Le mode Python de Processing, permet d'utiliser le langage Python et un certain nombre de classes Java. Ce qui permet de faciliter l'affichage notamment d'images dans des fenêtres.

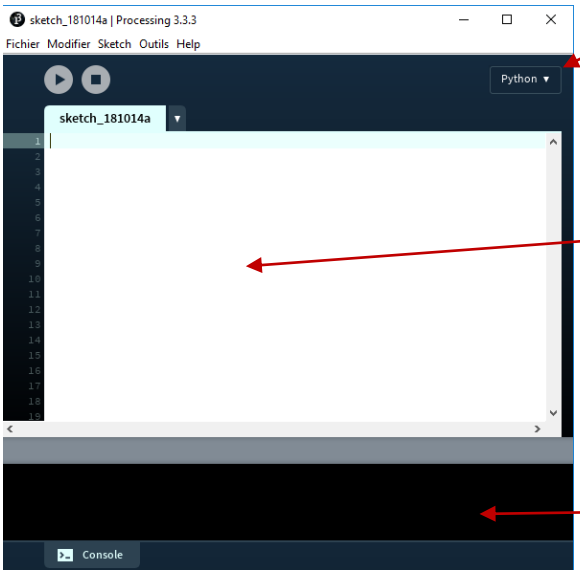
Davantage d'informations, de tutoriels et d'exemples sur : https://py.processing.org/

Télécharger Processing : https://processing.org/download/

Il est possible que dans certain lycée, on ne puisse pas (à cause des proxy) ajouter un mode.
Pour contourner ce problème :
Récupérer dans "mes documents/Processing/mode" Le dossier PythonMode là où cela a fonctionné et le coller dans ce même dossier là où cela n'a pas fonctionné.

L'espace de travail : Processing – mode Python

On utilise **Processing** un environnement de programmation en Java, JavaScript et python pour réaliser des programmes dont le rendu est essentiellement graphique, ce qui semble tout indiqué pour des images.



S'assurer que le mode Python est sélectionné, s'il n'est pas visible, il faudra l'installer (Ajouter un mode)

La zone où on écrit le code

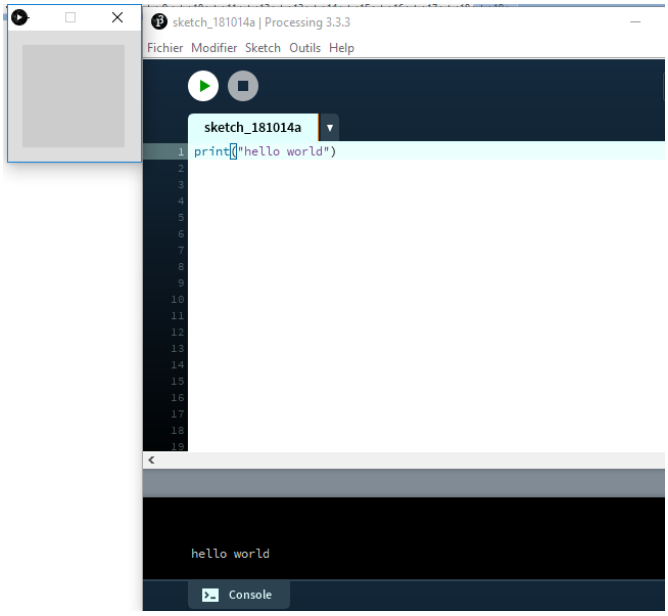
La console, qui permet de voir le résultat

Un incontournable 1er programme :

Écrire le code suivant :

```
Print(" Hello World ")
```

L'exécution produit le résultat escompté dans la console et une fenêtre de (100 x100 pixels) est également apparue, c'est dans cette fenêtre que nous ferons afficher nos images.

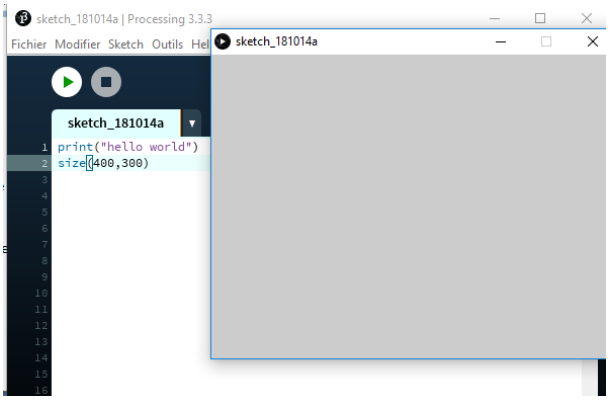


Commençons par modifier la taille de cette fenêtre :

Ceci se fait avec l'instruction :

```
size(400,300)
```

qui ouvrira une fenêtre de 400x300 pixels



Premières manipulations

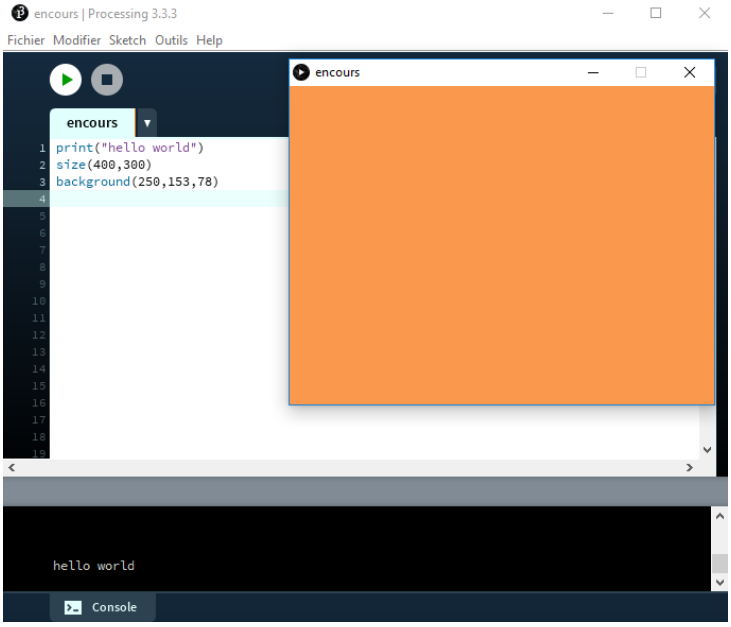
Cette fenêtre bien qu'apparemment vide, ne l'est pas, c'est une image constituée de 400 X 300 pixels tous de la même couleur.

La couleur d'un pixel est faite à partir d'un mélange de trois couleurs fondamentales : Le rouge, le vert et le bleu.

- Chacune de ces composantes à une valeur comprise entre 0 et 255
- Le noir : (0,0,0)
- Le blanc : (255,255,255)
- Le rouge : (255,0,0) ; le vert (0,255,0) et le bleu (0,0,255)

Le gris de cette fenêtre est : (204,204,204)

L'instruction pour modifier la couleur du fond de la fenêtre est : `background(255,0,0)` (en rouge)



Faites afficher une couleur de fond

Dans le menu "outils" il y a un sélecteur de couleur.

On utilise les nombres dans R, G et B

(H,S,B) et #..... sont d'autres façons de noter les couleurs, que nous n'utiliserons pas

Il faut imaginer une image comme un tableau de pixels, chaque case contient donc une couleur.

Abscisses →

Ordonnées ↓

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

Par exemple, une image de 10x10 pixels est un tableau où chaque case possède un **n° d'identification** (nommons le : **index**) et contient une couleur.

Pour repérer une case on utilise ses coordonnées (abscisse, ordonnée).

La 1ère ligne est la ligne 0 et la 1ère colonne est la colonne 0

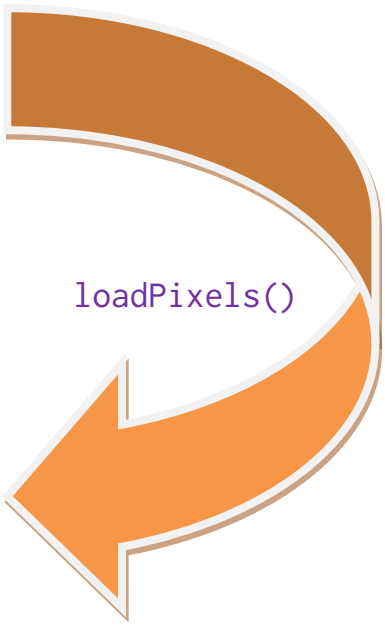
Exemples:

L'index 4 → coordonnées : (4,0)
L'index 30 → coordonnées : (0,3)
L'index 65 → coordonnées : (5,6)

L'instruction : `loadPixels()`

Cette instruction charge dans une liste tous les pixels :

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89



0	1	2	...	65	...	...	97	98	99

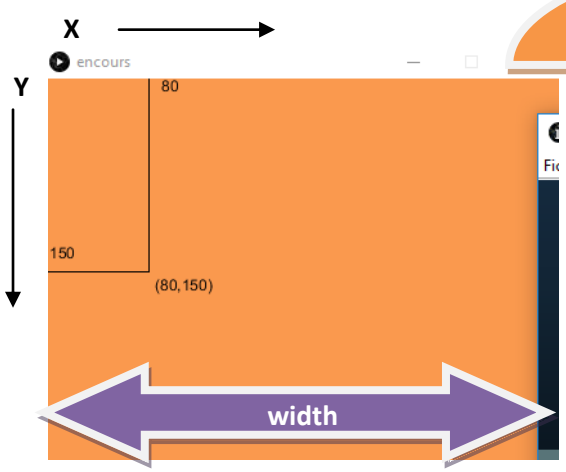
C'est cette liste que l'on pourra modifier, pour modifier l'image.

Dans cette liste on accède à "une case" par son index.

Par exemple : la case de coordonnées (5,6) donne l'index 65. La formule permettant de passer des coordonnées à l'index : **(abscisse) +(ordonnée *largeur de l'image)**

65=5+6*10

Dans une image de 400x300 pixels, le pixel de coordonnées (80, 150) se trouve à l'index 80+150*400=60080 de la liste



0	1	...	...	60080	...	...	119997	119998	119999

La formule générale est donc :

(x,y) → **x+y*width**

Nous allons écrire un programme qui modifie une partie de ces pixels.

L'objectif : Tous les pixels dont l'index est divisible par 10 seront mis en rouge, et les autres en vert

Commençons par donner une description de la méthode qui permet de réaliser cet objectif en langage naturel. Cela s'appelle : **un algorithme.**

1. Créer une fenêtre de 400x300 pixels
2. Charger les pixels dans une liste
3. En parcourant tous les pixels de la fenêtre:
   a. **Si** l'index du pixel est divisible par 10 **Alors** on le met en rouge
   b. **Sinon** on le met en vert
4. On rafraîchit la liste des pixels

Maintenant il faut traduire cet algorithme en Python…

Créer une fenêtre de 400x300 pixels →

```
size(400,300)
```

Charger les pixels dans une liste →

```
loadPixels()
```

En parcourant tous les pixels de la fenêtre:

Si l'index du pixel est divisible par 10 Alors on le met en rouge

Sinon on le met en vert →

```
for x in range(0,400,1):
    for y in range(0,300,1):
        index = x + y * width
        if (index % 10 == 0):
            pixels[index] = color(255,0,0)
        else:
            pixels[index] = color(0,255,0)
```

On rafraîchit la liste des pixels →

```
updatePixels()
```

Un peu d'explication sur la syntaxe de ce langage :

`for x in range(0,400,1):` signifie : pour x allant de 0 jusqu'à 399 par pas de 1 : on fait
`for y in range(0,300,1):` pour y allant de 0 jusqu'à 299 par pas de 1 : on fait
`index = x + y * width` on calcule l'index du pixel de coordonnées(x,y)
`if (index % 10 == 0):` si le reste de la division de l'index par 10 vaut 0: on fait
`pixels[index] = color(255,0,0)` on met le pixel en rouge
`else:` sinon
`pixels[index] = color(0,255,0)` on le met en vert

Il est très important de ne pas confondre "=" et "=="

`index = x + y * width` : **met le résultat de " x + y * width " dans index**

tandis que :

`index % 10 == 0` **compare le reste de la division de index par 10 avec 0**

Il est très important de respecter l'indentation :

L'indentation est le décalage d'une instruction à l'autre :

```
size(400,300)
loadPixels()
for x in range(0,400,1):
    for y in range(0,300,1):
        index = x + y * width
        if (index % 10 == 0):
            pixels[index] = color(255,0,0)
        else:
            pixels[index] = color(0,255,0)
updatePixels()
```

L'indentation permet au programme de "savoir" ce qu'il doit exécuter lors d'instruction comme "for" ou "if"

Le premier "for" exécutera toutes les lignes indentées ( )

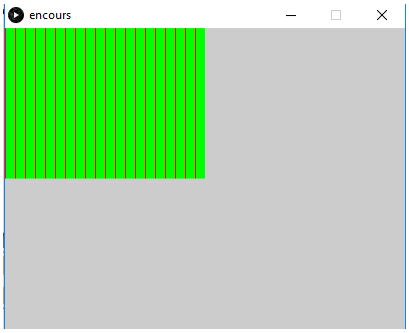
Le second "for" celles qui sont indentées ( )

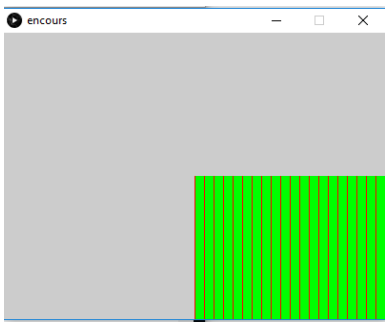
Le "if" celles qui sont indentées ( )

Voilà le rendu :

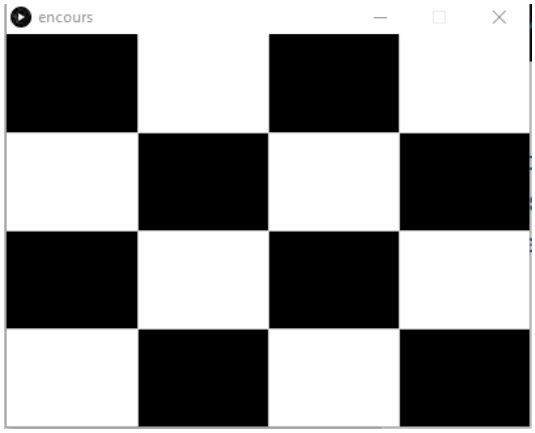
Remarquez au passage que les lignes précédées d'un # sont des commentaires de programmes (c'est très utile lorsqu'il y a beaucoup de lignes)

Sauriez-vous obtenir ces résultats :





Réalisez ce rendu :

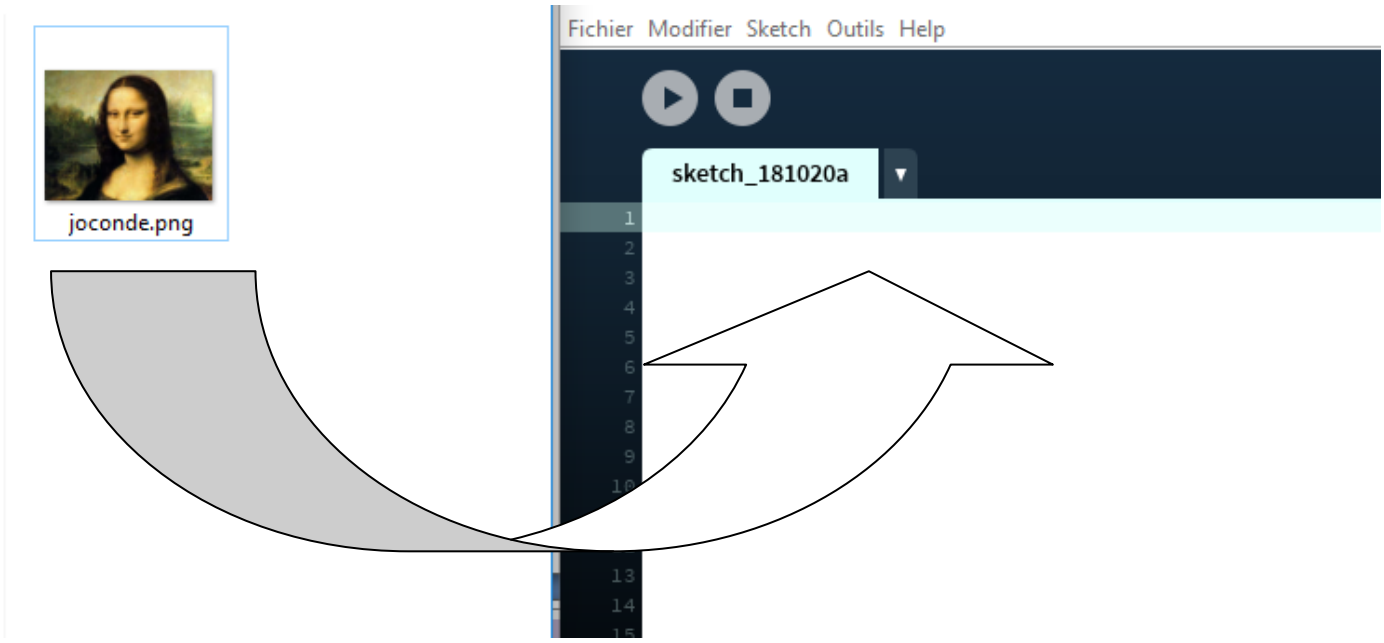


Voici le code pour la 1ère ligne:

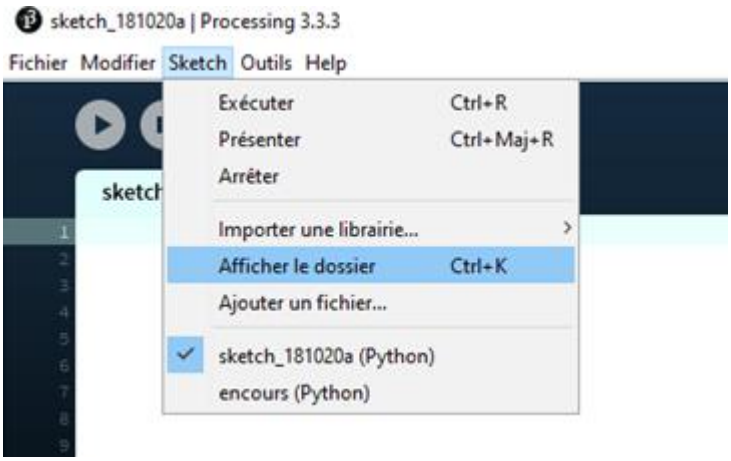
```
#première ligne
        if x<width/4 and y<height/4:
             pixels[index] = color(0,0,0)
        if x>width/4 and x<width/2 and y<height/4:
            pixels[index] = color(255,255,255)
        if x>width/2 and x<3*width/4 and y<height/4:
            pixels[index] = color(0,0,0)
        if x>3*width/4 and x<width and y<height/4:
            pixels[index] = color(255,255,255)
```

Faire afficher une image

On crée un nouveau "sketch" et on **glisse dépose** une image dans le sketch :

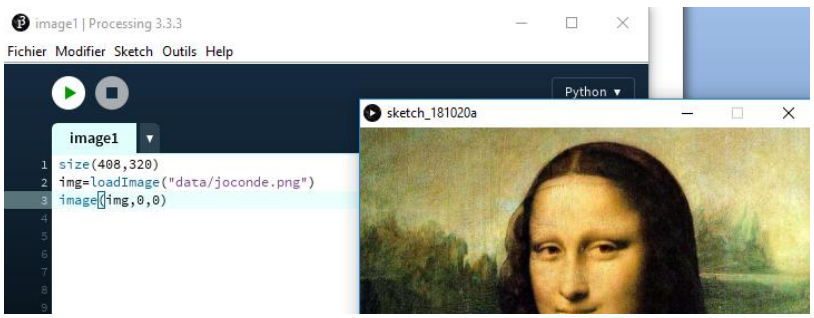


L'image est automatiquement mise dans un dossier "data" dans le dossier du sketch. Vous pouvez le vérifier dans l'onglet "sketch" / "Afficher le dossier"



L'image que j'ai choisie pour ce travail est celle de la Joconde (dimensions : 408 x 320 pixels)

On crée une fenêtre d'affichage aux dimensions de cette image, on charge l'image dans une variable, puis on l'affiche



L'instruction pour charger l'image dans la variable img :

```
img=loadImage("data/joconde.png")
```

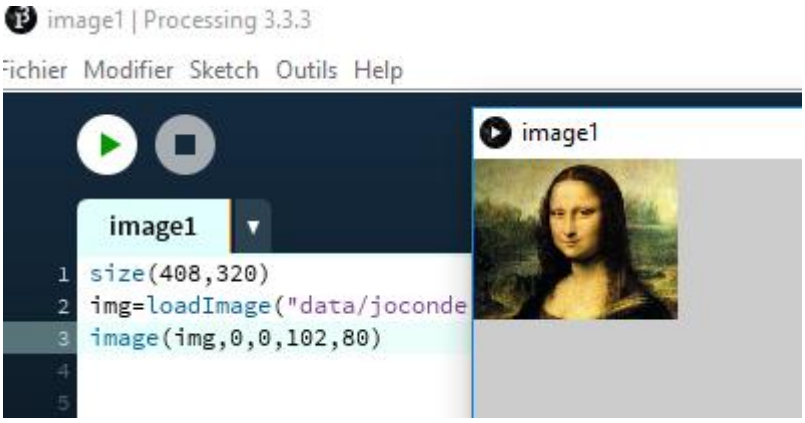
L'instruction pour l'afficher est :

```
image(img,0,0)
```

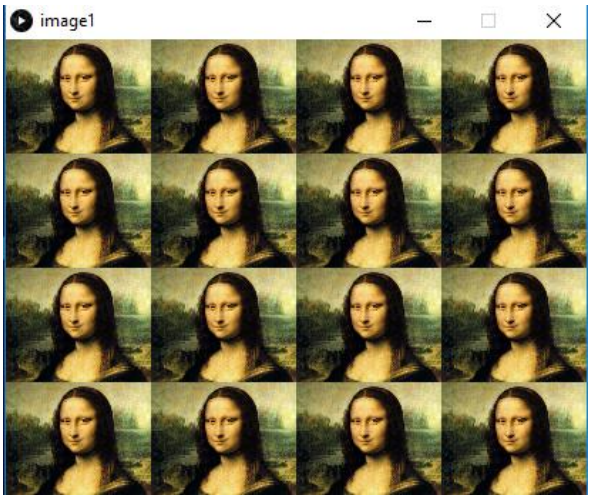
On affiche l'image de telle sorte que le coin supérieur gauche de l'image soit aux coordonnées (0,0) de la fenêtre

On peut également décider de l'afficher avec des dimensions plus petite:

```
image(img,0,0,102,80)
```



Réalisez un programme qui affiche le résultat suivant :



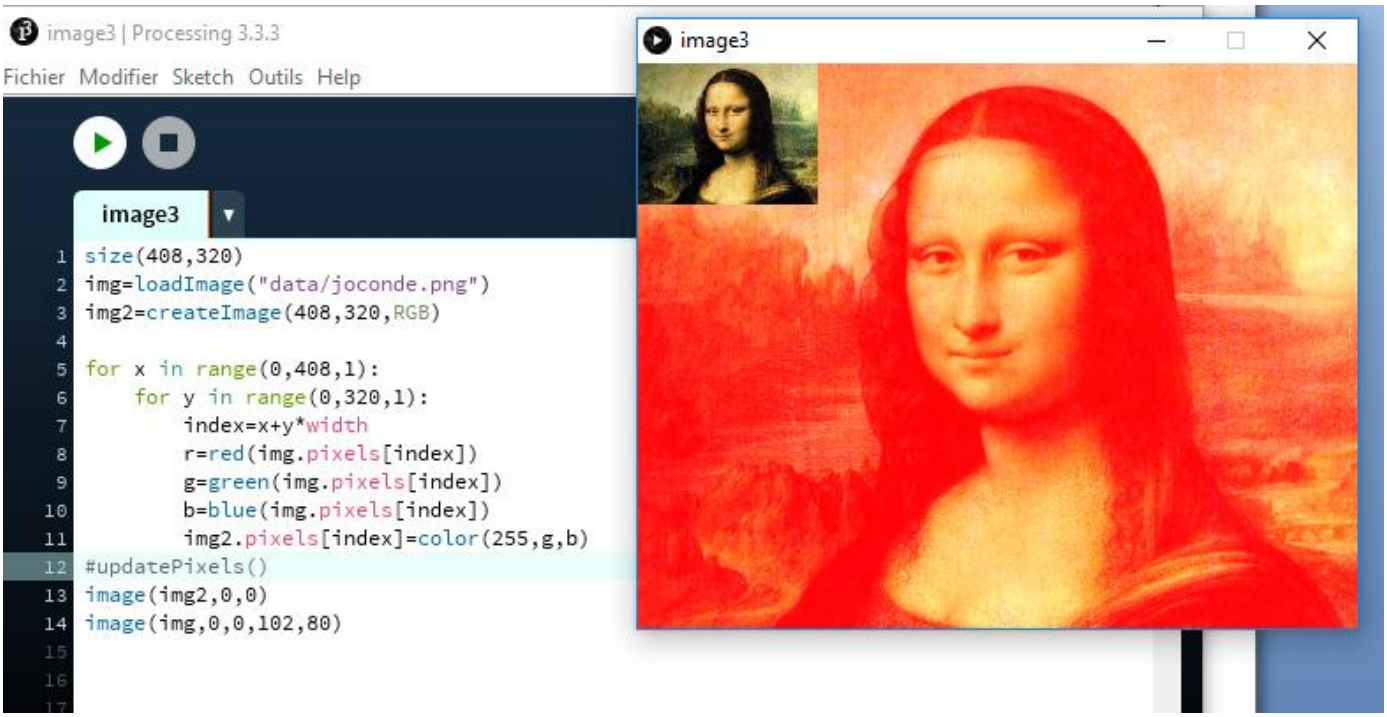
Cet exercice permet de s'entraîner sur l'utilisation d'une "double boucle" :

```
for i in range(  ,  ,  ):
    for j in range(  ,  ,  ):
        image(img,  ,  ,  ,  )
```

Manipulation des pixels d'une image

Le programme suivant :

- Charge l'image de la Joconde
- Crée une image de mêmes dimensions
- Récupère dans les variables r,g et b les composantes (rouge, verte et bleue) de chaque pixels de le joconde
- Remplit l'image "vide" avec ces mêmes composantes sauf pour le rouge qu'il met à 255.
- On affiche les deux images



On a réalisé un filtre rouge

Remarque : On n'utilise pas l'instruction `loadpixels()`, car le chargement d'une image et la création d'image créent automatiquement des listes de pixels accessibles avec `img.pixels[index]` (qui contient une couleur)
Et comme on ne "load" rien il n'y a rien à "uploader"

Réalisez d'autres filtres

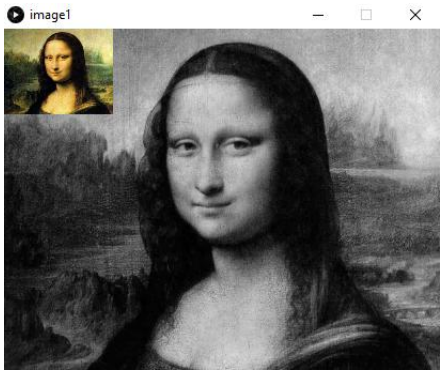
Niveaux de gris

Chaque pixel d'une image est une combinaison de trois couleurs (r, g, b).

Pour réaliser un niveau de gris (ce n'est pas la seule méthode):

On remplace ces trois valeurs par leur moyenne : $m = \frac{r+g+b}{3}$ (cette moyenne s'appelle la luminance)

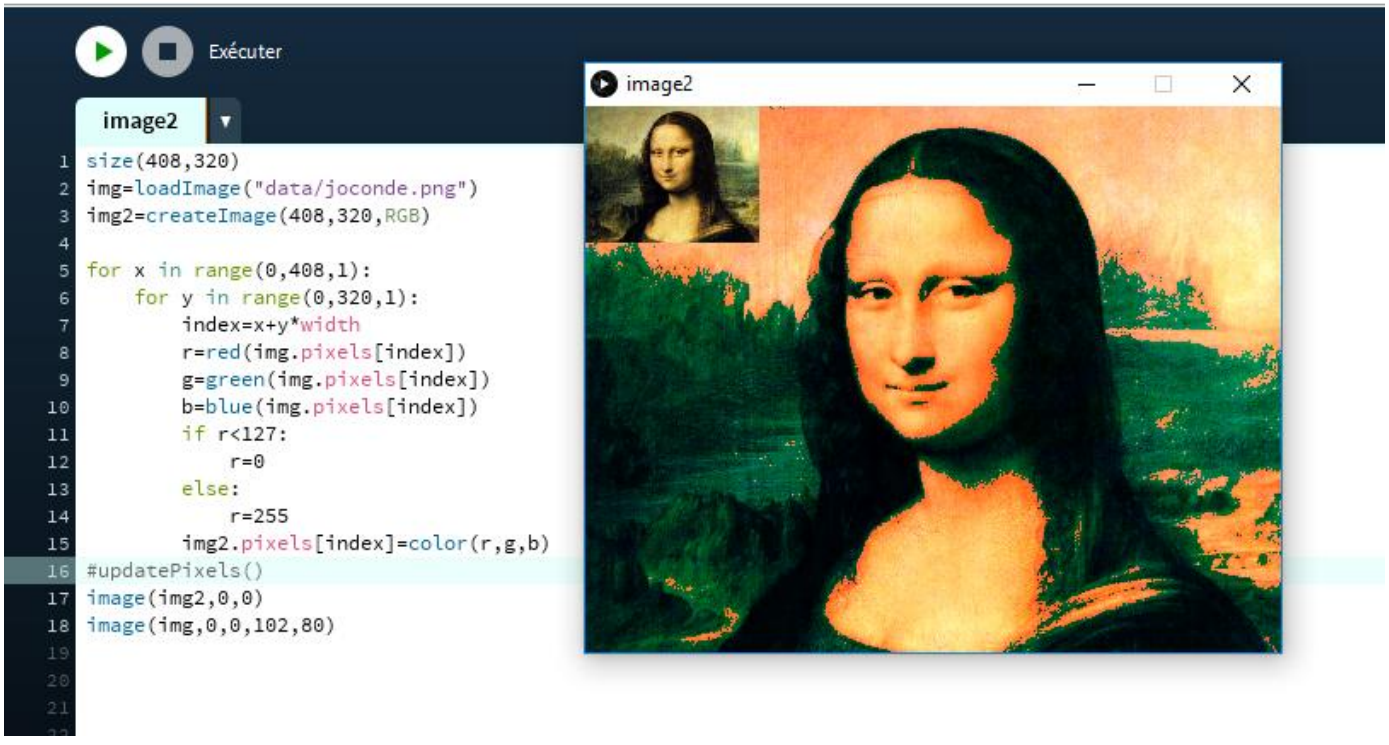
En vous aidant du programme précédent, réalisez un programme qui met en niveau de gris notre image de la Joconde.



Filtrage par seuil

Le filtrage par seuil consiste, pour chaque pixel d'une image, d'imposer une valeur à l'une des composantes si celle-ci est inférieure(ou supérieure) à un seuil que l'on aura fixé.

Par exemple le programme ci-dessous met à zéro les composantes rouges si elles sont inférieures à 127 et à 255 sinon (sans toucher aux autres composantes)



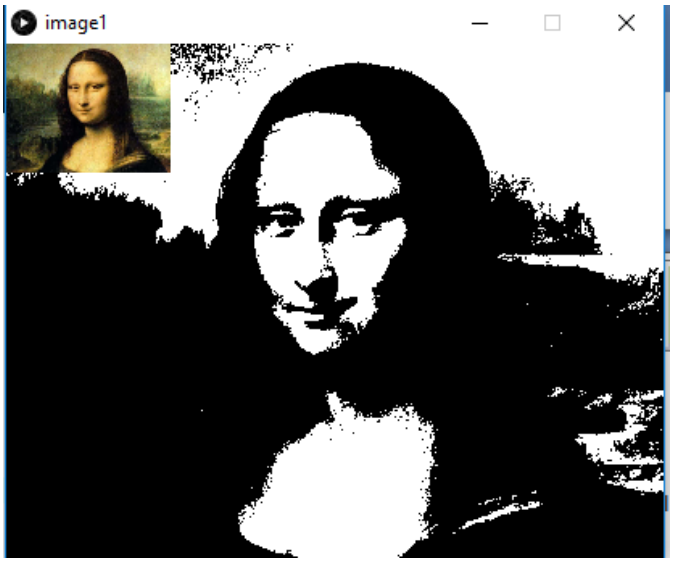
Réalisez d'autres filtrages par seuil

Noir et blanc

Pour réaliser une image en noir et blanc, on commence par mettre l'image en niveau de gris.

Donc chaque pixel est coloré avec une combinaison (m, m, m) puis on réalise un filtrage par seuil en mettant à zéro toutes les composantes du pixel si m est inférieur à 127 et à 255 sinon.

Réalisez ce programme :



Exercice : Remplacez les composantes (r, g, b) de chaque pixel de notre image par la combinaison (255-r, 255-g, 255-b)

Exercice : Faire afficher uniquement les composantes rouges d'une image (en mettant à 0 les autres)

Exercice : Même exercice, mais avec les composantes vertes (puis les bleues)

Exercice : Imaginez une modification et réalisez-là.

Modification par convolution

Modifier une image par convolution consiste à remplacer les composantes de chaque pixel par une combinaison des valeurs des composantes des pixels voisins et de lui-même (tout un programme …)

On ne prendra en compte que les pixels qui ont 8 voisins :

p a 8 voisins

a	b	c
d	p	e
f	g	h

Et leurs coordonnées →

(x-1,y-1)	(x,y-1)	(x+1,y-1)
(x-1,y)	(x,y)	(x+1,y)
(x-1,y+1)	(x,y+1)	(x+1,y+1)

Par exemple : On va lui appliquer la modification suivante : (**filtre passe haut**)

$$\begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}$$

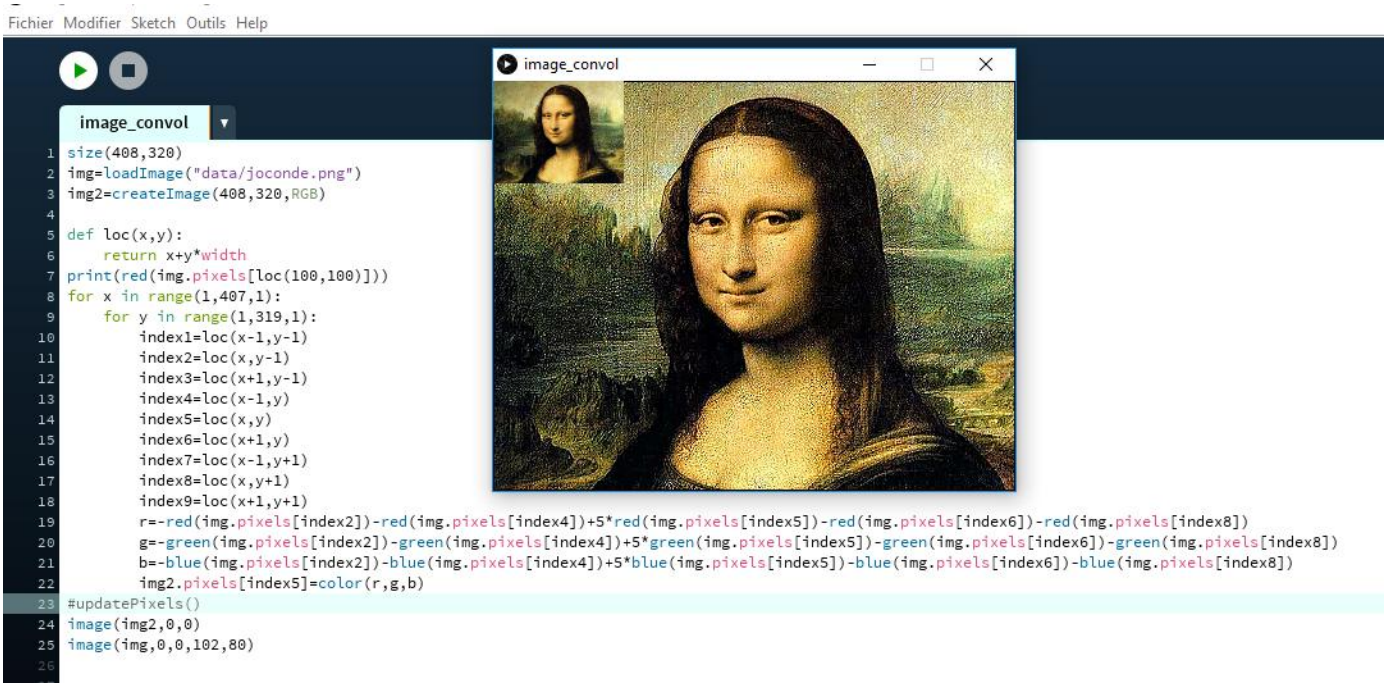
a	b	c
d	p	e
f	g	h

→

a	b	c
d	P'	e
f	g	h

Avec : $p' = 0 \times a - 1 \times b + 0 \times c - 1 \times d + 5 \times p - 1 \times e + 0 \times f - 1 \times g + 0 \times h$

Donc dans ce cas : $p' = -b - d + 5 \times p - e - g$



Remarque : On a ici créé une fonction `loc(x,y)` qui calcule et renvoie l'index du pixel dans la liste

Réalisation d'un flou

Pour le réaliser on applique la modification :

$$\frac{1}{9}\begin{pmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{pmatrix} \longrightarrow$$

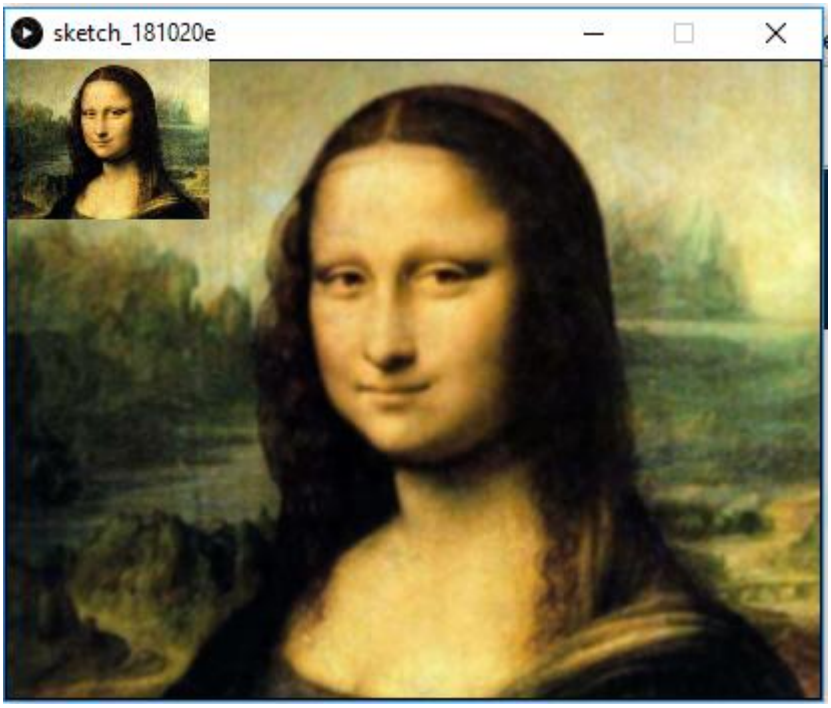
a	b	c
d	p	e
f	g	h

$\longrightarrow$

a	b	c
d	p	e
f	g	h

Avec : $p' = \frac{a+b+c+d+p+e+f+g+h}{9}$

Réalisez ce flou



Réalisez la transformation avec cette matrice :

$$\begin{pmatrix}1 & 2 & 1\\ 2 & -11 & 2\\ 1 & 2 & 1\end{pmatrix}$$

Un plus : Le programme qui utilise une fonction de convolution, ce qui évite d'avoir à écrire ces longues lignes de calculs :

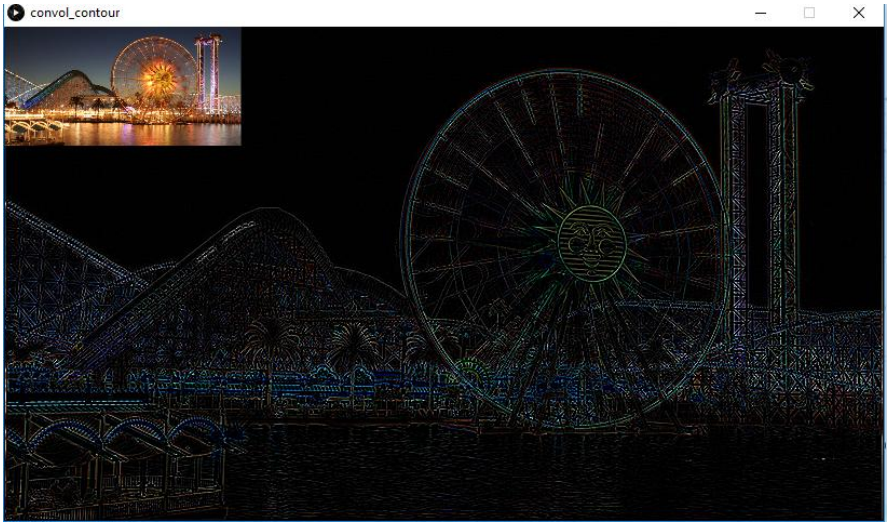
convol_matrice1

```
matrix =  [ [ -1, -1, -1 ],
            [ -1,  9, -1 ],
            [ -1, -1, -1 ] ]

def convolution(x, y, matrix, img):
    rtotal = 0.0
    gtotal = 0.0
    btotal = 0.0
    for i in range(-1,2,1):
        for j in range(-1,2,1):
            index=(x-i)+(y-j)*width
            rtotal=rtotal+red(img.pixels[index])*matrix[i+1][j+1]
            gtotal=gtotal+green(img.pixels[index])*matrix[i+1][j+1]
            btotal=btotal+blue(img.pixels[index])*matrix[i+1][j+1]
    rtotal = constrain(rtotal,0,255)
    gtotal = constrain(gtotal,0,255)
    btotal = constrain(btotal,0,255)
    return color(rtotal,gtotal,btotal)


def setup():
    global img
    size(408, 320)
    global img2
    img2=createImage(408,320,RGB)
    #print(matrix[2][2])
    frameRate(30)
    img = loadImage("data/joconde.png")
    for x in range(1,407):
        for y in range(1,319):
            index = x + y*width
            c = convolution(x,y,matrix,img)
            img2.pixels[index] = c
    #updatePixels()
    image(img2,0,0)
    image(img,0,0,102,80)
```

Taper ce programme et essayer avec d'autres matrices

Détection des contours

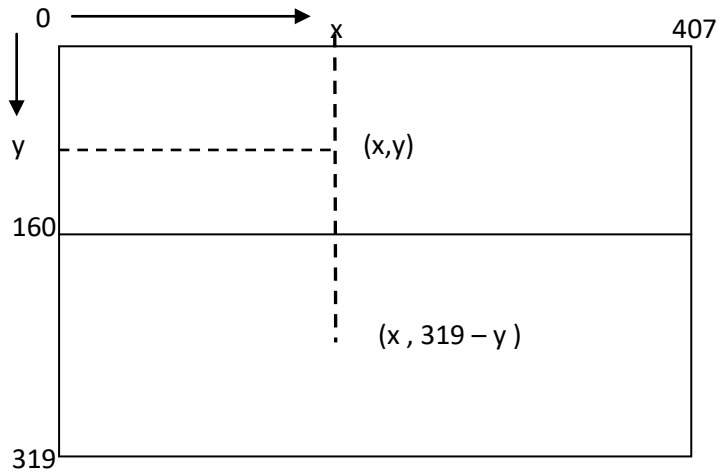
Utiliser le programme précédent et des recherches sur internet pour réaliser un programme de détection des contours sur une autre image (celle de la Joconde ne s'y prête pas bien)



Transformations géométriques

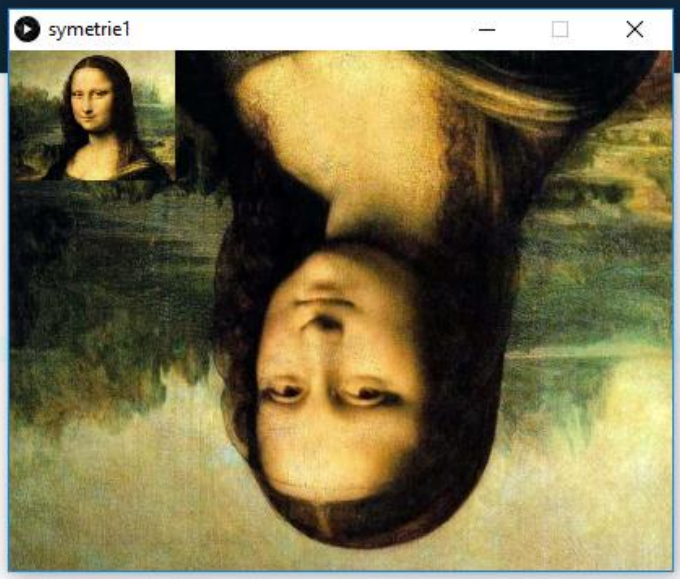
Symétrie horizontale :

Il faut réaliser la symétrie par rapport à la moitié de la hauteur :



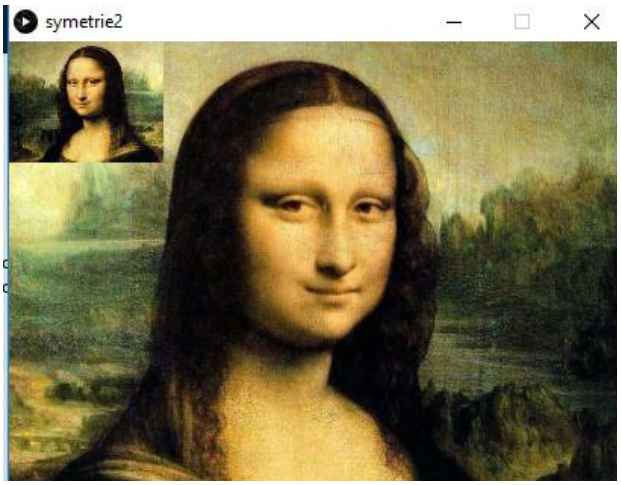
Il faut donc inter changer les pixels de coordonnées (x , y) et (x , 319 – y)

```
size(408,320)
img=loadImage("data/joconde.png")
img2=createImage(408,320,RGB)

for x in range(0,408,1):
    for y in range(0,160,1):
        index1=x+y*width
        index2=x+(319-y)*width
        img2.pixels[index1]=img.pixels[index2]
        img2.pixels[index2]=img.pixels[index1]
#updatePixels()
image(img2,0,0)
image(img,0,0,102,80)
```



Symétrie verticale :

- A vous de réaliser cette symétrie →
- Imaginez d'autres transformations géométriques



Récréation

Puisque nous sommes dans Processing, on peut utiliser la fonction draw() qui a la particularité de s'exécuter 60 fois par seconde, ce qui permet d'ajouter un peu d'animation:

- **Votre travail : Écrire ce programme et l'expliquer**
- **Modifiez l'instruction :** `fill(r,g,b)` **en** `fill(r,g,b,128)` **. Quel est l'effet ?**
- **Remplacez également l'instruction** `rect(x,y,10,10)` **par** `ellipse(x,y,10,10)`

Une image 2D mappée en 3D

Voici un programme emprunté au tutoriel de Daniel Shiffman

```
cellsize = 2 # Dimensions of each cell in the grid

def setup():
    global img, cols, rows, cellsize
    size(200, 200, P3D)
    img  = loadImage("data/sunflower.jpg")  # Load the source image
    cols = width/cellsize                   # Calculate number of columns
    rows = height/cellsize                  # Calculate number of rows

def draw():
    global img, cols, rows, cellsize

    background(0)
    loadPixels()
    # Begin loop for columns
    for i in xrange(cols):
        # Begin loop for rows
        for j in range(rows):
            x = i*cellsize + cellsize/2  # x position
            y = j*cellsize + cellsize/2  # y position
            loc = x + y*width            # Pixel array location
            c = img.pixels[loc]          # Grab the color
            # Calculate a z position as a function of mouseX and pixel brightness
            z = (mouseX/(float(width))) * brightness(img.pixels[loc]) - 100.0
            # Translate to the location, set fill and stroke, and draw the rect
            pushMatrix()
            translate(x,y,z)
            fill(c)
            noStroke()
            rectMode(CENTER)
            rect(0,0,cellsize,cellsize)
            popMatrix()
```

Je vous laisse le soin de voir ce qu'il fait et de le comprendre…

Éliminer le "bruit" d'une image

Voici une image avec du "bruit" :

Télécharger cette image :

Enregistrez là et glissez déposez-là dans un nouveau sketch

Pour éliminer le bruit on parcourt l'image et on remplace chaque pixel par la médiane de ce pixel et de ses 8 voisins. C'est ce qu'on appelle un **filtre médian**.

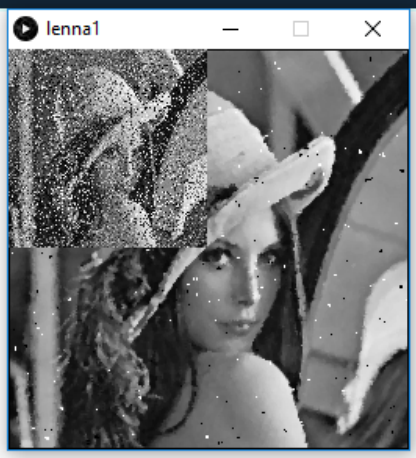
Comme c'est une image en niveaux de gris, il suffit de le faire pour l'une des composantes (par exemple le rouge)

Fichier Modifier Sketch Outils Help

Exécuter

lenna1

```
size(256,256)
img=loadImage("data/lennaSP25.gif")
img2=createImage(256,256,RGB)

def loc(x,y):
    return x+y*width  #fonction

for x in range(1,255,1): # on parcourt l'image sauf les bords
    for y in range(1,255,1):
        liste=[] # création d'une liste vide
        index1=loc(x-1,y-1)# on calcule les indexs des 9 pixels
        index2=loc(x,y-1)
        index3=loc(x+1,y-1)
        index4=loc(x-1,y)
        index5=loc(x,y)
        index6=loc(x+1,y)
        index7=loc(x-1,y+1)
        index8=loc(x,y+1)
        index9=loc(x+1,y+1)
        liste.append(red(img.pixels[index1]))# on remplit la liste avec les composantes rouges des 9 pixels
        liste.append(red(img.pixels[index2]))
        liste.append(red(img.pixels[index3]))
        liste.append(red(img.pixels[index4]))
        liste.append(red(img.pixels[index5]))
        liste.append(red(img.pixels[index6]))
        liste.append(red(img.pixels[index7]))
        liste.append(red(img.pixels[index8]))
        liste.append(red(img.pixels[index9]))
        liste.sort()# on trie la liste
        r=liste[4] # c'est ma médiane
        img2.pixels[index5]=color(r,r,r) # on modifie la deuxième image
#updatePixels()
image(img2,0,0)
image(img,0,0,127,127)
```



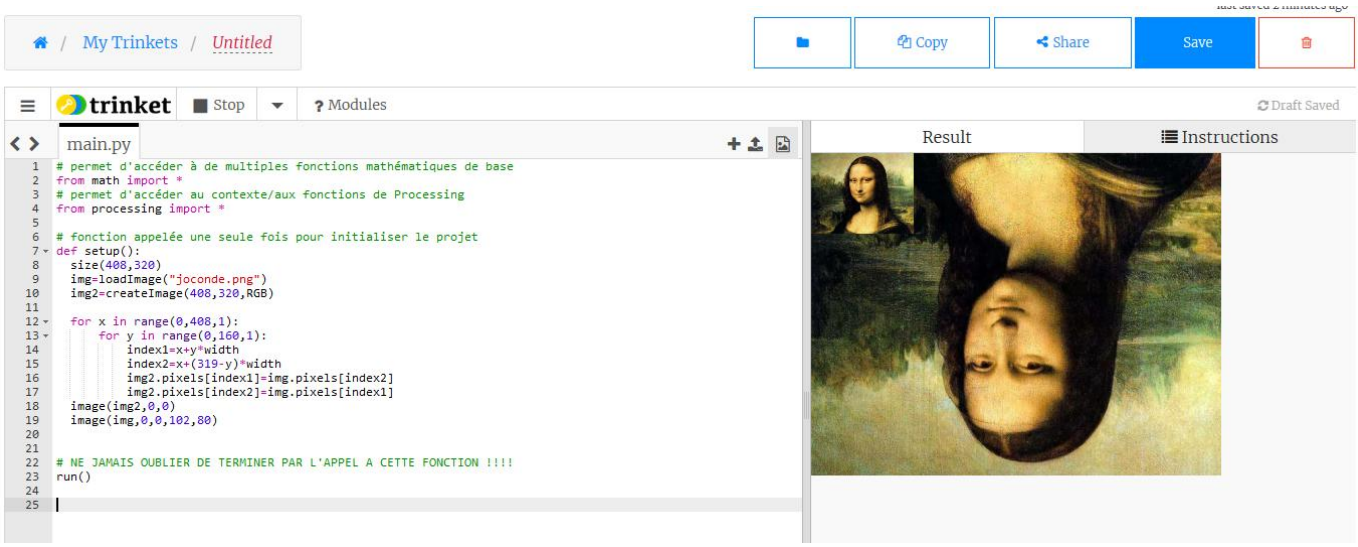
Comment améliorer ce résultat ?

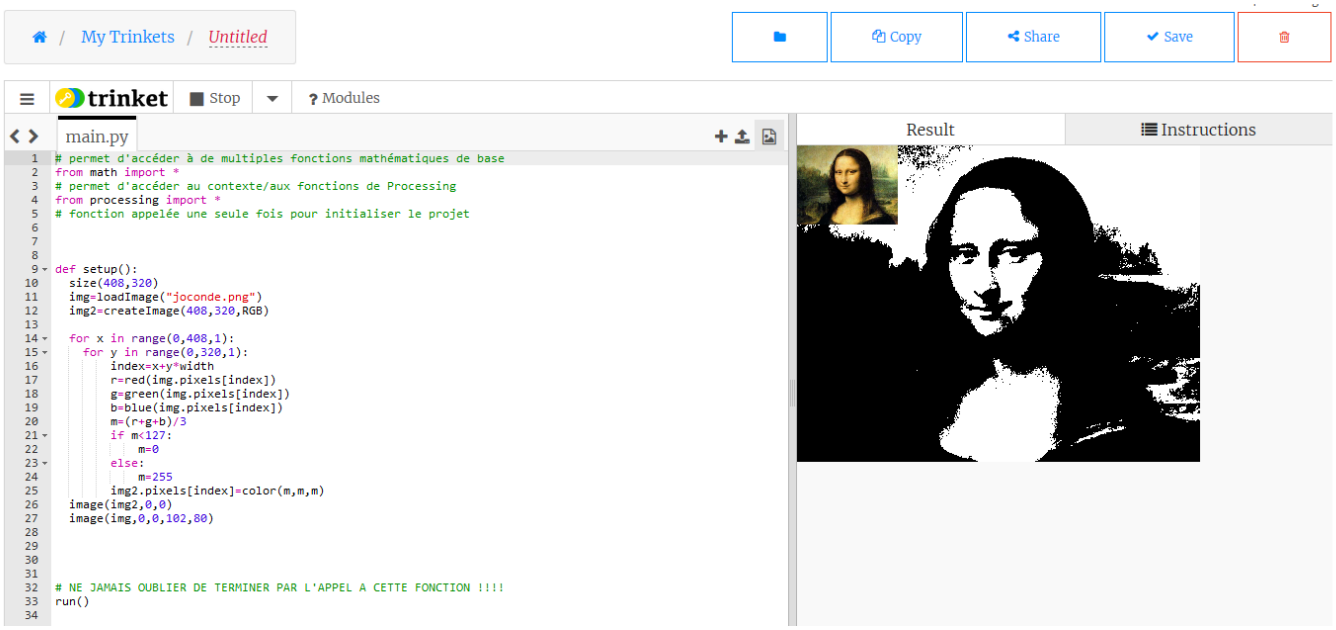
Travailler avec Processing dans un environnement Python

Tous les programmes présentés sont réalisés dans Processing qui "encapsule un mode Python", mais on peut faire l'inverse, travailler dans Python et "encapsuler Processing".
En effet la bibliothèque processing existe.

Processing produit des programmes en .pyde (donc exécutables dans processing) tandis qu'avec Python et la bibliothèque processing, les programmes sont avec l'extension .py

J'ai testé cette bibliothèque avec Trinket (environnement Python en ligne), et cela fonctionne, il faut faire bien attention à écrire les programmes dans la fonction setup(), sinon cela ne fonctionne pas





Un site qui présente cette bibliothèque