INFT1004 - SEMESTER 1 - 2017		LECTURE TOPICS	
Week 1	Feb 27	Introduction, Assignment, Arithmetic	
Week 2	Mar 6	Sequence, Quick Start, Programming Style	
Week 3	Mar 13	Pictures, Functions, Media Paths	
Week 4	Mar 20	Arrays, Pixels, For Loop, Reference Passing	
Week 5	Mar 27	Nested Loops, Selection, Advanced Pictures	~~Practical Test~~
Week 6	Apr 3	Lists, Strings, Input & Output, Files	Practical Test
Week 7	Apr 10	Drawing Pictures, Program Design, While Loop	Assignment set
Recess	Apr 14 – Apr 23	Mid Semester Recess Break	
Week 8	Apr 24	No Lecture / Revision and Assignment in Labs	
Week 9	May 1	Data Structures, Processing sound	
Week 10	May 8	Advanced sound	Assignment part 1 due 8:00am Tue, May 9
Week 11	May 15	Movies, Scope, Import	
Week 12	May 22	Turtles, Writing Classes	Assignment part 2 due 8:00am Tue, May 23
Week 13	May 29	Revision	
Mid Year Examination Period - MUST be available normal & supplementary period			

Lecture Topics and Lab topics are the same for each week

INFT1004 - SEMESTER 1 - 2017 LECTURE TOPICS
Week 1 Feb 27 Introduction, Assignment, Arithmetic
Week 2
Week 3
Week 4
Week 5 Practical Test
Week 6
Week 7 Assignment set
Recess
Week 8
Week 9
Week 10 Assignment part 1 due 8:00am Tue, May 9
Week 11
Week 12 Assignment part 2 due 8:00am Tue, May 23
Week 13 May 29 Revision
Mid Year Examination Period - MUST be available normal & supplementary period
Lecture Topics and Lab topics are the same for each week

Practical Test in Tutorial – will now be in (Week 6)
(we haven't covered enough yet)

Assignment set
(12 April, 8:00am)

INFT1004

Visual Programming

Module 5.1
Nested For and Pictures

Guzdial & Ericson - Third Edition - chapters 4, 5
Guzdial & Ericson - Fourth (Global) Edition – chapters 5, 6

Types of Loops

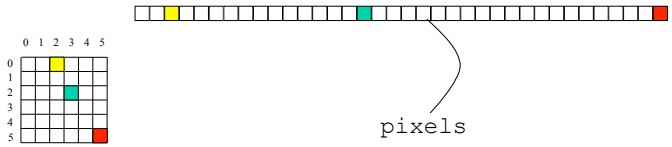
While loop	Tests some condition - If the condition is true it executes some statements Repeats until the test condition is false
For loop	Repeats some statements a predetermined number of times
Nested loop	One loop inside another loop

Revision

getPixels()

The *for* statement loops (iterates) through all the items in a sequence

```
pixels = getPixels(myPic)
```



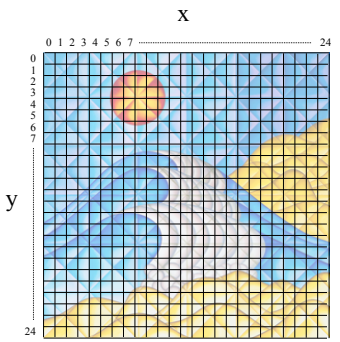
If we use `getPixels()` to get an array of all the pixels in a picture, we can use *for* to do the same thing to each of them in turn

Pixel coordinates

So long as we want to do the same thing to every pixel in a picture, `getPixels()` does a fine job (it returns a 1D array of ALL the pixels)

if we want to change only some of the picture we can use range() to process some pixels

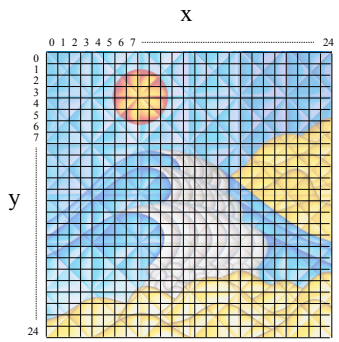
But it's a bit tricky to work out which ones because it is a 1D array and it would be good to work with (x,y)



Pixel coordinates

Because `getPixels(pic)` returns a 1D list it's a bit tricky to work out which ones correspond to which `(x,y)` pixels

Nested for loops work very well in this situation and we can then work with `getPixel(pic, x, y)`

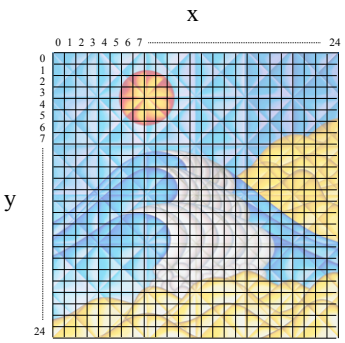


Types of Loops

While loop	Tests some condition - If the condition is true it executes some statements Repeats until the test condition is false
For loop	Repeats some statements a predetermined number of times
Nested loop	One loop inside another loop

Pixel coordinates

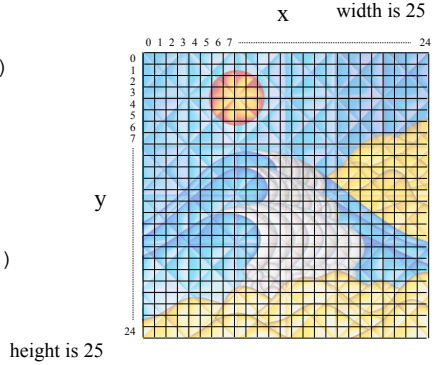
If we want to process just some of the pixels, we need to work with their x and y coordinates



Pixel coordinates

`getWidth(picture)` will give you the width

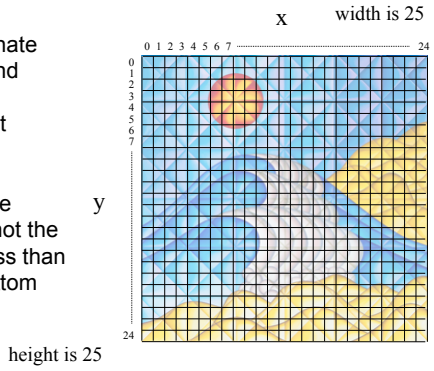
`getHeight(picture)` will give you the height



Pixel coordinates

The x (horizontal) coordinate starts from 0 at the left and goes to 1 less than `getWidth()` at the right

The y (vertical) coordinate starts from 0 at the top (not the bottom) and goes to 1 less than `getHeight()` at the bottom

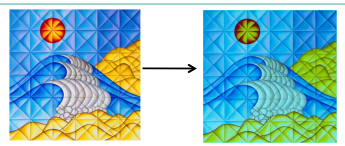


Ranges of pixels

Compare these chunks of code, which achieve exactly the same result:

```
for pixel in getPixels(pic):
  setRed(pixel, getRed(pixel) / 2)
```

```
for x in range(0, getWidth(pic)):
  for y in range(0, getHeight(pic)):
    pixel = getPixel(pic, x, y)
    setRed(pixel, getRed(pixel) / 2)
```



Nested loops

```
for x in range(0, getWidth(pic)):
    for y in range(0, getHeight(pic)):
        pixel = getPixel(pic, x, y)
        setRed(pixel, getRed(pixel) / 2)
```

This example has two nested loops, ie one loop inside another (look at the indentation)

Nested loops

```
for x in range(0, getWidth(pic)):
    for y in range(0, getHeight(pic)):
        pixel = getPixel(pic, x, y)
        setRed(pixel, getRed(pixel) / 2)
```

This example has two nested loops, ie one loop inside another (look at the indentation)

Think how this works:

for each value of the outer loop
x is 0 ... getWidth(pic)-1) (each column of picture)

the program goes through each value of the inner loop
y is 0 ... getHeight(pic)-1 (each row of picture)

Nested loops

```
for x in range(0, getWidth(pic)):
    for y in range(0, getHeight(pic)):
        pixel = getPixel(pic, x, y)
        setRed(pixel, getRed(pixel) / 2)
```

So if the outer loop has a range of 200, and the inner loop has a range of 400 (picture is 200 pixels wide and 400 high)

There will be 80,000 iterations! (200 x 400)

Nested loops

```
for x in range(0, getWidth(pic)):
    for y in range(0, getHeight(pic)):
        pixel = getPixel(pic, x, y)
        setRed(pixel, getRed(pixel) / 2)
```

So if the outer loop has a range of 200, and the inner loop has a range of 400 (picture is 200 pixels wide and 400 high)

there will be 80,000 iterations! (200 x 400)

Nested loops are very common in processing pictures, because pictures have two dimensions

Loop order - example

```
for y in range(0, getHeight(picture)):
    for x in range(0, getWidth(picture)):
       aPixel = getPixel(picture, x, y)
       setColor(aPixel, yellow)
       repaint(picture)
```

```
for x in range(0, getWidth(picture)):
   for y in range(0, getHeight(picture)):
       aPixel = getPixel(picture, x, y)
       setColor(aPixel, yellow)
       repaint(picture)
```

Mod5_1_NestedForPictures.py – testShowLoop(rowOrder)

Or Try this

Just to see how the nested loops work

```
for x in range(0, 4):
  for y in range(0, 8):
    print "(x=" + str(x) + ",y=" + str(y) + ")"
```

returns the string equivalent of x

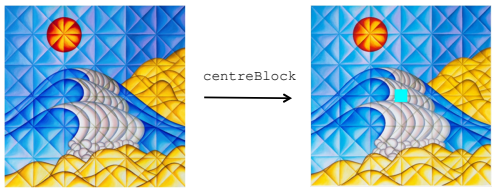
Different ranges

often we don't want to use the full range of pixels

```
def centreBlock(picture):
  midx = getWidth(picture) / 2
  midy = getHeight(picture) / 2
  for x in range(midx-20, midx+20):
    for y in range(midy-20, midy+20):
      pixel = getPixel(picture, x, y)
      setColor(pixel, cyan)
```


Different ranges

```
def centreBlock(picture):
  midx = getWidth(picture) / 2
  midy = getHeight(picture) / 2
  for x in range(midx-20, midx+20):
    for y in range(midy-20, midy+20):
      pixel = getPixel(picture, x, y)
      setColor(pixel, cyan)
```

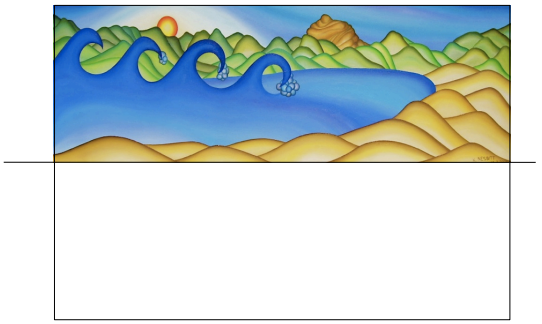


Arithmetic with coordinates

So far, whatever ranges x and y go through, we've just dealt with the (x, y) pixel

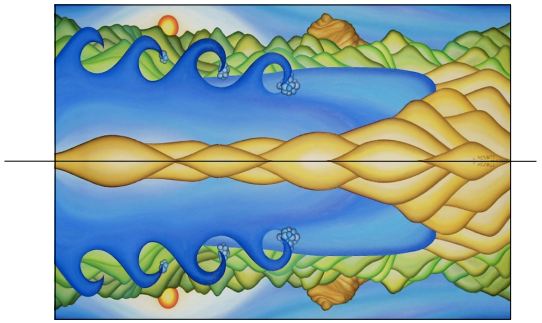
Some more interesting effects require a little more work with the coordinates

Horizontal Reflection



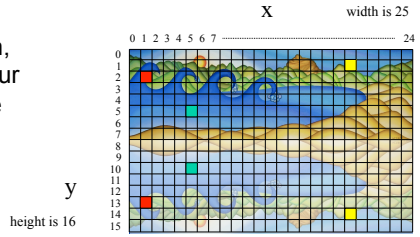
Reflect about the middle of the picture

Horizontal Reflection



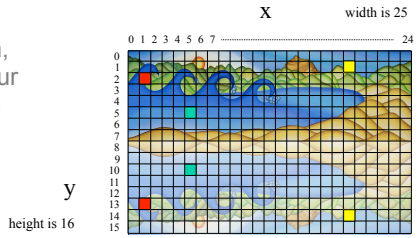
Horizontal Reflection

For a horizontal reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (x, height – 1 – y)



Horizontal Reflection

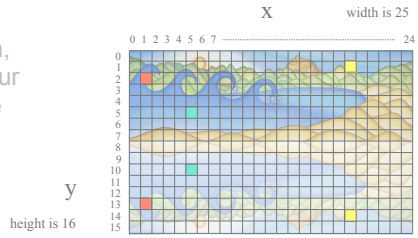
For a horizontal reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (x, height – 1 – y)



- pixel(1,2) copied to pixel(1, 16-1-2) = pixel(1,13)
- pixel(5,5) copied to pixel(5, 16-1-5) = pixel(5,10)
- pixel(19,1) copied to pixel(19, 16-1-1) = pixel(19,14)

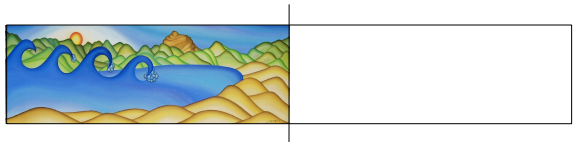
Horizontal Reflection

For a horizontal reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (x, height – 1 – y)



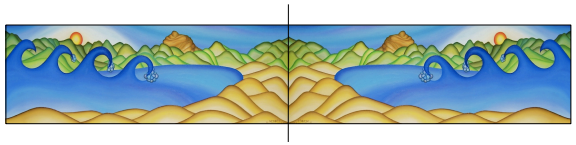
- pixel(1,2) copied to pixel(1, 16-1-2) = pixel(1,13)
- pixel(5,5) copied to pixel(5, 16-1-5) = pixel(5,10)
- pixel(19,1) copied to pixel(19, 16-1-1) = pixel(19,14)

Vertical Reflection

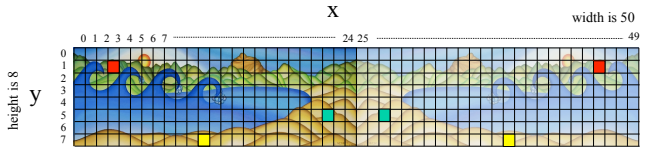


Reflect about the centre of the picture

Vertical Reflection

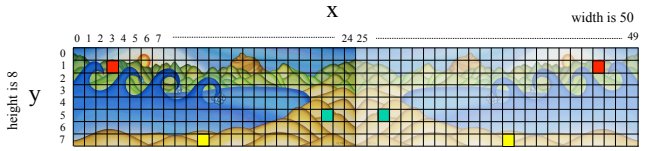


Vertical Reflection



For a vertical reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (width – 1 – x, y)

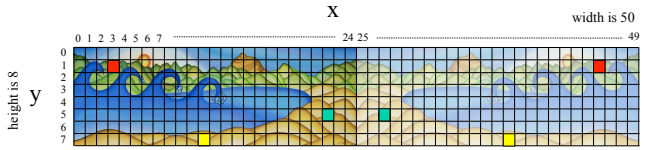
Vertical Reflection



For a vertical reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (width – 1 – x, y)

- pixel(3,1) copied to pixel(50-1-3, 1) = pixel(46,1)
- pixel(22,5) copied to pixel(50-1-22, 5) = pixel(27,5)
- pixel(11,7) copied to pixel(50-1-11, 7) = pixel(38,7)

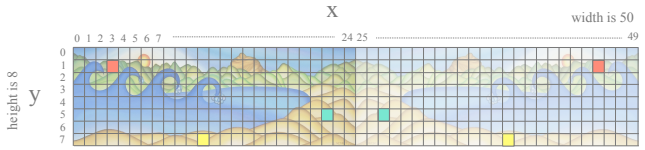
Vertical Reflection



For a vertical reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (width – 1 – x, y)

- pixel(3,1) copied to pixel(50-1-3, 1) = pixel(46,1)
- pixel(22,5) copied to pixel(50-1-22, 5) = pixel(27,5)
- pixel(11,7) copied to pixel(50-1-11, 7) = pixel(38,7)

Vertical Reflection



For a vertical reflection, we need to copy the colour of the pixel at (x, y) to the pixel at (width – 1 – x, y)

- pixel(3,1) copied to pixel(50-1-3, 1) = pixel(46,1)
- pixel(22,5) copied to pixel(50-1-22, 5) = pixel(27,5)
- pixel(11,7) copied to pixel(50-1-11, 7) = pixel(38,7)

Tutorial Question

Problem solving

This brings us back to the idea of problem solving!

It's not enough to be able to write code;

You need to be able to look at the problem, work out how to solve it, and then write the code that implements your solution

Programs in the book

Armed with what we've done here, the book shows (and explains) programs to . . .

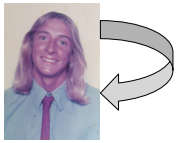
reflect an image about its vertical centre line

reflect an image about its horizontal centre line

reflect just a selected part of an image

Copying pictures to new pictures

Sometimes we might want to copy parts of a picture (ses the vertical reflection code)



original

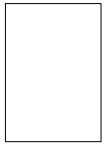
Copying pictures to new pictures

We can start the copy as a blank picture (the examples in the book use 7inx95in.jpg or 640x480.jpg)

Then we copy the colour of each pixel from the original to the corresponding pixel in the copy . . .

. . . and *return* the copy

original



blank copy

Why copy the colour?

We can't copy the actual pixel, because that pixel is a component of the object that is the original picture

Likewise, the copy picture has its own pixels (all of which happen to be white)

Remember: object types, reference copying

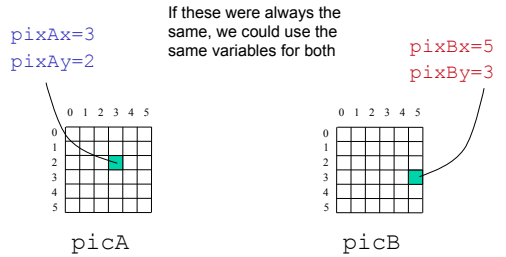
If we write `pixel2 = pixel1`, we just get two references to the same pixel

But if we change the colour of some of those white pixels, by copying the colour of the pixels in the original, we get a copy of the original picture

Two sets of coordinates

All of the copying functions in the book keep track of two sets of coordinates:

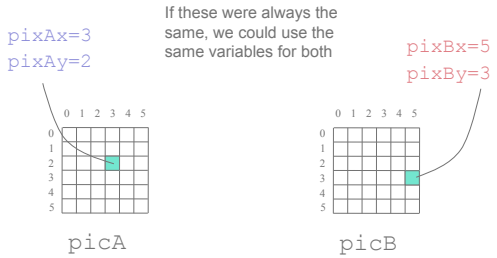
the coordinates of the source pixel being copied from, and the coordinates of the target pixel being copied to



Two sets of coordinates

But the functions do different sorts of copying, in which the target coordinates aren't the same as the source coordinates

We use loops to work through the source coordinates, and explicitly calculate the target coordinates



Programs in the book

Copying to the same position on the target
the source and target pixels are the same

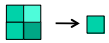
Copying to a different position on the target
every target pixel is further along and further down than the corresponding source pixel

Cropping a picture while copying it
taking only a selected range of source pixels

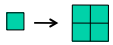
Programs in the book

Making a rotated copy
Copying from pixel(x, y) in the source to pixel(y, x) in the target

Copying and scaling down
only copy one pixel in two (actually, one in four)



Copying and scaling up
make four copies of every pixel, in a 2x2 square
(though you have to read the detail carefully to realise that that's what it's doing)



Revision
Functions to avoid repeated code

The collage demonstration in the book makes a really important point

The first version has a large chunk of code repeated five times with minor variations

For the second version, that large chunk of code has been rewritten as a function in its own right,
and the collage function just calls that function five times with varying arguments

Revision
Functions to avoid repeated code

The collage demonstration in the book makes a really important point

The first version has a large chunk of code repeated five times with minor variations

Whenever you find a significant chunk of code being repeated, rewrite it as a function, with arguments for the bits that vary, and replace the repeated chunks with repeated calls to this new function

INFT1004

Introduction to Programming

Module 5.2
Selection

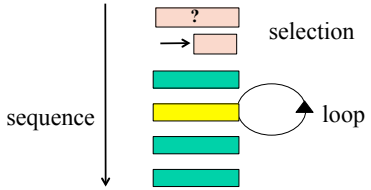
Guzdial & Ericson - Third Edition - chapters 4 and 5
Guzdial & Ericson - Fourth (Global) Edition – chapters 5 and 6

Sequence, selection, iteration

Programming has three essential building blocks

We've already met sequence and iteration

Selection determines which code to execute depending on specified conditions



Selection

Selection is done using the if statement. The if statement can have different parts (if, elif, else)

These parts are used (or not used) depending on the type of selection problem we need to solve.

The if, else, elif statements

Selection problems tend to be is done in three basic ways

if (test something is true)
    do something

The if, else, elif statements

Selection problems tend to be is done in three basic ways

if (test something is true):
    do something

```
if (today=="Tuesday"):
   gotoLecture("INFT1004")
```

The if, else, elif statements

Selection problems tend to be is done in three basic ways

```
if (test something is true):
    do something
else :
    do something else
```


The if, else, elif statements

Selection problems tend to be is done in three basic ways

```
if (test something is true):
    do something
else :
    do something else
```

```
if (today=="Tuesday"):
   gotoLecture("INFT1004")
else:
   workOnAssignment()
```


The if, else, elif statements

Selection problems tend to be is done in three basic ways

```
if (test something is true):
    do something
elif (test something else is true):
    do something different
elif (test something else is true):
    do something different
else:
    do something else
```


The if, else, elif statements

Selection problems tend to be is done in three basic ways

```
if (test something is true):
    do something
elif (test something else is true):
    do something different
elif (test something else is true):
    do something different
else :
    do something else
```

```
if (today=="Tuesday"):
   gotoLecture("INFT1004")
elif (today == "Wednesday"):
   gotoLecture("INFT1001")
elif (today == "Thursday"):
   gotoLecture("INFT1002")
else:
   gotoWork()
```

Lets look more closely at

- if
- else
- elif

The if statement

At its simplest, the if statement takes the form

```
if <condition> :
   <body>
```

selection

The if statement

Selection is done with the if statement. At its simplest, the statement takes the form

```
if <condition> :
   <body>
```

selection

```
if x < 5 :
   print "less than 5"
```


The if statement

```
if <condition> :
   <body>
```

selection

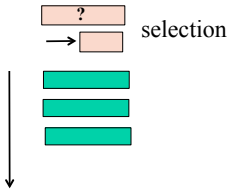
where `<condition>` means some condition to be tested

and `<body>` is one or more statements suitably indented

The if statement

If the condition is true when tested, the body will be executed; otherwise, the body will be ignored

```
if <condition> :
   <body>
```



Either way, any following statements are then executed

The if statement

If the condition is true when tested, the body will be executed; otherwise, the body will be ignored

```
if <condition> :
   <body>
```

So if the condition is false the statements associated with the if statement are just skipped

Conditions

A condition is something that, when evaluated, gives a value of true or false

```
if <condition> :
   <body>
```


Conditions

A condition is something that, when evaluated, gives a value of true or false

```
if <condition> :
   <body>
```

Technically, it is a boolean expression, which simply means an expression whose value is true or false

boolean is another simple type to add to int, float, and string

Conditions

Python actually implements booleans as integers.

The int 0 is regarded as the boolean false
any other int is regarded as the boolean true

```
false ⟶ 0
true  ⟶ 1 (usually)
         or any other integer
         (e.g. 6,10)
```


Python – true and false

Actually python uses a complex idea of what is true and false (you should ignore this except where odd bugs seem to arise)

```
false ⟶ 0, None, 0.0, '', []
true  ⟶ 1
         or just about anything else
```


Boolean expressions

We can make simple boolean expressions by comparing numbers with relational operators

	Expression	Value
3 is greater than 5	3 > 5	false
3 is less than 5	3 < 5	true
3 is equal to 5	3 == 5	false
3 is not equal to 5	3 <> 5	true
3 is greater than or equal to 5	3 >= 5	false
3 is less than or equal to 5	3 <= 5	true

Boolean expressions

Note that the check for equality uses a double '=' sign - because the single one is reserved for assignment

Expression	Value
3 > 5	false
3 < 5	true
3 == 5	true
3 <> 5	false
3 >= 5	false
3 <= 5	true

Boolean expressions

Expression	Value
3 > 5	false
3 < 5	true
3 == 5	true
3 <> 5	false
3 >= 5	false
3 <= 5	true

With the other 2-character operators, the order of the characters is important

Comparing two numbers

```
def testIfSelectionIntegers():
    numberA = 1
    numberB = 2

    if (numberA > numberB):
        print "numberA is greater than numberB"
    if (numberA == numberB):
        print "numberA is equal to numberB"
    if (numberA < numberB):
        print "numberA is less than numberB"
    if (numberA <> numberB):
        print "numberA is not equal to numberB"
    if (numberA >= numberB):
        print "numberA is greater than or equal to numberB"
    if (numberA <= numberB):
        print "numberA is less than or equal to numberB"
```

Mod5_2_testSelection.py

Comparing two numbers

```
def findMaximum(numberList):
    #A function that returns the maximum
    #number in the list
    #assumes it is a list of numbers

    maximum = numberList[0] #use first element as max

    for number in numberList:
        if number > maximum :
            maximum = number

    return maximum
```


What about other types

These comparison operators work well with integers and floats.

This makes a lot of sense.

What about strings?

String comparisons

```
stringA = "apple"
stringB = "orange"

if (stringA > stringB ):
    print "stringA is greater than stringB "
if (stringA == stringB ):
    print "stringA is equal to stringB "
if (stringA < stringB):
    print "stringA is less than stringB "
if (stringA <> stringB):
    print "stringA is not equal to stringB "
if (stringA >= stringB):
    print "stringA is greater than or equal to stringB"
if (stringA <= stringB):
    print "stringA is less than or equal to stringB"
```

Mod5_2_testSelection.py - testSelectionIfStrings()

Lets look more closely at

if

else

elif

if with *else*

We've seen the simplest form of the *if* statement

```
if <condition>:
  <body>
```

if with *else*

What if we want to do something when the condition is true and something different if the condition is false?

```
if <condition>:
  <statements>
else:
  <alternative statements>
```

if with *else*

When we follow an *if* and its body with an *else* and its body….

the body of the *if* will be done if the condition is true, and the body of the *else* will be done if the condition is false

```
if (numberA == numberB):
   print "numberA is equal to numberB"
else:
   print "numberA is NOT equal to numberB"
```

if with *else*

Guzdial & Ericson sometimes achieve the same effect with two separate *if*s

```
if (numberA == numberB):
   print "numberA is equal to numberB"
```


```
if (numberA <> numberB):
   print "numberA is NOT equal to numberB"
```

Guzdial & Ericson are probably not software engineers :-)

if with *else*

```
if (numberA == numberB):
   print "numberA is equal to numberB"

if (numberA <> numberB):
   print "numberA is NOT equal to numberB"
```

Most programmers prefer the proper use of *else*
Proper because it is easier to read and interpret

```
if (numberA == numberB):
   print "numberA is equal to numberB"
else:
   print "numberA is NOT equal to numberB"
```


Lets look more closely at

if

else

elif

elif

A common use of selection is to test for several mutually exclusive conditions

```
value = 101

if value < 64:
  cost = 31
elif value < 128:
  cost = 95
elif value < 192:
  cost = 159
else:
  cost = 223
```

elif is short for 'else if'

The oddly named *elif*

A common use of selection is to test for several mutually exclusive conditions

```
if value < 64:
  cost = 31
elif value < 128:
  cost = 95
elif value < 192:
  cost = 159
else:
  cost = 223
```

Note that only the first condition to be true will be executed.

If none of the conditions are true then the else part will be executed.

(There doesn't need to be an else part but often there is)

elif is short for 'else if'

Example: *Theme Park Tickets*

The cost of attending the Programmers Theme park depends on a persons age:

If a programmer is less than 3 years old admittance is free

If a programmer is less than 15 years old admittance is $40

If a programmer is more than 55 years old the price is $30

Otherwise the standard admission price is $75

write a function that calculates (and returns) the price

```
def calculateMoviePrice(age):
```


Example: *Theme Park Tickets*

```
def calculateMoviePrice(age):

  price = 0

  if age < 3:      #less than 3 - admittance is free
    price = 0
  elif age < 15:   #less than 15 - admittance is $40
    price = 40
  elif age > 55:   #more than 55 - price is $30
    price = 30
  else:            #standard admission price is $75
    price = 75

  return price
```

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost
13		

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost
13		
	0	

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

(13 < 3)
false

age	price	cost
13		
	0	

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

(13 < 15)
true

age	price	cost
13		
	0	

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost
13		
	0	
	40	

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost
13		
	0	
	40	

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

   price = 0

   if age < 3:
     price = 0
   elif age < 15:
     price = 40
   elif age > 55:
     price = 30
   else:
     price = 75

   return price
```

age	price	cost
13		
	0	
	40	
		40

Example: *Desk Check*

```
>>> cost = calculateMoviePrice(13)

def calculateMoviePrice(age):

    price = 0

    if age < 3:
      price = 0
    elif age < 15:
      price = 40
    elif age > 55:
      price = 30
    else:
      price = 75

    return price
```

age	price	cost
13		
	0	
	40	
		40

Selection Example - None

```
def testRequestStringIteration():

    listNames =[]

    numberOfNames = 3

    for number in range(1,numberOfNames+1):
      name = requestString("Please enter name " + str(number))

      if name <> None:
          listNames.append(name)

    return listNames
```


Selection Examples - Input

```
def testRequestStringSelection():

    answer = requestString("1, 2, 3 or 4?")

    if answer == None:
       print "No value entered"
    elif answer =="1":
       print "One"
    elif answer =="2":
       print "Two"
    elif answer =="3":
       print "Three"
    elif answer =="4":
       print "Four"
    else:
       print "Invalid selection = " + answer
```


Combining conditions

Sometimes a simple condition isn't enough

We might want to combine multiple conditions in various ways

We can do this with the 'logical operators' *and*, *or*, and *not*

Combining conditions

and, *or*, and *not*

mean pretty much what they mean in English

```
if age > 16 and < 25 :
```


Combining conditions

and, *or*, and *not*

mean pretty much what they mean in English

```
if age > 16 and < 25
```
✗

An important difference is that they must join complete conditions, not partial ones (the computer does not understand the English you speak)

```
if age > 16 and age < 25 :
```
✓

Combining conditions

and, *or*, and *not*

mean pretty much what they mean in English

```
if age > 16 and < 25
```
✗

An important difference is that they must join complete conditions, not partial ones (the computer does not understand the English you speak)

```
if age > 16 and age < 25 :
```
✓

Combining conditions

and, *or*, and *not*

```
if age > 16 and age < 25 :

#True only if both age > 16 and age < 25
# so 17,18,19,20,21,22,23,24 work
```

Combining conditions

and, *or*, and *not*

```
if age < 10 or age > 55 :

#True if either age < 10 or age > 55
# e.g.  .., 5,7,9, OR 56, 57, …
```


Combining conditions

and, *or*, and *not*

```
male = false

if not (male):

#Only true if male is false
```


Combining conditions

and, *or*, and *not*

```
number = 30

if not (number < 20): #true
```


INFT1004

Visual Programming

Module 5.3
Advanced Pictures

Guzdial & Ericson - Third Edition – chapter 5
Guzdial & Ericson - Fourth (Global) Edition – chapter 6

More Image Manipulation

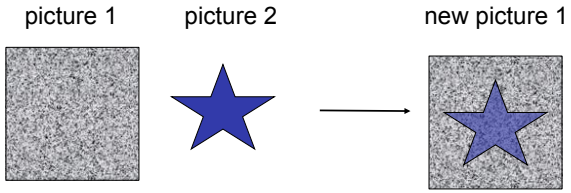
There's very little new programming in this module

Instead we revise the programming we've done by seeing some more approaches to manipulating images

- We'll see how to blend images
- subtract background
- use chromakey
- calculate distance between colours

Blending images

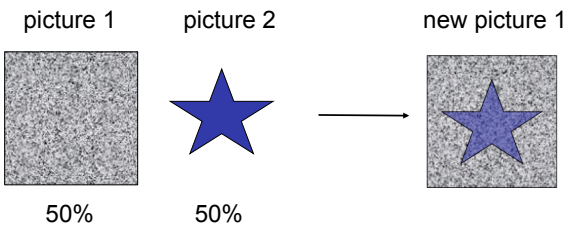
To get the effect of transparency, looking through one somewhat transparent image at another image, all we have to do is average the colours of the corresponding pixels.



Blending images

To overlay picture 1 and picture 2, we change the pixels of picture 1 to be the averages of the pixels of the two pictures

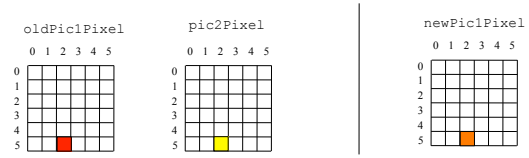
If we do a 50/50 average we get half of each picture; different proportions give different apparent transparency



Blending images

To overlay picture 1 and picture 2, we change the pixels of picture 1 to be the averages of the pixels of the two pictures

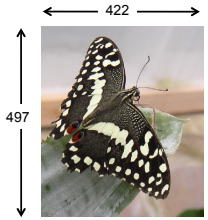
If we do a 50/50 average of the colours of each pixel we get half of each pixel; different proportions give different transparency effects



Transparent overlay

Let's overlay a picture of a butterfly on a picture of a bridge

bridge.jpg

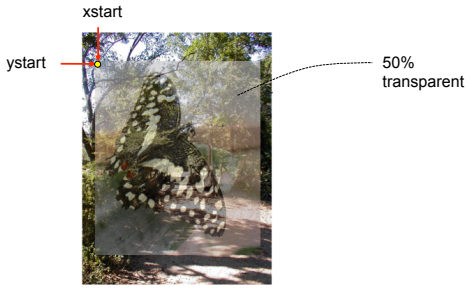


butterfly1.jpg

Transparent overlay

We need to specify at which coordinates of picture 1 (bridge) to start overlaying picture 2 (butterfly)

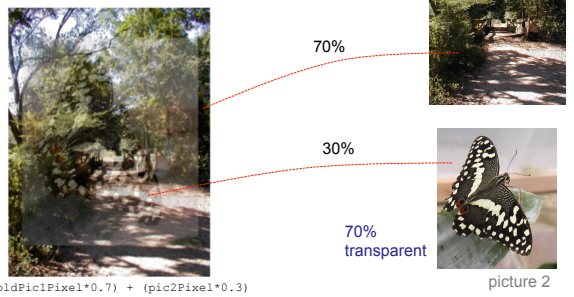
We need to specify how transparent to make picture 2



Conditions

Conditions

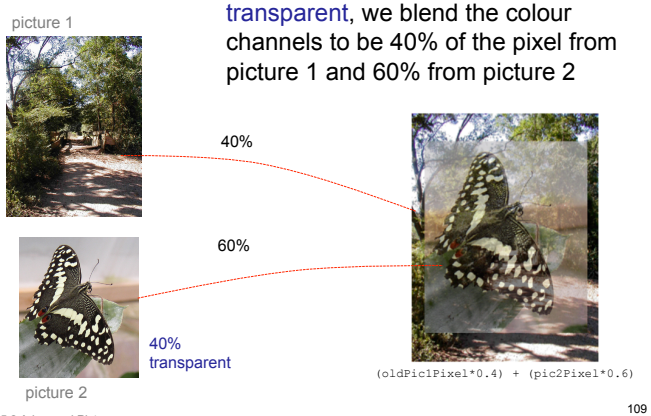
If we want picture 2 to be 70% transparent, we blend the colour channels to be 70% of the pixel from picture 1 and 30% from picture 2




```
(oldPic1Pixel*0.7) + (pic2Pixel*0.3)
```

Conditions

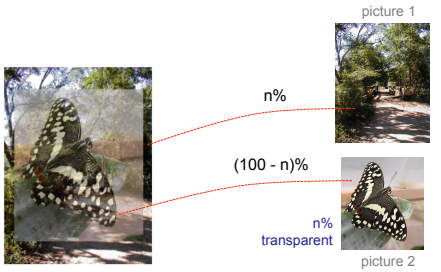
If we want picture 2 to be 40% transparent, we blend the colour channels to be 40% of the pixel from picture 1 and 60% from picture 2




```
(oldPic1Pixel*0.4) + (pic2Pixel*0.6)
```


Conditions

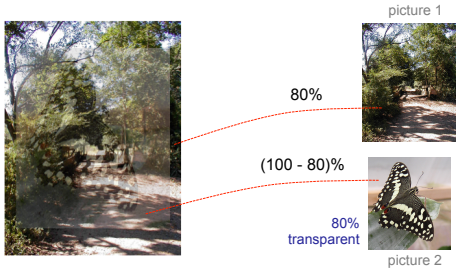
In general, if we want picture 2 to be n% transparent, we blend the colour channels to be n% of the pixel from picture 1 and (100 – n)% from picture 2




```
(oldPic1Pixel*(n/100)) + (pic2Pixel*(100-n)/100)
```


Example

e.g. if we want picture 2 to be 80% transparent, we blend the colour channels to be 80% of the pixel from picture 1 and (100 – 80)% from picture 2




```
(oldPic1Pixel*(80/100)) + (pic2Pixel*(100-80)/100)
(oldPic1Pixel* 0.8)) + (pic2Pixel* 0.2)
```


Example

```
def transparency(picture1, picture2, xStart, yStart, proportion):
    # overlays picture 2 in the centre of picture 1 with
    # a transparency of proportion (%)
    # returns a new picture - copy of picture 1 with the overlay
    # (there are no side effects)
```

Mod5_3_AdvancedPictures.py

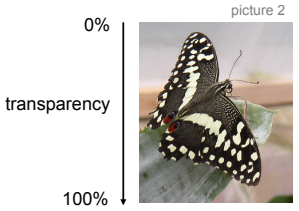
Fading

The book blends one picture into another by doing a section of picture 1 alone, a section of 50/50 transparency, and a section of picture 2 alone

This is worth examining to see how they manage the coordinates of picture 1, picture 2, and the new picture

(They paint to a new picture, whereas we're altering picture 1)

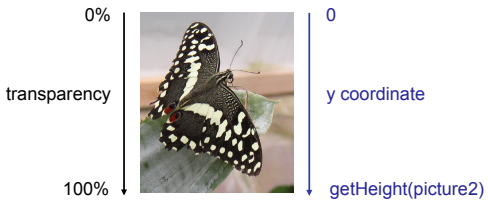
Fading



But we're going to do something quite different:

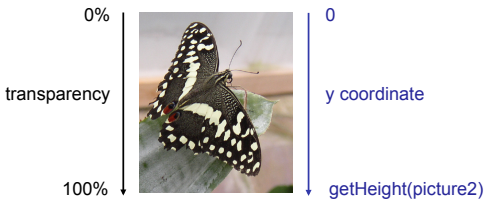
A full fade, from 0 transparency at the top of picture 2 to 100% transparency at the bottom

Fading



Every y value needs its own transparency level

Fading



How do the transparency levels relate to the coordinates?

```
proportion = 100 * y / getHeight(pic)
```

Varying transparency levels

How do we work that out?

By problem solving – looking at what we're given and using it to work out what we want

Problem solving is the first step in programming!

Subtracting Background

When we overlaid the picture of the butterfly on the bridge, the picture of the butterfly was a rectangle that included background

Subtracting Background

When we overlaid the picture of the butterfly on the bridge, the picture of the butterfly was a rectangle that included background

It might be good to get rid of the background, and only overlay the butterfly itself (the foreground)

That is, we want to subtract the background from the picture

To do this, we need to know which pixels are background and which are foreground

Subtracting Background

If we have one picture of the background alone, and one of the background with foreground . . .

We can compare the two, and choose only the pixels that are different

background

background + foreground

Subtracting Background

That way, we get just the foreground pixels.


background


background + foreground


foreground

Subtracting Background

```
#process all pixels - something like this
if backgroundColor <> backPlusForegroundColor
   setColor(foreGround, backPlusForegroundColor)
```


background


background + foreground


foreground

Subtracting Background

Or we can overlay on a third picture, (a new background)

foreground + new background


background

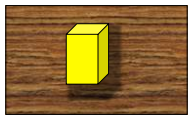

background + foreground


foreground

Where it falls over

If you use this in the real world – say you take a picture of something in front of a background and then a separate picture of the background –

you may encounter a couple of problems.

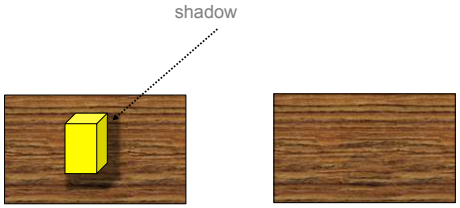

foreground + background


background

Where it falls over

Shadow is really background, but it's different from the background picture, so it registers as foreground

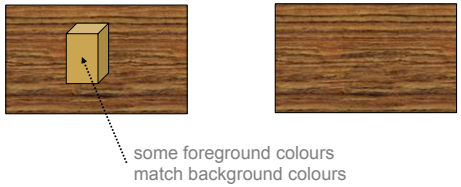
The problem with shadow can be fixed with direct lighting



Where it falls over

Another problem is that any colour in the foreground that is very similar to the background registers as background

The problem with similarity between foreground and background can be fixed only if we can ensure there is no such similarity



Chromakey

Chromakey is a technique for subtracting background that relies on the background being as close as possible to a single colour . . .

And a colour that doesn't occur very often in the foregrounds that we tend to work with

Why don't weather forecasters wear blue shirts?

Chromakey

Because they're filmed against a blue background, and blue is then eliminated from the picture in a background subtraction

(green is also used a lot – neither blue or green occur in skin colours – but watch what you wear)

Chromakey

If the foreground is photographed against a blue background . . .

The function becomes simpler, because it's always comparing with the same colour . . .

And the result is a lot cleaner

Distance Between Colours

```
distance(colour1, colour2)
```

This is a function that measures how close together two colours are

It goes up to about 442

Experiment with different colours to see what sorts of values it produces

Distance Between Colours

Let's enhance the green of colours close to white

(we will need to work out the distance between a pixel colour and the colour white)

Distance Between Colours

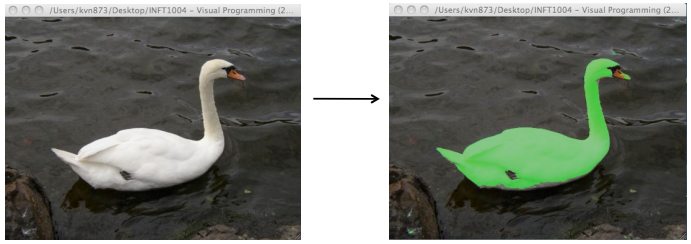
Let's enhance the green of colours close to white

(we will need to work out the distance between a pixel colour and the colour white)

For all the pixels in the picture,

- if the colour of that pixel is close to white
- enhance the green of a pixel

Irradiating a swan



For all the pixels in the picture,

- if the colour of that pixel is close to white
- enhance the green of a pixel

Irradiating a swan

This method works well with swan.jpg

```
for pixel in getPixels(picture):
    pixelColour = getColor(pixel)

    if distance(pixelColour , white) < 270:
       # Found distance by trial & error
       setBlue(pixel , getBlue(pixel ) / 2)
       setRed(pixel , getRed(pixel ) / 2)
       setGreen(pixel , 190)
```


What to do this week

- [ ] Do the Lab Week 5
- [ ] Prepare for the Practical Test (open book)
- [ ] Keep reading the textbook (Ch 4, 5, 6)