

A major practical activity exploring graphic manipulation with Python. The exercises give a concise introduction to the principles, using JES; a purpose built environment developed by Mark Guzdial.

Preparation required:

JES installed on all computers and sample files available.

These teacher notes available for all participants.

JES: A Python Interpreter

This practical exploration uses a Python environment developed by Mark Guzdial, who we met earlier through his Pixel Spreadsheet utility. One of the great things about the JES interpreter is that it allows you to define procedures that produce great visual effects. Being able to see a visual outcome is a well-known way of assisting in the debugging process. Don't worry if your efforts don't work first time. Persevere. Most of the exercises below don't require very complex coding, and you'll soon get used to the syntax. That said, if you are new to coding, these may prove challenging! JES v4 is included in the resources, but can be downloaded from goo.gl/GtNW1r. This is the teacher's page for Mark Guzdial's Media Computation course, pioneered at Georgia Tech, and includes links to other supporting material. More details about the specific operations demonstrated can be found in the slide notes. Read carefully before using the presentation.

```
>>> File=pickAFile()
>>> Picture=makePicture(File)
>>> openPictureTool(Picture)
>>> |
```

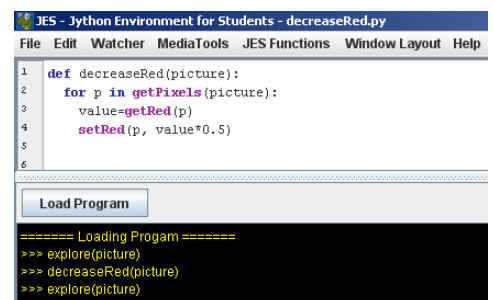
Once launched, the basic operations for selecting an image file and extracting the raw data is always the same (left). The picture tool allows you to interrogate the picture. This can be useful should you wish to select, for example, a range of pixels in your code. You can establish the range using the picture tool.

```
>>> pixels = getPixels(picture)
>>> print pixels [1600]
Pixel red=116 green=41 blue=106
>>>
```

Perhaps the most important operation is to extract a list of pixel objects. Most operations will then work on manipulating this list.

You can access each pixel by its position in the list (left), or iterate over a range of pixels (shown in the procedure below).

The best way to code effects is to create procedures which you can save and later modify. If you want to use a procedure in the interpreter, open it in the editor, then Load it into the interpreter. Here we have a procedure called decreaseRed(picture) defined (def) in the Editor. Once it is loaded into the program in the program window we can call the procedure just by using its name. The explore(picture) command will display the picture, but not give you the same interrogation tools that openPictureTool(picture) does.



```

1 def negative(picture):
2     for px in getPixels(picture):
3         red = getRed(px)
4         green = getGreen(px)
5         blue = getBlue(px)
6         negColor = makeColor(255-red, 255-green, 255-blue)
7         setColor(px, negColor)
8
Load Program

===== Loading Program =====
>>> file=pickAFile()
>>> picture=makePicture(file)
>>> explore(picture)
>>> negative(picture)
>>> explore(picture)
>>>

```

This example is worth studying for basic syntax. In the interpreter:

A file is selected

A 'picture' variable is assigned the values of the file.

The image is viewed with the explore(picture) tool

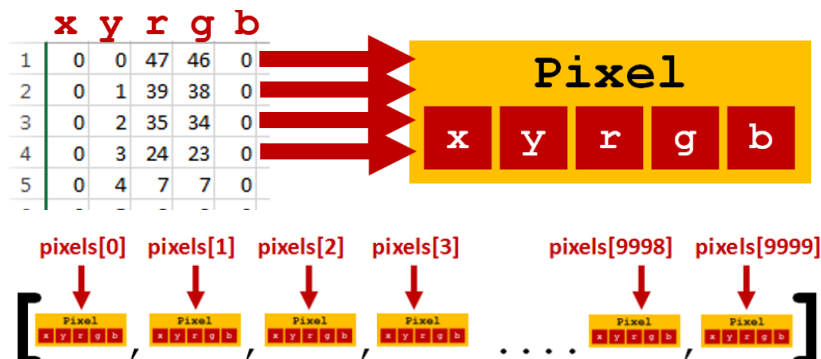
The procedure 'negative' is called. For this to work the procedure must have been defined (in the editor above) and loaded into the interpreter below. The procedure iterates through each pixel, gets the R, G and B values. It calculates another value (negColour) and assigns this to the pixel colour.

Once complete, the new image is viewed with the explore tool.

Emphasising The Concepts

With any new coding environment there is an inevitable overhead mastering the syntax and other particularities of the interface. The presentation talks through several simple procedure definitions but tries to root the detail in the broader concepts.

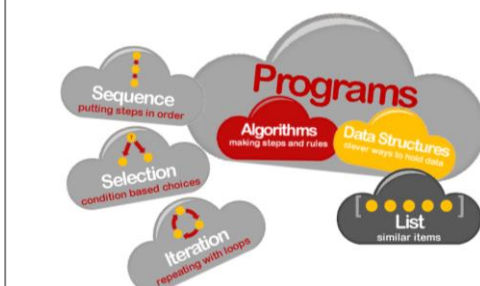
Some time is taken to ensure familiarity with a pixel object and its 5 attributes: x and y positions and the Red, Green and Blue colour channel values.



Once that is clear the getPixels(picture) function is visualised as returning a large array or list of pixel objects. Each pixel is identified by its list index position.

Most image manipulation involves opening a picture, extracting the pixel data into an array, then iterating over that array making changes as required. As such, it is an ideal medium for reinforcing simple algorithms, acting on the sort of linear structures children should already have some experience of by the end of KS3.

Emphasise the Big Picture



Most of the exercises that follow help reinforce the 'big 3' constructs that lie at the heart of all algorithms. Even when the focus is on the particulars of syntax, always take time to draw back and emphasise the bigger picture. Note here, how we refer to repetition as iteration. In many of our examples we will be using loops (repetition) to iterate over a long list.

We use the decreaseRed() procedure to reinforce key concepts as students read the code. Can they decompose the procedure and explain precisely how it works? Once defined, we can abstract away the detail of how the procedure works. Simply calling it by name makes the main sequence of code easier to understand.

We can generalise the procedures too. By introducing a parameter, we can call the same routine with different values, varying its intensity. Once we have a procedure for changing one colour, we can also duplicate and modify it in the Program Editor. This way we quickly produce 3 procedure definitions in the same program. Once it is loaded, we can call any of the three procedures it contains.

Concepts Are Key

```

1 def decreaseRed(picture):
2     for p in getPixels(picture):
3         value=getRed(p)
4         setRed(p, value*0.5)
5
Load Program

1 def decreaseRed(picture, amountInPercent):
2     for p in getPixels(picture):
3         value=getRed(p)
4         setRed(p, value*amountInPercent/100)
5

```

Our example also demonstrates how we can combine procedures to make more complex ones. It defines a `makeSunset()` procedure by combining calls to `decreaseBlue()` and `decreaseGreen()`. It deliberately omits a key element. What is missing in the definition? `makeSunset` needs to have an `amountInPercent` parameter to pass to the calls of `decreaseBlue()` and `decreaseGreen()`.



Take time to ensure these basics are understood. They are the key to further progress. There is no requirement to code all the subsequent examples. So long as everyone understands the principles, and each of the following algorithms is explained and discussed, they can be implemented at a later stage.

Armed with this basic knowledge we can turn our attention to some cool algorithms. These increase in difficulty. They are provided as exemplars to aid teachers understanding. Although they are referenced in the presentation, these are notes for the activities. Try them before delivering the presentation!

Building Cool Algorithms

Posterizing An Image

Posterizing an image involves reducing the number of colours that make up the image. We want to cycle through the R, G & B values of each pixel. For each value, we need to see what range it is in, and set a new value accordingly. The pseudocode is given below.

loop through the pixels:
get the RGB values
map values to smaller range
if < 64 set to 31
if > 63 and < 128 set to 95
if > 127 and < 192 set to 159
if > 191 and < 256 make 223

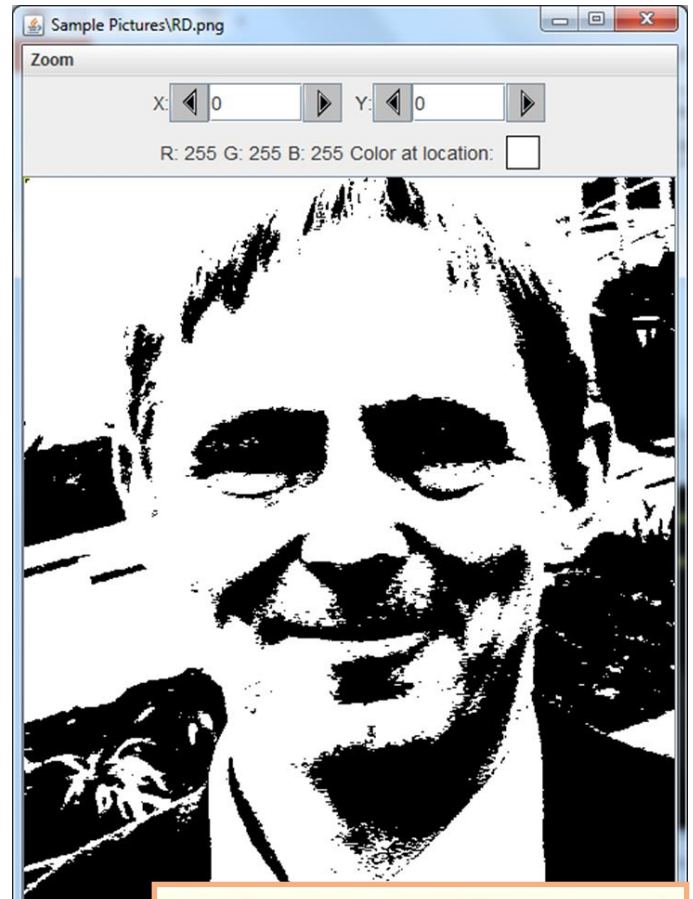
Knowing what we do about the Python syntax, you ought to be able to define a procedure to do this. We suggest you have a go yourself, but if you are struggling, a part solution is shown on the right. Can you see what else needs doing? Cover it if you want to try yourself!

```
def posterize(picture):  
    #loop through the pixels  
    for p in getPixels(picture):  
        #get the RGB values  
        red = getRed(p)  
        green = getGreen(p)  
        blue = getBlue(p)  
  
        #check and set red values  
        if(red < 64):  
            setRed(p, 31)  
        if(red > 63 and red < 128):  
            setRed(p, 95)  
        if(red > 127 and red < 192):  
            setRed(p, 159)  
        if(red > 191 and red < 256):  
            setRed(p, 223)
```

Grey Posterization

Once this is working, you should be able to create a black and white posterized image. In this case, we will want to calculate the luminance of a pixel. A simple measure of luminance would be to take the average the Red, Green and Blue values. If the luminance is less than a certain threshold, we could set the colour of the pixel to black, otherwise we could set it to white. Once working you could play about with the threshold and see the effect of changing it. Again, you'll learn most by doing it yourself, but if you want a solution look below.

```
def grayPosterize(picture):  
    for p in getPixels(picture):  
        r = getRed(p)  
        g = getGreen(p)  
        b = getBlue(p)  
        luminance = (r+g+b)/3  
        if luminance < 128:  
            setColor(p,black)  
        else:  
            setColor(p,white)
```



Antique

Photos

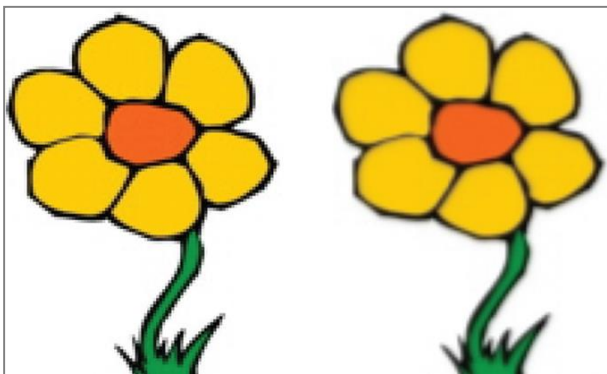
It's very easy to put a colour filter on to an image. We've already looked at how to decrease a colour channel, so here's a chance to

Turn to grayscale
For each pixel in turn:
Increase red slightly
Decrease blue

prove you can define a procedure from scratch (if you didn't before). Applying a sepia effect, like that on the right is very simple. The algorithm is given on the left. The rest is up to you.



Applying a Blur to a Picture



You'll want to select a simple picture with crisp lines for this. Blurring a picture involves an algorithm rather like that used to create a digital image—a demosaicing algorithm. Essentially, for each pixel, a new value is calculated by taking the R, G & B values and averaging them with the same values from the 4 adjoining pixels. This may seem complicated, so let's decompose the task. First establish you can get the R, G and

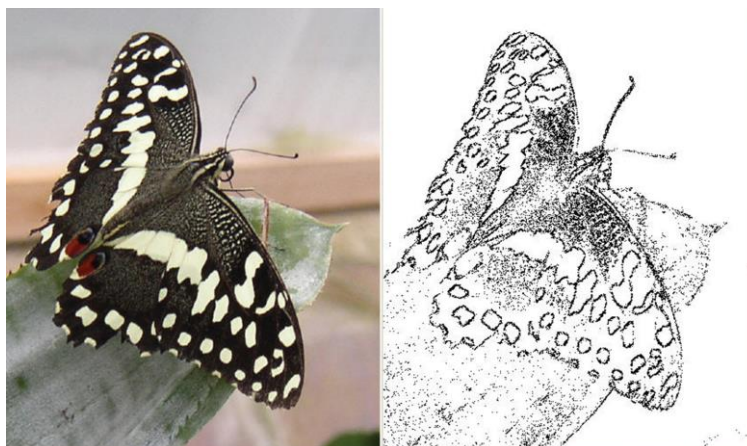
B values of the pixel. Then think about the relative x and y co-ordinates of the adjoining pixels. Try to get the same values from the one to the immediate left. If



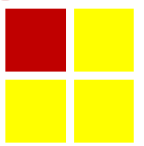
you can do that, a bit more copying, pasting and altering of the relative co-ordinates will get the rest. The final step is to figure out a way to add them up, divide by 5 and set the new value to the correct colour channel.

Creating An Outline

This is an interesting concept. Children always draw objects with thick outlines, but real world objects rarely have an outline. They do this because we perceive an outline where there is a sharp contrast between colours. We can harness this idea to convert a picture into an outline drawing. Warning! You may have mixed success with this, so experiment with values to find the optimum. The algorithm is simple, and builds on the understanding of sampling adjacent pixel values.



**If the difference in brightness
of the pixel below and to the right > 10:**
Copy a black pixel
Else:
Copy a white pixel



However, there are a new challenges here. Firstly we are creating a new image rather than altering an existing one. Secondly, we need to copy pixel values from one image to another at the right x.y co-ordinates. Thirdly, we need to consider how we will compute the luminance of a pixel so we can compare it with those to the bottom and right and determine whether it is an edge. We already have a way of computing luminance so if we define a procedure to do this for any pixel we can call it to calculate the values for 3 variables representing 3 pixels; here, right and down.

A second 'helper' function can then be defined to determine whether the comparison yields an edge. By separating these two helper functions out, you make the main loop of traversing and writing pixels much easier to manage. The two functions are shown right.

```
JES - Jython Environment for Students - outline.py
File Edit Watcher MediaTools JES Functions Window Layout Help
4 def isLine(here, down, right):
5     return abs (here - down) > 10 and abs (here - right) > 10
6
7 def lineDetect(filename):
8     source = makePicture(filename)
9     dest = makePicture(filename)
10    for x in range(0, getWidth(source)-1)
11        # comment: this will iterate across 1 line of the picture
12        # how will you iterate down each column?
13        # you need to solve this to get the y value used below
14        here = getPixel(dest, x, y)
15        # a similar approach is needed to get the 2 other pixels
16        hereLum = getLum(here)
17        # similar calls will need to be made for the other pixels
18        # once you have the luminance values you can call isLine()
19        # and consider how to setColor to white or black
20
```

```
JES - Jython Environment for Students - outline.py
File Edit Watcher MediaTools JES Functions Window Layout Help
1 def getLum(pixel):
2     return (getRed(pixel) + getGreen(pixel) + getBlue(pixel))/3
3
4 def isLine(here, down, right):
5     return abs (here - down) > 10 and abs (here - right) > 10
6
```

To create a second image file to write the new values to, we can simply open the source image file and use makePicture(filename) to create two data files, source and destination perhaps.

For every pixel interrogated in the source, we will set the corresponding pixel in the destination to black or white, depending on whether a line is detected.

So the start of the main lineDetect() procedure will look something like this. We've left the real challenge of iterating through the pixels to you to solve, with a few comments as hints to the approach.

This is not an easy challenge, but is included for those who already have some programming experience and familiarity with Python. If you are new to this, please don't be put off!

Merging Pictures

Once you are familiar with copying pixels from one file to another we can begin to understand how pictures can be merged.

The image shown copies pixels from one file to a blank white canvas. For the last third of each row of pixels, only half their values are copied. A second picture is also copied but offset by $\frac{2}{3}$ of the width of the first picture. Again, only half the values are copied for the same number of pixels as halved in the first.

With a bit more thought, we could have created the new picture without the mismatch of sizes shown. Further thought to a gradual reduction in values copied would have created a proper blend effect.



Taking The Ideas Further

We're just scratching the surface of image manipulation here. If the idea is interesting we suggest you look at Mark Guzdial's work on multimedia computation. The ideas (and some images) here are taken from him. He doesn't restrict himself to images but has tools for manipulating sounds and movies too. Realising that sound and pictures can both be represented as numbers opens up many interesting ideas. You could try, for example, listening to a picture, or seeing a sound. Take a look at Mark speaking at TEDx: [goo.gl/4QsS9T](https://www.youtube.com/watch?v=4QsS9T). If you're inspired his book is well worth the investment.

