

Pygame Part 4: Dictionary

Objectives

- Create dictionaries
- Add entries to a dictionary
- Modify the value of a key-value pair of a dictionary
- Retrieve the keys from a dictionary
- Retrieve the key-value pairs of a dictionary as a list of tuples
- Use dictionary to aggregate variables
- Write and use modules

Packing Data: Dictionaries

You should realize that our more interesting programs are getting longer and longer.

Forget about programming for the time being and focus on problem solving.

Good problem solvers always solve problems by breaking a problem into smaller (and hopefully simpler!) pieces. So let's see how our programs can be made up of smaller units.

Let's go back to the electric rays project. The whole wave of electric rays is made up of lines. So you can think of a "wave" is made up of "lines" and furthermore a "line" is made up of two "points". The above performs a breakdown of "things". You can also break down an "action". "Moving a wave" is then broken up into "moving a line" which is broken up into "moving a point".

But right now let's just focus on a line.

In our program. We "model" a line with 9 "things" and we collect them together into a list. For instance the line [100, 50, 1, -2, 30, 25, 2, 3, (200, 150, 205)] represents a line where one endpoint is at (100, 50) which is traveling at the speed of 1 on the x-axis and -2 on the y-axis, etc.

It's easy to remember which value gives the color: It's the last because you can see that it's made up of 3 integer values. But the problem is other 8 numbers are all the same.

Someone else might want to write [100, 1, 50, -2, 30, 2, 25, 3, (200, 150, 205)] instead.

Both are correct. It's just a matter of where you want to put your data in the list.

But the problem is that it can be very error-prone. One good way to tell if a program is well written is ask the following question: Is the program easy to read? Can it be written in a simpler and more readable way?

To make the program more readable, I'm going to use

something called a **dictionary** to model lines. In other words we're going to rewrite our electric ray program, using dictionaries instead of lists to model lines.

Only for the experts: Instead of using classes and objects, I'm using dictionaries to model objects. The dictionary is easier to understand. In fact, in Python, objects are internally implemented using dictionaries.

Dictionaries

Try this:

```
d = {0: 3.1415, 1: 2.718}
print d
print d[0]
print d[1]
```

The variable `d` is called a **dictionary**.

Read the above program carefully. Now compare the above `d` (a dictionary) with the following `e` (a list):

```
e = [3.1415, 2.718]
print e
print e[0]
print e[1]
```

At first sight it seems that a dictionary is the same as a list: we are just writing the data in different ways:

It seems like the only difference is that for a dictionary, you have to specify the index values and you have to use `{ }` instead of `[]`:

```
d = {0: 3.1415, 1: 2.718}
e = [3.1415, 2.718]
```

Actually a dictionary is very different. Try this:

```
height = {"John": 5.5, "Jane": 6.1}
print height["John"]
print height["Jane"]
```

In this example you see whereas in a list such as `e`, `e[0]` refers to the value at the zero-th position of `e`, for the dictionary `height` above, you can refer to a value with a string.

In other words the `0` in `e[0]` is positional. The `"John"` in `height["John"]` has nothing really to do with a position.

Here's how you should think of a dictionary. You should think of a dictionary as a **table**. You can create an empty

dictionary like this:

```
d = {}
```

The dictionary is a table with ***two columns***. You can insert as many rows as you like. For instance if you do this:

```
d = {}
d["foo"] = 1234
```

You can think of your dictionary (as a table) like this after the above statement:

"foo"	1234
-------	------

Let's add another row:

```
d["bar"] = 5678
```

Now d looks like this

"foo"	1234
"bar"	5678

This is only a visual guide. (An actual dictionary is a lot more complex).

This is what happens when you refer to `d["foo"]`: Python will look for the row where the first column is "foo" and then `d["foo"]` will then refer to the second column of the row that is found.

"foo"	1234
"bar"	5678

← This is `d["foo"]`

For that row, "foo" is the **key** while 1234 is the **value** (for that key). That's why a row in a dictionary is sometimes called a **key-value pair**.

Exercise. Can you declare an empty list:

```
e = []
```

and then add things like this (just like what we did with a dictionary)?

```
e[0] = "mamamia!"
```

Of course you can also use variables when you create key-value pairs in a dictionary:

```
d = {}  
k = "foo"  
v = 1234  
d[k] = v
```

The following program uses this idea to add key-value pairs to a dictionary:

```
d = {}  
lastname = raw_input("lastname: ")  
while lastname != "":  
    firstname = raw_input("firstname: ")  
    d[lastname] = firstname  
    print d  
    lastname = raw_input("lastname: ")  
  
print "\nlastname, firstname table:"  
print d
```

Modifying A Value in a Dictionary

You can modify the value of a key-value pair in a dictionary:

```
foo = {}
foo["bar"] = 42
print foo
foo["bar"] = "life, universe, everything"
print foo
```

You can check if a key is present in a dictionary:

```
hoursplayed = {"sim2": 200, "doom3": 5000}
print hoursplayed.has_key("sim2")
print hoursplayed.has_key("unreal")
```

Exercise. Write a program that continually prompts the user for the name of a computer game and the number of hours spent playing the game. If the name entered is an empty string "", the program stops prompting for data and prints everything in the dictionary. If the name is already found in the dictionary, the program prints an error message. Here's a skeleton:

```
hoursplayed = {}
name = raw_input("name: ")
while name != "":

    # FILL IN THE BLANK!

    name = raw_input("name: ")

print hoursplayed
```

Recall that you can scan the characters in a string one at a time like this:

```
s = "foobar"

for c in s:
    print c
```

(Don't forget that a character is just a string of length 1.)

Here's a simple program to count the number of times a character appears in a string:

```
s = "life, universe, and everything"
```

```
frequency = {}
for c in s:
    print "character", c,
    if frequency.has_key(c):
        frequency[c] = frequency[c] + 1
    else:
        frequency[c] = 1
    print "has frequency", frequency[c]

print frequency
```

Basically the program does the following: `c` scan across all the characters of the string `s`. If `c` is not a key in the dictionary, we insert `c` with value `1` into the dictionary. Otherwise we increment the value for key `c` by `1`. Study the above program carefully.

Frequency counting like this can be used to crack secret codes. You can encrypt a message by replacing characters. For instance you can replace `a` by `g`, `b` by `j`, etc. So to crack a message like “`alkjsdhsdidfsdlbjasldjsd`”, you count the frequencies and try to replace the most frequently occurring character by `e` or `t`, etc. because it's well known that some characters occur more frequently than others. Of course the secret message has to be “long” for this method to work ... and it has to be written in English!!!

Exercise. Write a program that continually prompts the user for the name of a computer game and the number of hours spent playing the game. If the name entered is an empty string “”, the program stops prompting for data and prints everything in the dictionary. If the name is already found in the dictionary, the program prints an error message. Here's a skeleton:

```
hoursplayed = {}
name = raw_input("name: ")
while name != "":

    # FILL IN THE BLANK!

    name = raw_input("name: ")

print hoursplayed
```


Removing a Key-Value Pair from a Dictionary

I've already shown you how to add a key-value pair.

You can remove a key-value pair from the dictionary. Here's how you do it:

```
hoursplayed = {"sim2": 200, "doom3": 5000}  
del hoursplayed["sim2"]  
print hoursplayed
```

That's all there is to it.

Running Across a Dictionary

There are several ways to run across all the rows in a dictionary.

A dictionary is actually an object. You can get all the keys (as a ***list***) from a dictionary like this:

```
hiscore = {}  
hiscore["john"] = 32  
hiscore["jane"] = 500  
  
print hiscore.keys()
```

Exercise. In the above dictionary, the first key inserted into the dictionary was “john”. In the printout of the above program, did “john” appear first?

With the list of key values, you can scan across a dictionary:

```
hiscore = {}  
hiscore["john"] = 32  
hiscore["jane"] = 500  
  
for name in hiscore.keys():  
    print name, hiscore[name]
```

Exercise. Modify the following program so that instead of printing the dictionary, it prints each key-value pair on a separate line:

```
s = "life, universe, and everything"  
  
frequency = {}  
for c in s:  
    if frequency.has_key(c):  
        frequency[c] = frequency[c] + 1  
    else:  
        frequency[c] = 1  
  
print frequency
```

You should get this output:

```
a 1  
 3  
e 5
```

```
d 1
g 1
f 1
i 3
h 1
, 2
l 1
s 1
r 2
u 1
t 1
v 2
y 1
n 3
```

Exercise. Recall that you can sort a list:

```
x = [5,3,1,2,4]
print x
x.sort()
print x
```

You can also sort tuples.

```
x = [(5,2), (3,3), (1,3), (2,4), (4,1), (3,1)]
print x
x.sort()
print x
```

Notice that Python will sort (in ascending order) by the first number in the tuples. If the first numbers of two tuples are the same, their second numbers are used to break the tie.

Now modify the above program

```
s = "life, universe, and everything"

frequency = {}
for c in s:
    if frequency.has_key(c):
        frequency[c] = frequency[c] + 1
    else:
        frequency[c] = 1

print frequency
```

so that the output prints the frequencies sorted in **descending** order. You should get this output:

```
e 5
```

```
n 3
i 3
 3
v 2
r 2
, 2
y 1
u 1
t 1
s 1
l 1
h 1
g 1
f 1
d 1
a 1
```

[Hint: Scan the dictionary but instead of printing each key-value pair, append the key-value pair to a list as a tuple with the value as first entry and key as second entry. Sort the list. Scan the list and print the contents one item on a line. You have to reverse the list because the list is sorted in the ascending order. The solution is given below.]

You can also create a list of key-value pairs as tuples in the following way:

```
hiscore = {}
hiscore["john"] = 32
hiscore["jane"] = 500

print hiscore.items()

for key,value in hiscore.items():
    print key, value
```

Solution to last exercise.

```
s = "life, universe, and everything"

frequency = {}
for c in s:
    if frequency.has_key(c):
        frequency[c] = frequency[c] + 1
    else:
        frequency[c] = 1
```

```
list = []
for key in frequency.keys():
    list.append((frequency[key], key))

list.sort()

for f, c in list[::-1]:
    print c, f
```

(There are many ways to solve this problem.)

To analyze the frequencies of the characters, you need a pretty long string. I'm too lazy to type in a long string. No problem. If you have internet connection, you can get Python to download a string from the web. The Project Gutenberg has lots of books online. Here's the URL for Doyle's "The Hounds of Baskervilles":

<http://www.gutenberg.org/dirs/etext01/bskrv11.txt>

Open your web browser and point to that URL and you should be able to read the story.

This is how you download a string using a URL:

```
import urllib
url = "http://www.gutenberg.org/dirs/etext01/bskrv11.txt"
s = urllib.urlopen(url).read()
print s
```

Make sure what you see on your screen matches your browser. So now you can analyze the frequencies on the character in string s.

Using the string from the URL

<http://www.gutenberg.org/dirs/etext01/bskrv11.txt>

and the previous program this is the frequencies I get:

```
59524
e 30989
t 22818
o 19959
a 19914
n 16739
h 16209
i 15769
s 15194
r 14925
```

```
d 10525
l 9898
u 7738
```

```
7531
m 6491
w 6024
c 5941
f 5267
y 5262
g 4428
p 3884
, 3572
. 3515
b 3262
" 2800
v 2660
k 1973
I 1961
H 839
T 783
S 648
- 584
W 546
? 522
B 473
x 466
' 449
A 399
M 375
C 278
D 251
Y 230
N 229
E 197
! 195
q 193
O 191
L 179
j 158
z 127
G 125
R 124
P 121
```

```
F 105
* 65
; 51
U 50
l 48
O 44
J 39
V 34
: 33
2 31
] 22
[ 22
) 22
( 22
8 17
3 16
9 13
7 13
4 13
/ 12
X 10
Q 9
@ 8
6 8
5 7
K 5
% 3
~ 2
> 2
= 2
# 2
_ 1
< 1
+ 1
$ 1
```

Note that lower and uppercase characters are counted separately. Can you modify your program so that they are combined?

Values for the Key-Value Pairs

You can put any value into the value part of a key-value pair for a dictionary: integers, strings, ... and even a dictionary!

Try this:

```
d = {}
d[1] = 2
d["buckle"] = "my shoe"
print d

e = {}
e["song"] = d
print e
```

But you **cannot** use anything as a key. Try this:

Exercise. From the following example, what type of values cannot be used for keys?

```
d = {}

"integer as key ..."
d[1] = 2

"string as key ..."
d["buckle"] = "my shoe"
print d

"tuple as key ..."
d[(1, 2)] = "buckle my shoe"
print d

"list as key ..."
d[[1, 2]] = "buckle my shoe"
print d

"dictionary as key ..."
e = {1: 2, "buckle": "my shoe"}
d[e] = "not my fav song"
```

Do you remember mutable and immutable values? An immutable value is a value that cannot be changed. (Remind yourself to check previous your notes later.) Keys must be immutable. For the time being just remember that you can only use integers, floating point numbers (example: 1.231),

boolean values (`True` and `False`), strings as keys. You can also use tuples involving integers, floating point numbers, boolean values, strings.

Dictionary and Hash Table (Optional)

You might skip this on a first reading.

The following gives you a better picture than the table that I have given you above – although the description given here is still not complete.

Actually a dictionary is not just a table that you fill with rows in such a way that you keep adding rows starting with an empty table.

You should think of a dictionary as a table *already* with rows. Rows are marks as used or unused. Initially every row is marks as unused. Suppose you create an empty dictionary:

```
d = { }
```

Python will create a table with rows marks as unused:

On executing

```
d["foo"] = 1234
```

Python computes an index position using a special function that converts the string "foo" into an “appropriate” integer that can then be converted into an index position into the table. Such a function is called a **hash function**.

Let's just say that the hash of "foo" converts to index 1 of the table. So we have this picture:

"foo"	1234

Right now one row is marked as “used”. Three rows are marks as unused.

In Computer Science, the above table is called a **hash table**.

By the way it's easy to find the hash value of "foo" :

```
hash("foo")
```

(This is not used directly as an index value. This integer value has to be converted first.)

The hash table created by Python is actually much larger. By the way, the hash function can take two different keys and return the same hash value. This will create a problem: The index positions for these keys will be the same! I won't go into the "collision resolution" of such situations. I will only say that a good hash function is a very good "scrambling" function in the sense that values are hashed in such a way that the table is filled as "uniformly" as possible in order to minimize collision of hash values.

Electric Rays Again

Let's look at our line from the first section again: [100, 50, 1, -2, 30, 25, 2, 3, (200, 150, 205)]

Again we do not want to depend on our ability to remember meaning of the positions within the line.

Let's give the values in our line some names. I'll use `x,y` to denote the coordinates of one endpoint and `X,Y` for the other. Furthermore, I'll use `xspeed, yspeed` for the speed (or velocity) of the point `x,y` and `Xspeed, Yspeed` for the speed for the point `X, Y`. Furthermore I'll use `color` for the color of the line.

Using this naming scheme, I can now remodel our line:

```
line = [100, 50, 1, -2, 30, 25, 2, 3, \
        (200, 150, 205)]
```

as:

```
line = {}
line["x"] = 100
line["y"] = 50
line["xspeed"] = 1
line["yspeed"] = -2
line["X"] = 30
line["Y"] = 25
line["Xspeed"] = 2
line["Yspeed"] = 3
line["color"] = (200, 150, 205)
```

Exercise. Go ahead and take the electric ray program and modify it so that each line is a dictionary. The lines variable will become not a list of lists but a list of dictionaries. Mind-boggling ... but think about it a bit ... and just hit the code. Everything will become clear after trying out your knowledge of dictionaries on the program.

The solution is in the next section. Don't peek without trying it out first!

By the way you already know that when you create a rect, say `r`, you get `r.x`, `r.y`, `r.w`, and `r.h`. It's possible to

create a line variable `line` so that we have `line.x`, `line.y`, etc. in other words `line` is an object. We're doing something similar but using dictionaries.

Using dictionary	Using object
<code>line["x"]</code>	<code>line.x</code>

I'll show you how to create objects much much later.

Solution to the Dictionary Version of Electric Rays

OK. Here's the solution.

```
import pygame, sys, random
pygame.init()
random.seed()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

BLACK = (0,0,0)

def move(d, v, m):
    d = d + v
    if d > m:
        d = m
        v = -v
    elif d < 0:
        d = 0
        v = -v
    return d, v

while 1:
    lines = []

    Rspeed, Gspeed, Bspeed = random.randrange(-5,6), random.randrange(-5,6), \
                               random.randrange(-5,6)
    R, G, B = (random.randrange(256), random.randrange(256), \
               random.randrange(256))

    x, y = random.randrange(WIDTH), random.randrange(HEIGHT)
    xspeed, yspeed = random.randrange(1,2) * random.randrange(-1,2,2), \
                     random.randrange(1,2) * random.randrange(-1,2,2)

    X, Y = random.randrange(WIDTH), random.randrange(HEIGHT)
    Xspeed, Yspeed = random.randrange(1,2) * random.randrange(-1,2,2), \
                     random.randrange(1,2) * random.randrange(-1,2,2)

    for i in range(100):
        R, Rspeed = move(R, Rspeed, 255)
        G, Gspeed = move(G, Gspeed, 255)
        B, Bspeed = move(B, Bspeed, 255)
```

```

    color = (R, G, B)

    # NEW CODE:
    line = {}
    line["x"] = x
    line["y"] = y
    line["xspeed"] = xspeed
    line["yspeed"] = yspeed
    line["X"] = X
    line["Y"] = Y
    line["Xspeed"] = Xspeed
    line["Yspeed"] = Yspeed
    line["color"] = color

    lines.append(line)

FRAME_RATE = 1000.0 / 60

count = 0
num_lines = 1

    initial_time = pygame.time.get_ticks()

while 1:

    next_initial_time = pygame.time.get_ticks()
    if next_initial_time - initial_time > 60000:
        break

    starttime = pygame.time.get_ticks()

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()

    # Move lines in lines[:num_lines]
    for line in lines[:num_lines]:
        # NEW CODE
        line["x"], line["xspeed"] = move(line["x"], line["xspeed"], WIDTH - 1)
        line["y"], line["yspeed"] = move(line["y"], line["yspeed"], HEIGHT - 1)
        line["X"], line["Xspeed"] = move(line["X"], line["Xspeed"], WIDTH - 1)
        line["Y"], line["Yspeed"] = move(line["Y"], line["Yspeed"], HEIGHT - 1)

    surface.fill(BLACK)

    # Draw lines in lines[:num_lines]
    for line in lines[:num_lines]:
        # NEW CODE
        pygame.draw.line(surface, line["color"], (line["x"],line["y"]), \
                        (line["X"],line["Y"]))

    pygame.display.flip()

    if num_lines < 100:

```

```
        count = count + 1

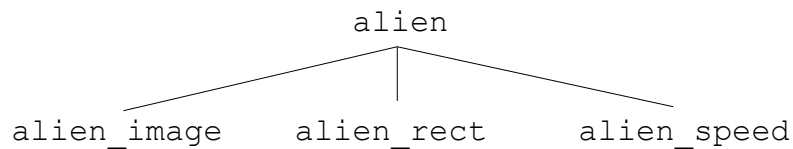
    if count >= 10:
        if num_lines < 100:
            num_lines = num_lines + 1
            count = 0

    endtime = pygame.time.get_ticks()
    totaltime = starttime - endtime
    timeleft = int(FRAME_RATE - totaltime)
    if timeleft > 0:
        pygame.time.delay(timeleft)
```


Dictionary-Cleanup of Our Galaxian

Refer to the simple galaxian game from the previous set of notes. We will now go through a series of “clean-up” operations to break the program up into pieces.

First we will create “higher level concepts”. What I mean is this: We have `alien_image`, `alien_rect`, `alien_speed`. These are three different things. We want to create an `alien` concept that contains the above three concepts. When we work with too many concepts, we will then prefer to think of the alien concept only. Here's the hierarchy of the 4 different concepts:



Here's what we'll do: We'll create a dictionary called `alien`. The dictionary `alien` has keys `"image"`, `"rect"`, and `"speed"`. In other words, `alien_image` is packaged into `alien` as `alien["image"]`.

Here's your task: Modify your simple galaxian program (from last week) so that you have the following dictionaries:

- `alien`
- `ship`
- `laser`
- `explosion`

From now on I'll refer to the simple galaxian program (from last week) as `simplegalaxian.py` because that's what I called mine.

Hint: The dictionary `alien` has keys `"image"`, `"rect"`, and `"speed"`. `alien_image` in `simplegalaxian.py` should be replaced by `alien["image"]`.

Solution

```
import pygame, sys, random
pygame.init()
random.seed()

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

pygame.key.set_repeat(10, 10)

SCORE_HEIGHT = 24
BLACK = (0, 0, 0)
RED = (255, 0, 0)
sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

# Create alien
alien = {}
alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
alien["rect"] = alien["image"].get_rect()
alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
alien["speed"] = [1, 0]
ALIEN_Y_INCREMENT = 10

# Create flagship
ship = {}
ship["image"] = pygame.image.load("GalaxianGalaxip.gif")
ship["rect"] = ship["image"].get_rect()
x = (WIDTH - ship["rect"].w) / 2
y = HEIGHT - ship["rect"].h
ship["rect"] = ship["rect"].move([x, y])
ship["speed"] = [0, 0]

# Ship's laser
laser = {}
laser["rect"] = pygame.Rect(0, 0, 4, 8)
laser["speed"] = [0, 0]
laser["alive"] = False
# Laser sound
laser["sound"] = pygame.mixer.Sound("laser.wav")

# Explosion sound
explosion = {}
explosion["sound"] = pygame.mixer.Sound("gexplode.wav")
explosion["played"] = False

def move(d, v, m):
    d = d + v
    if d < 0:
        d = 0
        v = -v
```

```

elif d > m:
    d = m
    v = -v
return d, v

collides = False
while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            keypress = pygame.key.get_pressed()
            if keypress[pygame.K_LEFT]:
                ship["speed"] = [-1, 0]
            if keypress[pygame.K_RIGHT]:
                ship["speed"] = [1, 0]
            if keypress[pygame.K_SPACE]:
                if not laser["alive"]:
                    laser["alive"] = True
                    laser["speed"] = [0, -2]
                    laser["rect"].x = ship["rect"].x + (ship["rect"].w - \
                                                            laser["rect"].w)/2
                    laser["rect"].y = ship["rect"].y - laser["rect"].w
                    laser["sound"].play()

    surface.fill(BLACK)

    if not collides:
        alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                                    alien["speed"][0], WIDTH - alien["rect"].w)
        if random.randrange(100) == 0:
            alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT

        ship["rect"].x, ship["speed"][0] = move(ship["rect"].x, \
                                                    ship["speed"][0], WIDTH - ship["rect"].w)
        ship["speed"] = [0, 0]

        if laser["alive"]:
            laser["rect"].y, laser["speed"][1] = move(laser["rect"].y, \
                                                        laser["speed"][1], HEIGHT - laser["rect"].h)
            if laser["rect"].y < SCORE_HEIGHT:
                laser["alive"] = False

    collides = alien["rect"].colliderect(ship["rect"]) or \
                laser["rect"].colliderect(alien["rect"])

    if collides == True and explosion["played"] == False:
        explosion["sound"].play()
        explosion["played"] = True
        pygame.time.delay(100)

    surface.blit(alien["image"], alien["rect"])
    surface.blit(ship["image"], ship["rect"])

```

```
if laser["alive"]:
    pygame.draw.rect(surface, RED, laser["rect"])
pygame.display.flip()

# Display message is there is a collision and then break
if collides == True:
    if alien["rect"].colliderect(ship["rect"]):
        message = "The alien had you for lunch"
        score = 0
    else:
        message = "You saved the world!"
        score = 500 - alien["rect"].y

# Draw the score
WHITE = (255, 255, 255)
font = pygame.font.Font(None, SCORE_HEIGHT)
image = font.render("Score: " + str(score), 1, WHITE)
rect = image.get_rect()
surface.blit(image, rect)

# Draw a message
font = pygame.font.Font(None, 48)
image = font.render(message, 1, WHITE)
rect = image.get_rect()
surface_rect = rect.move((WIDTH - rect.w)/2, (HEIGHT - rect.h)/2)
image_rect = pygame.Rect(0, 0, 0, rect.h)

while 1:
    if image_rect.w < rect.w:
        image_rect.w = image_rect.w + 1
        surface.blit(image, surface_rect, image_rect)
    else:
        break
    pygame.display.flip()
    pygame.time.delay(10)
pygame.time.delay(3000)
break
```

Breaking up Our Galaxian into Functions

Now we'll break up chunks of code into functions and call these functions. Before we do that, I will review and give you more information about functions.

You know from our heavily-used function `move`, that a function accepts value/values, do some computation, and pass some value/values back.

Here's an example:

```
def f(a, b):  
    print "in f ... ", a, b  
    sum = a + b  
    print "returning", sum  
    return sum  
  
x = 2  
print f(x, 3)
```

You can create a function that does not accept any value:

```
def fortytwo():  
    return 42  
  
print fortytwo()
```

and a function that does not return a value:

```
def shoutfortytwo():  
    print "FORTY-TWO!!!"  
    return  
  
shoutfortytwo()
```

But the point is that the main program calls a function, passing and receiving stuff from the function. Now look at the first example:

```
def f(a, b):  
    print "in f ... ", a, b  
    sum = a + b  
    print "returning", sum  
    return sum
```

```
x = 2  
print f(x, 3)
```

The function `f` has two inputs and one output. Look at the two inputs of `f`: i.e. the two variables `a` and `b`. The main program “plugs” into `f` by passing in `x` and `3`. You should view the calling and returning of `f` as a communication between the main program and `f`.

Different programming languages implement variables in slightly different ways, and the differences can be very subtle especially when variables communicate with each other through a function call.

For the most part, all variables are the same. However there are times where a Python variable is different from a variable in math or a variable from another programming language. This happens most frequently when variables are passed into functions and methods.

So I'll talk a little bit more about variables below.

OK ... so what do we need to talk about? Variables and functions.

There's one last thing of GREAT importance: scope. This means “where can I call on the name of a variable?”. It's also possible to have more than one variable with the same name! The reason why Python won't be confused (but you might!) is because Python create different “areas” for variable names. So it's possible to have an `x` in an “area” and an `x` in another “area”. All you need to know is which “area” to look at for `x`. Instead of talking about theoretical rules, I'll just give you enough examples to play with until you get it.

header

Variables: id and type

Run this program:

```
x = 5
print x
x = x + 1
print x
```

In the main part of the program, `x = 5` will create a variable `x` and make it point to a box with a value of 5.



When it comes to programming, it's critical to understand that the above picture has **two** things: the **name** (i.e. `x`) of the variable and the **value** (i.e. 5) the variable is referring to. (Doesn't this remind you of the dictionary? ...)

Each “box” for a value actually has an id. You can think of it like a PO Box number. Here's how you get the id of the box `x` is referring to:

```
x = 5
print x, id(x)
x = x + 1
print x
```

Run the program.

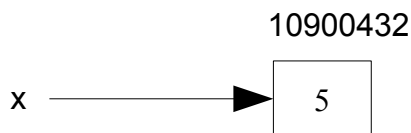
You will notice that the first line of output now gives you the id of `x`. It's extremely important to remember that the id of `x` really refers to the id of the **value** that `x` is **referring** to. **REMEMBER THAT!!!**

When I run the program on my machine I get this output:

```
5 10900432
6
```

The `id` function is extremely important in debugging programs. In fact we are now into debugging Python's memory and not just the high level logic of the program. Note that the `id` refers a place in my computer's memory. Therefore your `id` might be different.

This means that the `id` of `x` (again ... when I say the `id` of `x`, I really mean the `id` of the value `x` is referring to!!!) is 10900432. As you can see the `id` is always an integer. So I can draw this picture for `x`:



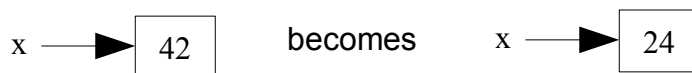
where I put the `id` of the box `x` is referring to above the box. Now that there are **two** values at the box, I need to remind you that in your program `x` refers to the value in the box and not the `id` of the box. Remember that: read the previous statement several times!!!

Do you recall in a previous set of worksheets I said that integers are immutable? At that time I said you cannot change the value of an immutable value. I also said that it seems puzzling since the following statements:

```
x = 42
print x
x = 24
print x
```

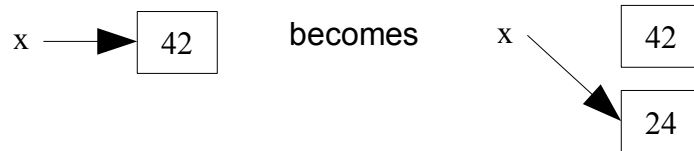
does seem to change the value of `x`. I explained that actually the value `x` is referring to is NOT changed. Rather, a

completely new value is created. In other words the following pictorial depiction of Python's memory is **INCORRECT**:



I say it again: **THE ABOVE IS INCORRECT!**

The **correct** picture is this:



Now we have a way of verifying my claim. Run this:

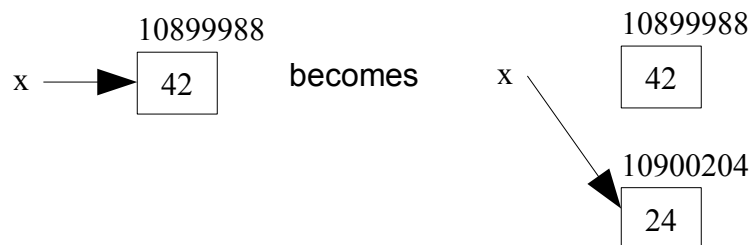
```
x = 42
print x, id(x)
x = 24
print x, id(x)
```

Run it.

Here's my output:

```
42 10899988
24 10900204
```

(Remember: You id's might be different from mine.) Note in particular that the id's **are** in fact different. So if I include the id's of the boxes in my drawing, I would have this:



Exercise. On the other hand, I said that lists are mutable. In other words you can change their values. This means that no new “box” is created when you modify the value of a list. Run this and interpret the result:

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

But try this:

```
a = [1,2,3]
print id(a)
a = [4,5,6]
print id(a)
```

Exercise. Try this:

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

This explain why you have the following:

```
a = [1, 2, 3]
b = a
b[0] = 42
print a, b
```

(I've already mentioned this).

Since we're talking about values, I might as well give you another useful piece of information. We have seen several types of values: integers, floating point numbers, boolean, strings, lists, etc. You can actually get the **type** of value.

Try this:

```
print type(1)
print type(1.2)
print type(True)
print type("hello world")
print [1, 2, "b"]
```

And you can do that for variables too:

```
x = 1
print type(x)
```

Remember that the type of a variable really refers to the type of the **value** the variable is **referring** to.

JARGON

Python is a highly **introspective** (or type-introspective) language. You can analyze the type of all the names.

OK. Enough of details about variables: their id's and types.

Variable: Scope

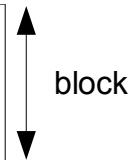
Besides having a name and a value, a variable has

scope. The scope of a variable refers to the “place where you can refer to that variable's name”. For instance the following is silly:

```
print x # err .... x not created yet ...
x = 1
print x # this is OK
print "tada!"
```

In terms of a picture, you can think of the above program as a block with 4 statements:

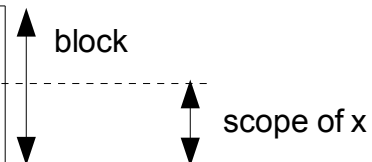
```
print x # err .... x not created yet ...
x = 1
print x # this is OK
print "tada!"
```



block

and the scope of `x`, in other words the place where you can refer to `x`, is within the block where `x` is created **after** it's point of declaration:

```
print x # err .... x not created yet ...
x = 1
print x # this is OK
print "tada!"
```



block

scope of x

Exercise. Of course a function has a name too, just like a variable. Try this:

```
f()

def f():
    print "in f ..."
```

What's the problem? (Duh ...)

But the plot thickens ... because Python code can have lots of blocks, even blocks within blocks.

If you look at a piece of Python code, you will see blocks. Don't forget that a block is just a chunk of code with the same indentation. The following program inserts strings of

length at least 3 into a list:

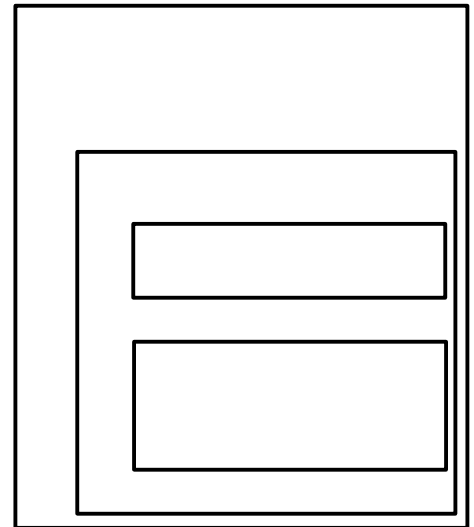
```
xs = []

x = raw_input("Enter space to end: ")
while x != "":
    print "you entered", x
    if len(x) < 3:
        print "your string is too short"
        print "the length must be >= 3"
    else:
        print "putting", x, "into list ..."
        xs.append(x)
        print "... done!"
        print "Here's the new list: ", xs
    x = raw_input("Enter space to end: ")
```

As you write more and more programs you find that blocks are visual guides for reading programs quickly. For the above program you should see the following block structure:

```
xs = []

x = raw_input("Enter space to end: ")
while x != "":
    print "you entered", x
    if len(x) < 3:
        print "your string is too short"
        print "the length must be >= 3"
    else:
        print "putting", x, "into list ..."
        xs.append(x)
        print "... done!"
        print "Here's the new list: ", xs
    x = raw_input("Enter space to end: ")
```



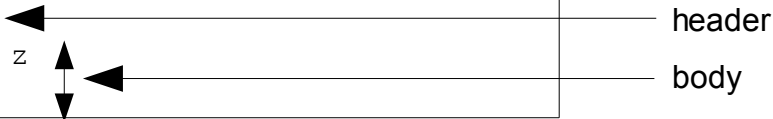
See that?

As you will see in the next section, not all blocks are created equal ...

Functions

Note that a function is like your if statement or the while statement: There is a (function) header and a (function) body:

```
def f(x, y, z):  
    a = x + y + z  
    print a
```



The following two examples show you a curious fact. I have a block of two statements that I want to execute:

```
a = 5  
print a
```

In the first example, I put this block inside an if statement:

```
a = 0  
  
if True:  
    a = 5  
    print a  
  
print a
```

(the condition is always `True` so the block will definitely execute).

In the second example, the block is placed into a function. I have to call it to execute the body.

```
a = 0  
  
def f():  
    a = 5  
    print a  
  
f()  
print a
```

In both cases, I create a variable `a` as the first line of the program and set it to 0. Both blocks attempt to set `a` to 5 and verifies that `a` is set to 5 by printing it.

Once out of the block, I print `a` again.

Go ahead and run both programs. Make sure you see the

difference.

The point: For the if block, the `a` refers to the `a` outside the block. But for the function, the `a` does NOT refer to the `a` outside the function; there are actually two different `a`'s!!!

But what if you really want the `a` in your `f` to refer to the outside `a`? In other words you don't want to create a new `a`.

Try this:

```
a = 0

def f():
    global a
    a = 5
    print a

f()
print a
```

Advice: minimize the use of `global`.

Run this program and study the output carefully:

```
y = 0
print id(y), y

def f(y):
    print id(y), y
    y = 6
    return

f(42)
print id(y), y
```

This program tells you that the `y` in “`def f(y):`” has nothing to do with the `y` outside the function.

The `y` in function `f` is called a **local variable** – it exists only when you call `f`.

Now first look at this function (don't run it yet!):

```
y = 0
print id(y), y
```

```
def f(y):  
    print id(y), y  
    y = 6  
    return  
  
f(y)  
print id(y), y
```

You call `f(y)` and while in the body of `f`, you see that `y` is set to 6. When you return and print `y` what do you see? Do you get 6? Run the program and check!

The above program tells you that ... **the `y` inside `f` is not the same as the `y` outside `f`.**

Now look at this version:

```
y = 0  
print id(y), y  
  
def f():  
    print id(y), y  
    y = 6  
    return  
  
f()  
print id(y), y
```

You will get an error.

Here's another example. This is similar to the above but with one line removed:

```
y = 0  
print id(y), y  
  
def f():  
    print id(y), y  
    return  
  
f()  
print id(y), y
```

Study the output carefully!!!

Simple Galaxian: Functions for Creating Things

Back to our Galaxian ...

We have a variable `alien` representing a higher-level concept that bundles up lower level concepts like `alien_image`, `alien_rect`. Remember?

It's common to create functions that create such high level variables. Let me do one for you. You have the following code in your simple Galaxian:

```
...
alien = {}
alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
alien["rect"] = alien["image"].get_rect()
alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
alien["speed"] = [1, 0]
...
```

(I'm only showing the relevant part).

Create the following function that creates a dictionary and then return it like this:

```
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien
```

I'm going to put my function near the top of the program:

```
import pygame, sys, random
pygame.init()
random.seed()

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
```



```

        return alien

ALIEN_Y_INCREMENT = 10
...

```

(Again I'm only showing the relevant part of the program.)

Of course the main program does NOT have an `alien` dictionary. I still need to create an `alien` in the main program by calling the `get_alien` function.

```

import pygame, sys, random
pygame.init()
random.seed()

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10

...

alien = get_alien()

collides = False
while 1:
    ...

```

Exercise. Create one function for each of the following:

- `get_ship` for creating and returning a dictionary that represents a ship
- `get_laser` for creating and returning a dictionary that represents a laser
- `get_explosion` for creating and returning a dictionary that represents an explosion

Use these function to create a ship, laser and explosion variable. Your code should look like this:

```

import pygame, sys, random
pygame.init()
random.seed()

```

```
# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10

# Create flagship
def get_ship():
    ... FILL IN THE BLANK ...

# Ship's laser
def get_laser():
    ... FILL IN THE BLANK ...

# Explosion sound
def get_explosion():
    ... FILL IN THE BLANK ...

def move(d, v, m):
    d = d + v
    if d < 0:
        d = 0
        v = -v
    elif d > m:
        d = m
        v = -v
    return d, v

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

pygame.key.set_repeat(10, 10)

SCORE_HEIGHT = 24
BLACK = (0, 0, 0)
RED = (255, 0, 0)
sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")
```

```
alien = get_alien()
... FILL IN THE BLANK ...

collides = False
while 1:
    ... as before ...
```

Solution

```
import pygame, sys, random
pygame.init()
random.seed()

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10

# Create flagship
def get_ship():
    ship = {}
    ship["image"] = pygame.image.load("GalaxianGalaxip.gif")
    ship["rect"] = ship["image"].get_rect()
    x = (WIDTH - ship["rect"].w) / 2
    y = HEIGHT - ship["rect"].h
    ship["rect"] = ship["rect"].move([x, y])
    ship["speed"] = [0, 0]
    return ship

# Ship's laser
def get_laser():
    laser = {}
    laser["rect"] = pygame.Rect(0, 0, 4, 8)
    laser["speed"] = [0, 0]
    laser["alive"] = False
    laser["sound"] = pygame.mixer.Sound("laser.wav")
    return laser

# Explosion sound
def get_explosion():
    explosion = {}
    explosion["sound"] = pygame.mixer.Sound("gexplode.wav")
    explosion["played"] = False
    return explosion

def move(d, v, m):
    d = d + v
    if d < 0:
        d = 0
        v = -v
    elif d > m:
        d = m
        v = -v
    return d, v
```

```

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
surface = pygame.display.set_mode(SIZE)

pygame.key.set_repeat(10, 10)

SCORE_HEIGHT = 24
BLACK = (0, 0, 0)
RED = (255, 0, 0)
sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

alien = get_alien()
ship = get_ship()
laser = get_laser()
explosion = get_explosion()

collides = False
while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            keypress = pygame.key.get_pressed()
            if keypress[pygame.K_LEFT]:
                ship["speed"] = [-1, 0]
            if keypress[pygame.K_RIGHT]:
                ship["speed"] = [1, 0]
            if keypress[pygame.K_SPACE]:
                if not laser["alive"]:
                    laser["alive"] = True
                    laser["speed"] = [0, -2]
                    laser["rect"].x = ship["rect"].x + (ship["rect"].w - \
                                                            laser["rect"].w)/2
                    laser["rect"].y = ship["rect"].y - laser["rect"].w
                    laser["sound"].play()

    surface.fill(BLACK)

    if not collides:
        alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                                    alien["speed"][0], WIDTH - alien["rect"].w)
        if random.randrange(100) == 0:
            alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT

        ship["rect"].x, ship["speed"][0] = move(ship["rect"].x, \
                                                    ship["speed"][0], WIDTH - ship["rect"].w)
        ship["speed"] = [0, 0]

        if laser["alive"]:
            laser["rect"].y, laser["speed"][1] = move(laser["rect"].y, \
                                                        laser["speed"][1], HEIGHT - laser["rect"].h)

```

```
        if laser["rect"].y < SCORE_HEIGHT:
            laser["alive"] = False

    collides = alien["rect"].colliderect(ship["rect"]) or \
        laser["rect"].colliderect(alien["rect"])

    if collides == True and explosion["played"] == False:
        explosion["sound"].play()
        explosion["played"] = True
        pygame.time.delay(100)

    surface.blit(alien["image"], alien["rect"])
    surface.blit(ship["image"], ship["rect"])
    if laser["alive"]:
        pygame.draw.rect(surface, RED, laser["rect"])
    pygame.display.flip()

    # Display message is there is a collision and then break
    if collides == True:
        if alien["rect"].colliderect(ship["rect"]):
            message = "The alien had you for lunch"
            score = 0
        else:
            message = "You saved the world!"
            score = 500 - alien["rect"].y

    # Draw the score
    WHITE = (255, 255, 255)
    font = pygame.font.Font(None, SCORE_HEIGHT)
    image = font.render("Score: " + str(score), 1, WHITE)
    rect = image.get_rect()
    surface.blit(image, rect)

    # Draw a message
    font = pygame.font.Font(None, 48)
    image = font.render(message, 1, WHITE)
    rect = image.get_rect()
    surface_rect = rect.move((WIDTH - rect.w)/2, (HEIGHT - rect.h)/2)
    image_rect = pygame.Rect(0, 0, 0, rect.h)

    while 1:
        if image_rect.w < rect.w:
            image_rect.w = image_rect.w + 1
            surface.blit(image, surface_rect, image_rect)
        else:
            break
        pygame.display.flip()
        pygame.time.delay(10)
    pygame.time.delay(3000)
    break
```

Simple Galaxian: Modules

Now that we have a function for creating things, we can for instance use the same function to create **two** aliens very easily. For instance in the code we can have

```
...
alien1 = get_alien()
alien2 = get_alien()
...
```

Of course their rects are the same so that they are on top of each other. So we need to move, for instance, the alien2["rect"] away. I'll leave that to you to play around with modifying the game to have two aliens.

But right now, I will show you how to simplify your program using modules.

Remember that a module is more or less a program file. I'm going to create an Alien.py file. I'm going to put everything related to the alien in this file. So the file looks like this:

```
# file: Alien.py

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10
```

Now note this ... the program file Alien.py uses pygame. You did import (and initialize) pygame in your simple galaxian program file. But Alien.py does not have access to that. You need to import and initialize pygame in Alien.py. So I add this to Alien.py:

```
# file: Alien.py

import pygame
```

```
pygame.init()

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10
```

Note that I'm not importing sys or random since Alien.py does not use them.

Now the program file for the simple Galaxian should look like this:

```
import pygame, sys, random
pygame.init()
random.seed()

import Alien

... THE FUNCTION get_alien IS REMOVED ...

# Create flagship
def get_ship():
    ... as before ...

... as before ...

alien = Alien.get_alien()

...

collides = False
while 1:

    ... as before ...

    if random.randrange(100) == 0:
        alien["rect"].y = alien["rect"].y + Alien.ALIEN_Y_INCREMENT

    ... as before ...
```

Notice that I've imported Alien. Also any time I used the names in Alien.py I have to change them. For instance get_alien is now Alien.get_alien and

ALIEN_Y_INCREMENT is now changed to
Alien.ALIEN_Y_INCREMENT.

If you run the program , you will get an error. Looking at
stderr.txt you see the problem:

```
import pygame
pygame.init()

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10
```

Alien.py uses SCORE_HEIGHT. Now if you look at your
simple Galaxian program, you see that SCORE_HEIGHT is
used in quite a few places. That's not surprising if you think
about it. It concerns the playing area of the game. The
variable is a constant variable: We don't change the value of
the variable. In writing large programs, it's common to have a
file of constants. So let's do that. Here's the file which I'm
calling CONSTANTS.py:

```
# file: CONSTANTS.py

SCORE_HEIGHT = 24
```

And here is the new Alien.py.

```
import pygame
pygame.init()

import CONSTANTS

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, CONSTANTS.SCORE_HEIGHT])
```

```
alien["speed"] = [1, 0]
return alien
```

```
ALIEN_Y_INCREMENT = 10
```

Of course I need to import `CONSTANTS`. Furthermore

`SCORE_HEIGHT` is now called `CONSTANTS.SCORE_HEIGHT`.

And finally here's your simple Galaxian program:

```
import pygame, sys, random
pygame.init()
random.seed()

import CONSTANTS
import Alien

# Create flagship
def get_ship():
    ... as before ...

... as before ...

pygame.key.set_repeat(10, 10)

# SCORE_HEIGHT is removed ...
BLACK = (0, 0, 0)

... as before ...

collides = False
while 1:

    ... as before ...

    if not collides:
        ... as before ...
        if laser["alive"]:
            if laser["rect"].y < CONSTANTS.SCORE_HEIGHT:
                laser["alive"] = False

        ... as before ...

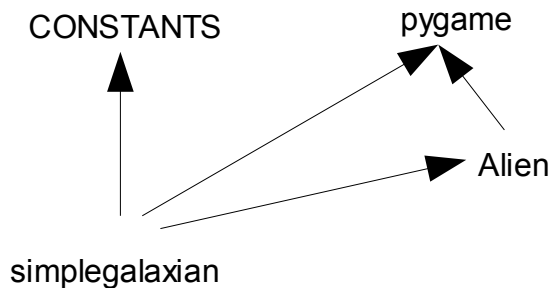
        # Draw the score
        WHITE = (255, 255, 255)
        font = pygame.font.Font(None, CONSTANTS.SCORE_HEIGHT)

        ... as before ...
```

This is easy: Again I import `CONSTANTS` and replace `SCORE_HEIGHT` with `CONSTANTS.SCORE_HEIGHT`.

Now run the program ... and voila ... it works. Sometimes it

helps to keep track of module dependencies. We can draw this picture:



Here's a new syntax for you: Sometimes we are too lazy to write the module name before a name. For instance in our simple Galaxian program, we refer to `SCORE_HEIGHT` in `CONSTANTS` as `CONSTANTS.SCORE_HEIGHT`. If you want to refer to a name in a module without typing the module's name you can do a different kind of import:

```
...
from CONSTANTS import SCORE_HEIGHT
...
```

After this special import you can use `SCORE_HEIGHT` instead of `CONSTANTS.SCORE_HEIGHT`.

If you want to import all the names in `CONSTANTS`, you do this:

```
...
from CONSTANTS import *
...
```

You should not overuse “from ... import ...” because it can lead to bugs which are difficult to find. I'll leave it to you to decide if you want to use this import for your `CONSTANTS` module. Using “from ... import ...” is appropriate for a module of constants.

Exercise. Move other constants into `CONSTANTS.py`:

- `WIDTH`
- `HEIGHT`
- `SIZE`
- `RED`
- `BLACK`

Modify all your files accordingly to make your game work.

Don't move on until you're done with this part!

Exercise. Create a Ship, Laser, Explosion modules (use the Alien module as an example.) The Ship module should contains the `get_ship` function, etc. ADVICE: Work on the Ship module first and make sure it works – destroy the alien, move your ship, let the alien get your ship – before creating the Laser module. Remember to take baby steps when writing programs. (Actually this applies to any problem solving activity!)

Solution

```
# file: CONSTANTS.py

WIDTH, HEIGHT = 640, 480
SIZE = (WIDTH, HEIGHT)
BLACK = (0, 0, 0)
RED = (255, 0, 0)
SCORE_HEIGHT = 24
```

```
# file: Alien.py

import pygame
pygame.init()

from CONSTANTS import *

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10
```

```
# file: Ship.py

import pygame
from CONSTANTS import *

# Create flagship
def get_ship():
    ship = {}
    ship["image"] = pygame.image.load("GalaxianGalaxip.gif")
    ship["rect"] = ship["image"].get_rect()
    x = (WIDTH - ship["rect"].w) / 2
    y = HEIGHT - ship["rect"].h
    ship["rect"] = ship["rect"].move([x, y])
    ship["speed"] = [0, 0]
    return ship
```

```
# file: Laser.py

import pygame
pygame.init()

from CONSTANTS import *

# Ship's laser
def get_laser():
    laser = {}
    laser["rect"] = pygame.Rect(0, 0, 4, 8)
    laser["speed"] = [0, 0]
    laser["alive"] = False
    laser["sound"] = pygame.mixer.Sound("laser.wav")
    return laser
```

```
# file: Explosion.py

import pygame
pygame.init()

from CONSTANTS import *

# Explosion sound
def get_explosion():
    explosion = {}
    explosion["sound"] = pygame.mixer.Sound("gexplode.wav")
    explosion["played"] = False
    return explosion
```

```
# file: simplygalaxian.py

import pygame, sys, random
pygame.init()
random.seed()

from CONSTANTS import *
import Alien, Ship, Laser, Explosion

def move(d, v, m):
    d = d + v
    if d < 0:
        d = 0
        v = -v
    elif d > m:
        d = m
```

```

        v = -v
    return d, v

surface = pygame.display.set_mode(SIZE)

pygame.key.set_repeat(10, 10)

sys.stdout = file("stdout.txt", "w")
sys.stderr = file("stderr.txt", "w")

alien = Alien.get_alien()
ship = Ship.get_ship()
laser = Laser.get_laser()
explosion = Explosion.get_explosion()

collides = False
while 1:

    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            sys.exit()
        elif event.type == pygame.KEYDOWN:
            keypress = pygame.key.get_pressed()
            if keypress[pygame.K_LEFT]:
                ship["speed"] = [-1, 0]
            if keypress[pygame.K_RIGHT]:
                ship["speed"] = [1, 0]
            if keypress[pygame.K_SPACE]:
                if not laser["alive"]:
                    laser["alive"] = True
                    laser["speed"] = [0, -2]
                    laser["rect"].x = ship["rect"].x + (ship["rect"].w - \
                        laser["rect"].w)/2
                    laser["rect"].y = ship["rect"].y - laser["rect"].w
                    laser["sound"].play()

    surface.fill(BLACK)

    if not collides:
        alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
            alien["speed"][0], WIDTH - alien["rect"].w)
        if random.randrange(100) == 0:
            alien["rect"].y = alien["rect"].y + Alien.ALIEN_Y_INCREMENT

        ship["rect"].x, ship["speed"][0] = move(ship["rect"].x, \
            ship["speed"][0], WIDTH - ship["rect"].w)
        ship["speed"] = [0, 0]

        if laser["alive"]:
            laser["rect"].y, laser["speed"][1] = move(laser["rect"].y, \
                laser["speed"][1], HEIGHT - laser["rect"].h)
            if laser["rect"].y < SCORE_HEIGHT:
                laser["alive"] = False

```

```

collides = alien["rect"].colliderect(ship["rect"]) or \
    laser["rect"].colliderect(alien["rect"])

if collides == True and explosion["played"] == False:
    explosion["sound"].play()
    explosion["played"] = True
    pygame.time.delay(100)

surface.blit(alien["image"], alien["rect"])
surface.blit(ship["image"], ship["rect"])
if laser["alive"]:
    pygame.draw.rect(surface, RED, laser["rect"])
pygame.display.flip()

# Display message is there is a collision and then break
if collides == True:
    if alien["rect"].colliderect(ship["rect"]):
        message = "The alien had you for lunch"
        score = 0
    else:
        message = "You saved the world!"
        score = 500 - alien["rect"].y

# Draw the score
WHITE = (255, 255, 255)
font = pygame.font.Font(None, SCORE_HEIGHT)
image = font.render("Score: " + str(score), 1, WHITE)
rect = image.get_rect()
surface.blit(image, rect)

# Draw a message
font = pygame.font.Font(None, 48)
image = font.render(message, 1, WHITE)
rect = image.get_rect()
surface_rect = rect.move((WIDTH - rect.w)/2, (HEIGHT - rect.h)/2)
image_rect = pygame.Rect(0, 0, 0, rect.h)

while 1:
    if image_rect.w < rect.w:
        image_rect.w = image_rect.w + 1
        surface.blit(image, surface_rect, image_rect)
    else:
        break
    pygame.display.flip()
    pygame.time.delay(10)
    pygame.time.delay(3000)
    break

```

(I'm giving the whole `simplegalaxian.py` but really ... most of it remains unchanged.)

The Goal in Module Decomposition

The above way of placing functions into different modules is not arbitrary.

Basically you want to do this: Look at all the “thinks” your program involves. For each type of thing, create a module. Put all the functions “most associated” with a type of thing into its module.

For instance we have a ship thingy. So we have a `Ship` module. We have a `get_ship` function which is of course about the ship. So it's natural that we put this function into the `Ship` module.

With this organization, if we have a function that looks like it's associated with an alien, we immediately go to the `Alien` module for information, whether it is adding some new feature to the alien or debugging an error that might be related to the alien. It's really like categorizing books in a library for instance.

That's all there is to it. (Well ... it's not that simple.)

Let's look for code associated with the alien and try to create a function for that piece of code and place the function in the `Alien` module.

Now locate this piece of code in your `simplegalaxian.py`:

```
alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                         alien["speed"][0], WIDTH - alien["rect"].w)
if random.randrange(100) == 0:
    alien["rect"].y = alien["rect"].y + Alien.ALIEN_Y_INCREMENT
```

What does it do? ... at a high level? i.e. at the level of the alien itself? It moves the alien. Right? Oui?

I'm doing to create a function called `move_alien` and pass in the `alien` dictionary. Why pass in the `alien` dictionary? Look at the code:

```
alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                         alien["speed"][0], WIDTH - alien["rect"].w)
if random.randrange(100) == 0:
    alien["rect"].y = alien["rect"].y + Alien.ALIEN_Y_INCREMENT
```

What do you need to execute this code? Basically:

- `alien["rect"].x`
- `alien["speed"][0]`
- `move`
- `WIDTH`
- `alien["rect"].w`
- `random.randrange`
- `alien["rect"].y`
- `Alien.ALIEN_Y_INCREMENT`

Now `alien["rect"].x`, `alien["speed"][0]`, `alien["rect"].w`, `alien["rect"].y` can all be found in `alien`. So The code depends on:

- `alien`
- `move`
- `WIDTH`
- `random.randrange`
- `Alien.ALIEN_Y_INCREMENT`

If we import `random` in `Alien`, then we can use `random.randrange`. So that's not a problem. The constant `Alien.ALIEN_Y_INCREMENT` is in fact in `Alien.py`. So again this is fine. To have access to `WIDTH`, we need to import `CONSTANTS`. The only problem is the `move` function. We need to use it. It's kept in `simplegalaxian.py`. `simplegalaxian.py` imports `Alien.py`. We cannot get `Alien.py` to import `simplegalaxian.py` otherwise we get into a circular reference. In other words you cannot have `A.py` import `B.py` and `B.py` import `A.py`.

Should we move that function into `Alien.py`?

Well ... a quick look at your program tells you that the `move` function is used by many “things” in your program. So it's better to put the `move` function into a module that the `Alien`, `Ship`, `Laser` modules can use. (Either that we have to copy and paste the `move` function into all of them – a very bad idea. Remember we are smart and do not want to do duplicate work. Duplication of code is BADDDDDDDDD!!!!) `CONSTANTS.py` is in fact imported by all of them. Should we move the `move` function into `CONSTANT.py`?

NO!!!

Because it's a function and not a constant. Doing so would

be misleading and confusing ourselves in the future.

Let's just create a new module. Let's just call the module `util.py` – it's a module of utility functions. (Admittedly this is kind of too general. Since it is related to motion, we can call it `physics.py`. But I'll just stick to `util.py`.)

Exercise. Create a `util.py` and move the `move` function out of `simplegalaxian.py` into this new file. Modify `simplegalaxian.py` to make it work.

Now let's get back to our `move_alien` function in `Alien.py`. Recall that we want the function to perform this piece of code from our `simplegalaxian.py`:

```
alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                         alien["speed"][0], WIDTH - alien["rect"].w)
if random.randrange(100) == 0:
    alien["rect"].y = alien["rect"].y + Alien.ALIEN_Y_INCREMENT
```

Remember that the function must accept the `alien` dictionary, modify it (using the `move` function in `util.py`) and then return the `alien`. So if `move_alien` is written correctly, the above code in `simplegalaxian.py` can be replaced by:

```
alien = Alien.move_alien(alien)
```

Note one thing about the change we're making to `simplegalaxian.py` (THIS IS IMPORTANT!!!): We're trying to “think” about alien, ship, laser, explosion in our `simplegalaxian.py`. Lower level details such as `alien["image"]`, `alien["rect"]` is pushed to `Alien.py`, `Ship.py`, `Laser.py`, `Explosion.py`.

Do you see the hierarchy of concepts and corresponding hierarchy of modules we are building?

Exercise. Finish the `util.py` and `move_alien` function in `Alien.py`. Modify `simplegalaxian.py` and make sure it works.

Solution

There is no change to CONSTANTS.py, Ship.py, Laser.py, and Explosion.py. So I'll just show the these file:

```
# file: util.py

def move(d, v, m):
    d = d + v
    if d < 0:
        d = 0
        v = -v
    elif d > m:
        d = m
        v = -v
    return d, v
```

```
import pygame, random
pygame.init()
random.seed()

from CONSTANTS import *
from util import *

# Create alien
def get_alien():
    ... as before ...

ALIEN_Y_INCREMENT = 10

def move_alien(alien):
    alien["rect"].x, alien["speed"][0] = move(alien["rect"].x, \
                                              alien["speed"][0], WIDTH - alien["rect"].w)

    if random.randrange(100) == 0:
        alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT
    return alien
```

```
# file: simplygalaxian.py

... as before ...
from util import *
import Alien, Ship, Laser, Explosion

... as before ...

while 1:
    ... as before ...
    if not collides:
        alien = Alien.move_alien(alien)

    ... as before ...
```

Simple Galaxian: Other Move Functions

Exercise. Look at this code segments in `simplegalaxian.py`:

```
ship["rect"].x, ship["speed"][0] = move(ship["rect"].x, ship["speed"][0], \
                                         WIDTH - ship["rect"].w)
ship["speed"] = [0, 0]
```

And

```
if laser["alive"]:
    laser["rect"].y, laser["speed"][1] = move(laser["rect"].y, laser["speed"][1], \
                                              HEIGHT - laser["rect"].h)
    if laser["rect"].y < SCORE_HEIGHT:
        laser["alive"] = False
```

Convert the first piece of code to `move_ship` in the `Ship` module and the second piece of code to `move_laser` in the `Laser` module. Modify `simplegalaxian.py` so that you have the following code segment:

```
if not collides:
    alien = Alien.move_alien(alien)
    ship = Ship.move_ship(ship)
    laser = Laser.move_laser(ship)
```

Solution

Here are the modified files:

```
# file: Ship.py

from util import *

... as before ...

def move_ship(ship):
    ship["rect"].x, ship["speed"][0] = move(ship["rect"].x, ship["speed"][0], \
                                             WIDTH - ship["rect"].w)

    ship["speed"] = [0, 0]
    return ship
```

```
# file: Laser.py

from util import *

... as before ...

ALIEN_Y_INCREMENT = 10

def move_laser(laser):
    if laser["alive"]:
        laser["rect"].y, laser["speed"][1] = move(laser["rect"].y, \
                                                    laser["speed"][1], HEIGHT - laser["rect"].h)

        if laser["rect"].y < SCORE_HEIGHT:
            laser["alive"] = False
    return laser
```

```
# file: simplygalaxian.py
... as before ...

collides = False
while 1:

    ... as before ...

    surface.fill(BLACK)

    if not collides:
        alien = Alien.move_alien(alien)
        ship = Ship.move_ship(ship)
        laser = Laser.move_laser(laser)

... as before ...
```

Four move Functions???

Now I'm going to do something that seem mysterious and redundant at first. But it's important because the type of program decomposition (into modules) that we're doing is actually related to object-oriented decomposition of a problem. I won't explain why I want to do this but this will be clear some day when we talk about object-oriented programming.

I want to have four different move functions. There's already a move function in util.py. Correct?

I want you to modify Alien.py as follows:

```
import pygame, random
pygame.init()
random.seed()

from CONSTANTS import *
import util

# Create alien
def get_alien():
    alien = {}
    alien["image"] = pygame.image.load("GalaxianAquaAlien.gif")
    alien["rect"] = alien["image"].get_rect()
    alien["rect"] = alien["rect"].move([0, SCORE_HEIGHT])
    alien["speed"] = [1, 0]
    return alien

ALIEN_Y_INCREMENT = 10

def move(alien):
    alien["rect"].x, alien["speed"][0] = util.move(alien["rect"].x,
                                                    alien["speed"][0], WIDTH - alien["rect"].w)

    if random.randrange(100) == 0:
        alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT
    return alien
```

I'm changing `move_alien` to `move`. Since there are now two moves here (one in Alien.py and one in the util.py), I need to make sure that I'm not doing "from util import *". I have to do "import util" so that the `move` function in util.py is `util.move` otherwise the `move` function in util becomes `move` in Alien.py and Python will do the wrong thing. Make sure you see this point. Ask questions if you need to!

Now that the `move_alien` function is change to `move`, I need

to update my `simplegalaxian.py` as follows. Look for this piece of code in `simplegalaxian.py`:

```
if not collides:
    alien = Alien.move_alien(alien)
    ship = Ship.move_ship(ship)
    laser = Laser.move_laser(laser)
```

and change it to:

```
if not collides:
    alien = Alien.move(alien)
    ship = Ship.move_ship(ship)
    laser = Laser.move_laser(laser)
```

Run your program and make sure it works correctly.

Exercise. Modify `Ship.py` and `Laser.py` so that the above code segment in `simplegalaxian.py` becomes:

```
if not collides:
    alien = Alien.move(alien)
    ship = Ship.move(ship)
    laser = Laser.move(laser)
```

I won't bother giving you the modifications to `Ship.py` and `Laser.py` since they are pretty similar to the changes for `Alien.py`.

Solution

Here are the modified files:

Simple Galaxian: blit functions

Now move all the blit code to the respect module. For instance

```
surface.blit(alien["image"], alien["rect"])
```

Call the function `draw`. Note that the above statement requires `surface` and `alien`. After the draw function in `Alien` is written the above code in `simplegalaxian.py` becomes:

```
Alien.draw(surface, alien)
```

Likewise

```
surface.blit(ship["image"], ship["rect"])
```

Should be placed in `Ship.py` as the draw function.

And finally the following code:

```
if laser["alive"]:
    pygame.draw.rect(surface, RED, laser["rect"])
```

Should be placed in the draw function in `Laser.py`.

Once all the three draw functions are written, the following code in `simplegalaxian.py`:

```
surface.blit(alien["image"], alien["rect"])
surface.blit(ship["image"], ship["rect"])
if laser["alive"]:
    pygame.draw.rect(surface, RED, laser["rect"])
```

Is replaced by

```
Alien.draw(surface, alien)
Ship.draw(surface, ship)
Laser.draw(surface, laser)
```

Now in this case, although the statements are shorter and you do not see the keys in the dictionary, the length of the code is only one line shorter than before. But you realize that an alien and a ship and a laser looks very similar – at least in terms of the code you execute on them.

The Big Picture

I will stop here and give you the big picture.

The goal is to see only the dictionaries alien, ship, laser, explosion in simplegalaxian.

Whenever the code goes deeper into the dictionaries to use the key-value pairs, the code should be moved to the appropriate module.

This allows the program to be developed at two different levels. This has (at least!) two benefits. First this allows you to work with less code at one time. Of course when you need to work with the code you have to figure out where the code lies. For instance if you want to add something to the way an alien works, you either look at simplegalaxian and search for alien or you go to Alien.py.

Another benefit is that when a program is made up of pieces, it's easier for a team to work on the program together.

Now a challenge for you is to complete the program decomposition so that simplegalaxian.py does not need to use the keys of the dictionaries. There are also some code that can be packaged up into functions in util.py. Here are some suggestions:

- Write a function wipeacross that displays the text when the game ends. Put the function in util.py
- For ship and laser write functions to process the keypress. Call the functions process_keypress.
- Etc.

Another challenge: Once you're done you will find that your simplegalaxian.py is really simple. But something else strikes you: Some functions in Alien.py, Ship.py, Laser.py, Explosion.py are actually very similar. Even if there are not, you can “massage” some of them into a similar form. For instance look at the move function in Alien.py:

```
def move(alien):
    alien["rect"].x, alien["speed"][0] = util.move(alien["rect"].x, \
                                                    alien["speed"][0], WIDTH - alien["rect"].w)
    if random.randrange(100) == 0:
        alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT
    return alien
```

And the move function in Ship.py

```
def move(ship):
    ship["rect"].x, ship["speed"][0] = util.move(ship["rect"].x, \
                                                  ship["speed"][0], WIDTH - ship["rect"].w)
    ship["speed"] = [0, 0]
    return ship
```

Do you see that the first line of both functions are very similar???

One possibility is to create another level in your program decomposition, something that represents both an alien and a ship. Here's a suggestion: Call the module GameDict.py. In GameDict.py you have a move function:

```
# file: GameDict.py

import util

def move(d):
    d["rect"].x, d["speed"][0] = util.move(d["rect"].x, \
                                           d["speed"][0], WIDTH - d["rect"].w)
    return d
```

With the module you can now rewrite your Alien.py as

```
# file: Alien.py

... as before
import GameDict
... as before ...

def move(alien):
    GameDict.move(alien)
    if random.randrange(100) == 0:
        alien["rect"].y = alien["rect"].y + ALIEN_Y_INCREMENT
    return alien
```

And your Ship.py becomes

```
# file: Ship.py

... as before
import GameDict
... as before ...

def move(ship):
    GameDict.move(ship)
    ship["speed"] = [0, 0]
    return ship
```