scratch.

Boolean Values

The `if` statement looks like this:

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

We'll look at this `[condition]` business. Actually it can be quite complex. While an integer (or arithmetic) expression is something that evaluates to an integer value, the

`[condition]` is actually called a **boolean expression** which evaluates to a **boolean value**.

Something like 3 or -54 or 42 is an **integer value** (and there are lots of them). A **boolean value** is one of only two things: `True` or `False`.

Try these:

```
>>> print 42  
42  
>>> print True  
True  
>>> print False  
False  
>>> |
```

This tells you that Python understands `True` and `False` because you did not get an error, just as Python understood the integer 42.

WARNING: Here's a common typo among beginning Python programmers: It's `True` and not `true` and `False` and not `false`. (Verify yourself using Python.)

Now try these:

```
>>> x = 0  
>>> y = 1  
>>> print x > 0  
False  
>>> print y > 0  
True  
>>>
```

Next try these:

```
>>> x = 0
>>> y = 1
>>> a = x > 0
>>> b = y > 0
>>> print a
False
>>> print b
True
>>> |
```

Of course `x > 0` is `False`. The example shows that you can give the value of `x > 0` (i.e. `False`) to a variable (i.e. `a`).

So what? Well when Python sees a statement like

```
z = x + y
```

you know that Python computes the value of `x + y` and then give it to `z`. You should think of Python doing two steps. When Python sees something like

```
if x > 0:
    print "spam"
```

Python first computes the boolean value of `x > 0`, and then if the value is `True` Python will execute the statement in the `if` statement. If it is `False`, it will not execute the statement in the `if` statement. Again Python works in two steps.

In the case of the `if-else` and `if-elif` Python computes values of boolean expressions too.

Try these:

```
x = 0
y = 1
print x + y < x * y
print x * y < x + y
```

When Python sees something like these, it (or she or he) will first evaluate the arithmetic expressions and then look at `<`.

Exercise. What is the print out of the following program? First figure it out without Python's help. Next verify your

answer with Python. (This is a drill.)

```
a = 5
b = a + 1
c = a * b
print a * b + a > c * a - 5 * b
print a * b + a <= c * a - 5 * b
print a - c + b == a * b
print (a + 1) * b != c + 5
```

Boolean Operators: and

Suppose you want to write a game where there is a power up delay between laser blasts. You want something like:

```
if spacebar is pressed and
    laser pods are powered up:
    fire laser
```

In other words now you want to execute a statement (fire laser) when two conditions (spacebar is pressed, laser pods are powered up) are both True. It turns out that Python understands the word **and**.

Try these:

```
x = 0
y = 1
z = 2
print x < y and y < z
```

The new word here is **and**.

Exercise. What is the output of the following program?

```
x = 0
y = 1
z = 2
print x < y and z < y
print x > y and x < z
print x == y and z > x
```

It should be clear that if you have the following:

```
[blah1] and [blah2]
```

then this evaluates to True exactly when **both** [blah1] and [blah2] evaluates to True. Otherwise it's False.

You can think of it this way: Just as Python combine two integer values with + to give you a third (example: 1 + 42 gives 43), Python can also combine two boolean values to get a third using and.

Boolean Operators: **or**

There is another way to combine boolean values. Look at this:

```
if my ship collides with laser or
    my ship collides with asteroid:
    explode my ship
```

The **or** is different from the **and**. In the above example, you want the program to explode your ship when either it collides with a laser or when it collides with an asteroid or both. It turns out that Python understands the word **or**.

Try these:

```
x = 0
y = 1
z = 2
print x < y or y < z
```

The new word here is **or**.

Exercise. What is the output of the following program?

```
x = 0
y = 1
z = 2
print x < y or z < y
print x > y or x < z
print x == y or z > x
```

It should be clear that if you have the following:

```
[blah1] or [blah2]
```

then this evaluates to `True` exactly when either one or **both** of `[blah1]` and `[blah2]` evaluates to `True`. Otherwise it's `False`.

Boolean Operator: `not`

There is one last boolean operator called `not`. It's easy to understand it just gives the opposite. Try this and you see what I mean:

```
print not True
print not False
x = 0
y = 1
print not (x < y)
```

Of course `not (x < y)` is the opposite of `x < y` which is `x >= y`. So

```
print not (x < y)
```

is the same as

```
print x >= y.
```

Precedence Rule for Boolean Operators

Now just like you can do `1 + 2 + 3` (two `+`s), you can also have two `ands`:

```
a < b and b < c and c < d
```

In fact you can mix as many `ands` and `ors` as you like. Which brings us to this question:

Exercise. Is the print out `True` or `False`?

```
a = 1
b = 2
c = 3
print a < b and a > c or b < c
```

Which is evaluated first, `and` or `or`?

Chaining up Boolean Expressions

Try this:

```
x = 0
y = 1
z = 2
print x < y < z
print x < z < y
```

The boolean expression

$$x < y < z$$

is the same as

$$x < y \text{ and } y < z$$

WARNING FOR C/C++ AND JAVA PROGRAMMERS: I

have to **warn** you: Many things you learn in Python appear in other languages especially C, C++, Java. (The reason is because C++, Java and Python have the same ancestor – C.) However something like $x < y < z$ is **not valid in most languages**; you have to write $x < y$ and $y < z$, or in the case of C, C++ and Java you write **$x < y \ \&\& \ y < z$** .

Exercise. What if instead of doing $x < y < z$ you do $x < y \leq z$. Does it work? What about $x == y == z$? What about ... just how many possibilities are there???

Print

This is easy

```
print 1, 2
print 3, 4
print 5, 6
```

What if you do this:

```
print 1, 2,
print 3, 4,
print 5, 6,
```

What's the point? This is useful when you want to print things on a single line but you need to do some computation in the middle of the output.

Exercise. Write a program that prints the number 1, the number 2, the string "buckle", if a random integer between 0 and 1 is 0 it prints "my" otherwise it prints "your", and finally it prints "shoes". In other words the program:

- print number 1, number 2, and buckle
- if a random number from 0 to 1 is 0 print my;
otherwise print your
- print shoes

This is what I get when I run the same program seven times:

```
1 2 buckle my shoes
>>> =====
>>>
1 2 buckle my shoes
>>> =====
>>>
1 2 buckle your shoes
>>> =====
>>>
1 2 buckle your shoes
>>> =====
>>>
1 2 buckle your shoes
>>> =====
>>>
1 2 buckle my shoes
>>> =====
>>>
1 2 buckle your shoes
>>> |
```

Pygame

Now that you know the idea of modules and functions, you're ready for a little more Pygame. There are several concepts we have not gone over yet. But if we wait too long you might fall asleep. So once in a while I will say “More on concept X later.”

Here's the code for `bouncing_alien.py` (you can also open up your copy in IDLE):

```
import sys, pygame

# Initialize Pygame
pygame.init()

# Set surface to 680x480
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
BLACK = (0, 0, 0)

# Set speed of alien to [1,2]
XSPEED = 1
YSPEED = 2
speed = [XSPEED, YSPEED]

# Load alien image and get image rect
alien = pygame.image.load("GalaxianAquaAlien.gif")
alienrect = alien.get_rect()

# Load sound
tag = pygame.mixer.Sound("ChatTag.wav")

while 1:

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

    # Move alien
    alienrect = alienrect.move(speed)

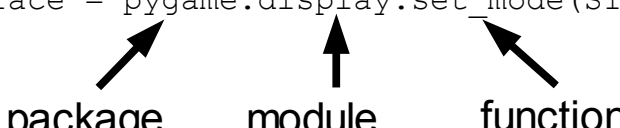
    # Deflect along left and right walls
    if alienrect.left < 0 or alienrect.right > WIDTH:
        speed[0] = -speed[0]
        tag.play()

    # Deflect along top and bottom walls
    if alienrect.top < 0 or alienrect.bottom > HEIGHT:
        speed[1] = -speed[1]
        tag.play()

    # Draw surface
    surface.fill(BLACK)
    surface.blit(alien, alienrect)
    pygame.display.flip()
```

As mentioned earlier pygame is not a module. You can think of it as a bunch of modules: pygame is a package. For instance `pygame` is a package including the module `display` which has a function called `set_mode` ... look at the statement in the code:

```
surface = pygame.display.set_mode(SIZE)
```



package
module
function

Exercise. `bouncing_alien.py` uses four modules in pygame:

- `display`
- `image`
- _____ (find it in the program)
- _____ (find it in the program)

Pygame has the following:

`pygame.init()`

This initializes the pygame package. You have to call it before using anything in pygame.

`pygame.display.set_mode`

This creates and returns a surface (the window) for drawing. The parameter looks like (a, b) where a is the width of the surface and b is the height of the surface. Both a and b are integers. However the “thing” (a,b) is called **tuple**. You can think of a tuple like (a,b) as made up of two integers. More on tuples later.

So after a statement like this in `bouncing_alien`:

```
surface = pygame.display.set_mode(SIZE)
```

you have a surface variable, i.e. `surface`, to draw on. How do you draw on a surface variable? There are basically two different way of doing that.

- You either draw a shape (line, circle, rectangle, ...) or
- You copy an image onto a region of the surface.

Let's do the second.

Pygame: Images

There are two things you have to do: You need to load the image from a file into a variable and you need to draw it to the surface.

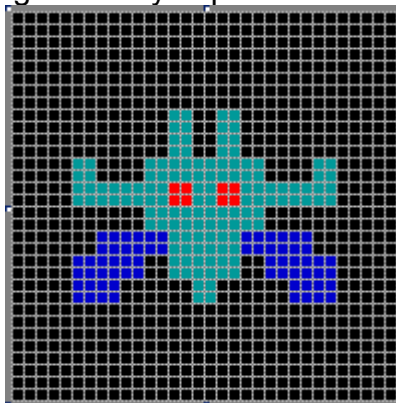
`pygame.image.load`

This is how you load an image file into a variable:

```
alien = pygame.image.load("GalaxianAquaAlien.gif")
```

The variable is called `alien`. The input into `pygame.image.load` is the name of the image file.

We want to put the image onto the surface. Now images are rectangular. For instance the `GalaxianAquaAlien.gif` image is a rectangular array of pixels:

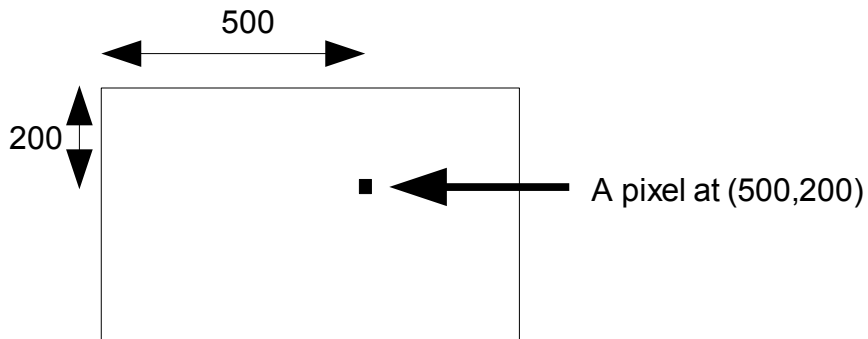


(If you wish, you may look for a program called mspaint and load the file into this program. Play around with this program and you should be able to get the above grid view.)

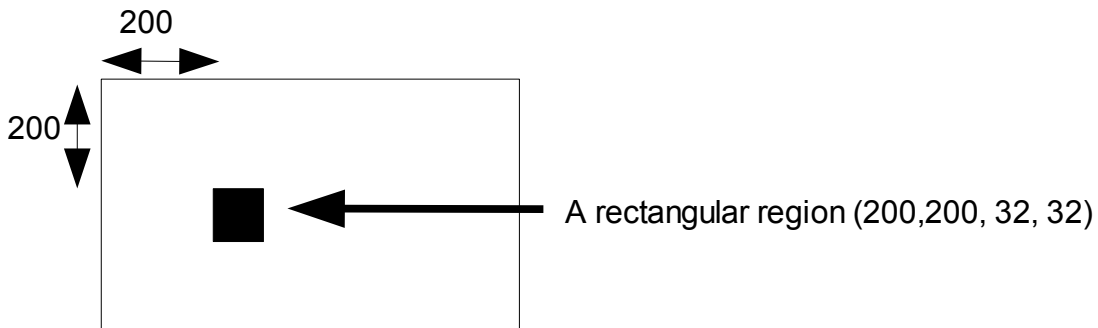
In particular, it's a 32-by-32 pixel image. You can copy this image to the `surface` by specifying a rectangular region on `surface`. This copy operation is called blitting. The word **blit** comes from **block image transfer**. To blit onto the surface we will need to talk about the geometry of the surface.

Pygame: Surface

The pixels on the surface is numbered in this way. The position of each pixel is measured by two integers: x and y. x is the distance from the left edge and y is the distance from the top edge. The position for such a point is written (x,y). For instance:



Note that the point (0,0) is the top left corner of the surface. A rectangular region is given by the (x,y) and the width and height of the rectangle. For instance this is a 32-by-32 area with its top-left corner at (200,200). You can describe the rectangular region by (200, 200, 32, 32).



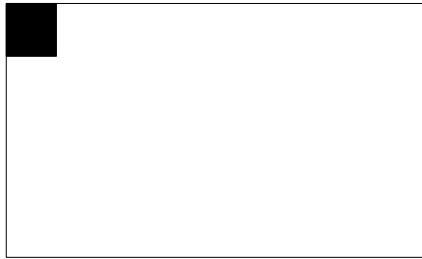
Now let's copy the GalaxianAquaAlien.gif image onto a rectangular region. This is done by the

```
surface.blit(alien, alienrect)
```

where `alienrect` is a rectangular region. Where does `alienrect` come from? It was created here:

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

`alientrect` was initially the rectangle (0, 0, 32, 32), i.e., a rectangle at the top left corner (0,0) of the surface with width 32 and height 32:



Pygame: Game Loop

The next chunk of code is the **heart** of the program. It's

```
while 1:
    # Exit if window is closed
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()

    # Move alien
    alienrect = alienrect.move(speed)

    # Deflect along left and right walls
    if alienrect.left < 0 or alienrect.right > WIDTH:
        speed[0] = -speed[0]
        tag.play()

    # Deflect along top and bottom walls
    if alienrect.top < 0 or alienrect.bottom > HEIGHT:
        speed[1] = -speed[1]
        tag.play()

    # Draw surface
    surface.fill(BLACK)
    surface.blit(alien, alienrect)
    pygame.display.flip()
```

This is the block
of the while
statement

called the **game loop**. While the `if` statement controls conditional execution, **the while statement controls repetition**. The above `while` statement has a **huge** block. We will talk more about the `while` loop later. You only need to know that the `while` executes the block of statements repeatedly.

The structure of our game loop looks like this:

```
continually do the following:
    process user inputs/events
    move the objects to draw
    draw all images
```

Now let's go over the block of statements inside the `while` statement.

Game Loop Part 1:

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

does the following: If the user clicks on X on to top-right corner of the window,



the program closes. You see that there is function `sys.exit()`. `exit()` is a function in the `sys` module. This is a command to Python to stop running the program. This is all you need to know about this part of the `while` loop for now.

Game Loop Part 2: Part 2 of the game loop:

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

# Deflect along left and right walls
if alienrect.left < 0 or alienrect.right > WIDTH:
    speed[0] = -speed[0]
    tag.play()

# Deflect along top and bottom walls
if alienrect.top < 0 or alienrect.bottom > HEIGHT:
    speed[1] = -speed[1]
    tag.play()
```

involves moving `alienrect`. As you can see there are three parts.

The first

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

moves the rectangle where we will blit the alien's image. The amount moved is in variable `speed`. Look at the top part of the code:

```
# Set speed of alien to [1,2]
XSPEED = 1
YSPEED = 2
speed = [XSPEED, YSPEED]
```

In each execution of the block of the `while` statement, the alien's rectangle moved by 1 pixel to the right and 2 pixels down. Note that the “thing” `[1,2]` is new. This is a **list**.

We'll talk about them later. That's all I will explain.

Exercise. Change the value of `YSPEED` to 0. Run the program. Do you see why it works this way?

The second

```
# Deflect along left and right walls
if alienrect.left < 0 or alienrect.right > WIDTH:
    speed[0] = -speed[0]
    tag.play()
```

does exactly what the comment says. There is an `if` statement. `alienrect.left` is the x value of `alienrect`. `alienrect.right` is the sum of x and width of `alienrect`. The condition is the same as saying that the `alienrect` is beyond the left side or the right side of the surface. If the condition is `True`, the program reverses the speed along the x-axis (it turns around) and a sound is played.

The third part is similar to the second except that it checks if the alien's rectangle is beyond the top or bottom of the surface.

Game Loop Part 3:

```
# Draw surface
surface.fill(BLACK)
surface.blit(alien, alienrect)
pygame.display.flip()
```

`surface.fill(BLACK)` paints the whole surface black. It's the same as clearing the surface. The statement

```
surface.blit(alien, alienrect)
```

blits the image to the main surface at `alienrect`. I will only say that `pygame.display.flip()` is related to double-buffering of video surfaces. Just remember to do a flip when you're done with all your drawing.

The following exercise is very

important. I'll show you how to slow down `bouncing_alien.py` and also to print out the `alienrect`. Modify `bouncing_alien.py` as shown:

```
bouncing_alien.py - C:\Docu
import sys, pygame

# Initialize Pygame
pygame.init()

# Set surface to 680x480
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
BLACK = (0, 0, 0)

# Set speed of alien to [1,2]
XSPEED = 1
YSPEED = 2
speed = [XSPEED, YSPEED]

# Load alien image and get image rect
alien = pygame.image.load("GalaxianAquaAlien.gif")
alienrect = alien.get_rect()
print alienrect

# Load sound
tag = pygame.mixer.Sound("ChatTag.wav")

while 1:

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

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

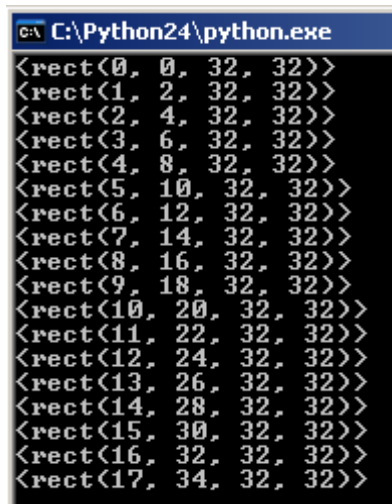
    # Deflect along left and right walls
    if alienrect.left < 0 or alienrect.right > WIDTH:
        print "left or right collision", alienrect.left, alienrect.right
        speed[0] = -speed[0]
        tag.play()

    # Deflect along top and bottom walls
    if alienrect.top < 0 or alienrect.bottom > HEIGHT:
        print "top or bottom collision", alienrect.top, alienrect.bottom
        speed[1] = -speed[1]
        tag.play()

    # Draw surface
    surface.fill(BLACK)
    surface.blit(alien, alienrect)
    pygame.display.flip()

    pygame.time.delay(200)
```

After modifying the program, save it. Run `bouncing_alien.py` by double-clicking on it. You notice that the program runs a lot slower. Furthermore the console window shows the result of the `print alienrect` statements:

A screenshot of a Windows command prompt window titled "C:\Python24\python.exe". The window displays a series of 18 lines of Python output, each representing the state of an 'alienrect' object. The output is as follows:

```
<rect(0, 0, 32, 32)>  
<rect(1, 2, 32, 32)>  
<rect(2, 4, 32, 32)>  
<rect(3, 6, 32, 32)>  
<rect(4, 8, 32, 32)>  
<rect(5, 10, 32, 32)>  
<rect(6, 12, 32, 32)>  
<rect(7, 14, 32, 32)>  
<rect(8, 16, 32, 32)>  
<rect(9, 18, 32, 32)>  
<rect(10, 20, 32, 32)>  
<rect(11, 22, 32, 32)>  
<rect(12, 24, 32, 32)>  
<rect(13, 26, 32, 32)>  
<rect(14, 28, 32, 32)>  
<rect(15, 30, 32, 32)>  
<rect(16, 32, 32, 32)>  
<rect(17, 34, 32, 32)>
```

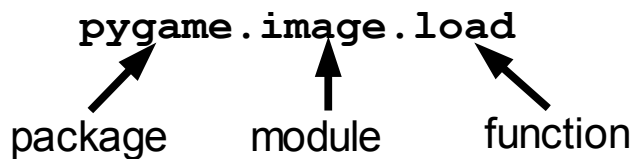
As you can see the initial value of `alienrect` is `(0,0,32,32)`. This means that the top-left corner of the rectangle is at coordinates `(0,0)` and it has width 32 and height 32. The next is `(1, 2, 32, 32)`. Etc. Run it until you see a collision with a side of the surface.

Inserting print statements in order to **understand a program** is **extremely important**. It's also important when it comes to **debugging** a program – i.e. figuring out what the program is doing in order to track down an error.

An Object Quickie

Look at `alien.get_rect()`. It looks suspiciously like `random.randrange!!!`

Well, for `random.randrange`, `randrange` is a function inside the `random` module. But `alien` is an image loaded using `pygame.image.load` which is a function (i.e. `load`) inside a module (i.e. `image`) which is in a package (i.e. `pygame`):



Does that mean that `alien` is a module?

No.

Actually the variable `alien` is an **object**. And yes, you can have **functions inside objects**, just like you can have functions inside modules: that's why you have

```
alien.get_rect()
```

There are crucial differences between a module and an object but I do not want to get into it right now. Just remember that you can have functions inside both modules and objects.

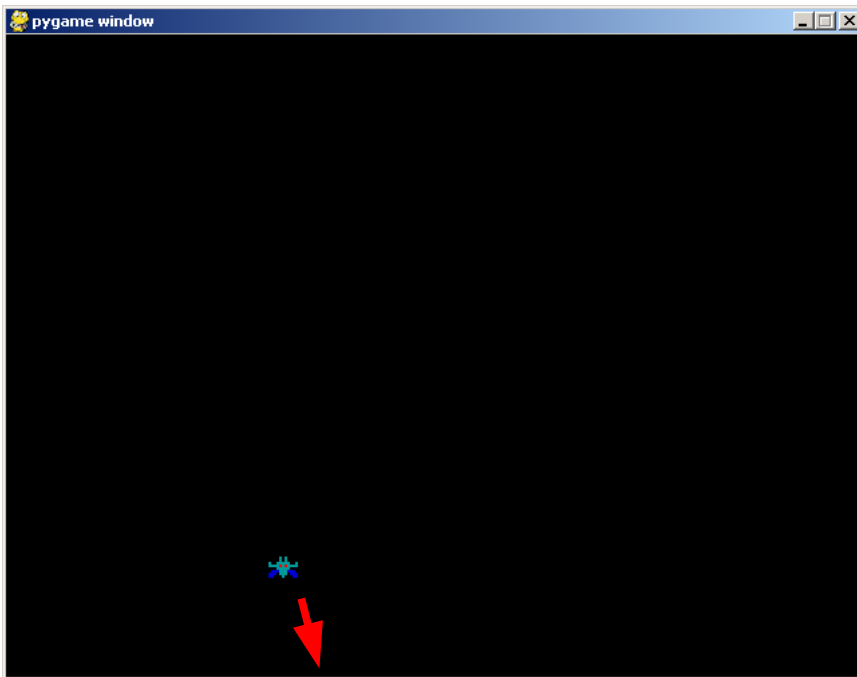
In fact there is another object: `surface` is also an object. That's why you see

```
surface.fill(BLACK)
```

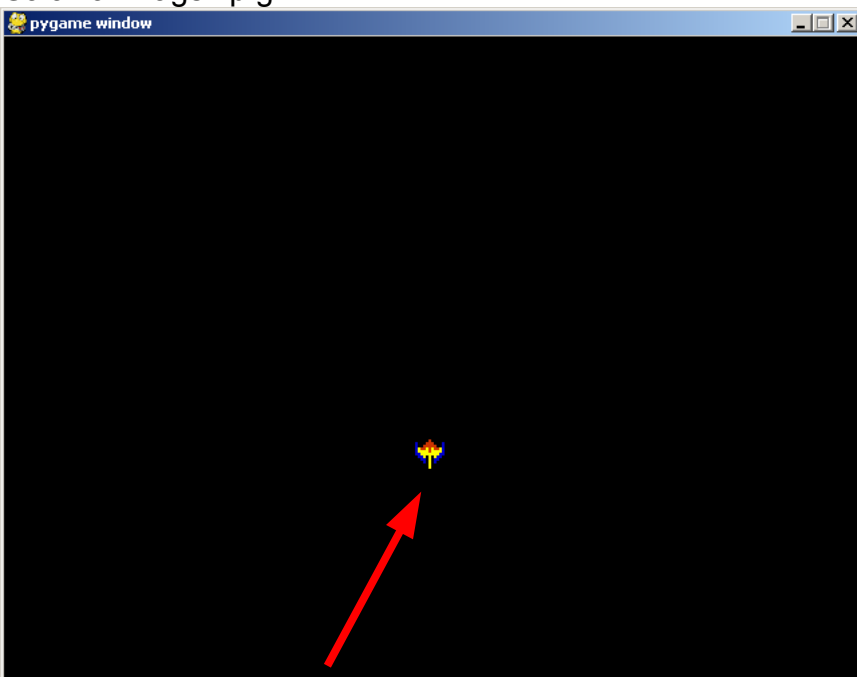
Exercise. Can you find another object in modified program?

Your First Pygame Project

Modify `bouncing_alien.py` so that it behaves just as before except that the alien changes its image whenever it hits a wall. In other words, it starts off with the aqua alien image `GalaxianAquaAlien.gif`,



and when it hits a wall, its image becomes `GalaxianFlagship.gif`.



When the alien hits another wall it changes its image to `GalaxianAquaAlien.gif`, etc.

The hints are on the next page. Look at the hints only if you need to.

Your First Pygame Project: Hints

You need both images. So create two alien objects. You can give your alien objects any name you like. For instance you can call them `alien0` and `alien1`.

Now think about it. You need to know which alien image to blit onto the surface: is it `alien0` or `alien1`?

Create a variable to remember this. Let's call it `i`. When `i` is 0, you blit with `alien0` and when `i` is 1 you blit with `alien1`. You start off with `i = 0`. So your program looks like this:

```
import sys, pygame

# Initialize Pygame
pygame.init()

# Set surface to 680x480
WIDTH = 640
HEIGHT = 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)
BLACK = (0, 0, 0)

# Set speed of alien to [1,2]
XSPEED = 1
YSPEED = 2
speed = [XSPEED, YSPEED]

# Load alien image and get image rect
alien0 = pygame.image.load("GalaxianAquaAlien.gif")
alien1 = pygame.image.load("GalaxianFlagship.gif")
alienrect = alien0.get_rect()
i = 0

# Load sound
tag = pygame.mixer.Sound("ChatTag.wav")
```

Now look for places where the alien hits a wall. At those places, you need to “toggle” the value in variable `i`. In other words if `i` is 0, it should become 1 and if `i` is 1 it should become 0.

Now look for the place where there program blits the alien's image. You need to change it so that if `i` is 0, you blit with `alien0` and if `i` is 1 you blit with `alien1`.