DESSIN PIXEL PAR PIXEL

TP

Dans ce tp on utilisera les librairies TKinter et Pillow.

```
from tkinter     import Tk, Canvas
from PIL.Image   import new
from PIL.ImageTk import PhotoImage
```

Partie I — Pixels

On dispose de trois boîtes noires. Commençons par définir (dans deux variables globales) les dimensions (en pixels) du dessin.

```
WIDTH = 500 ; HEIGHT = 500
```

La première boîte noire est un programme qui crée un dessin (de la taille ci-dessus) tout blanc (et qu'on pourra modifier par la suite, bien sûr).

```
def CréerDessin() :
    return new("RGB", (WIDTH , HEIGHT), "white")
```

La deuxième boîte noire est un programme qui permet d'afficher (et donc de *voir*) un dessin.

```
def Afficher(Dessin) :

    # La fenêtre
    F = Tk() ; F.title("Mon dessin")
    Cnv = Canvas(F, width = WIDTH, height = HEIGHT)
    Cnv.pack()

    # Le dessin
    PhImg_Dessin = PhotoImage(Dessin)
    X = 1 + WIDTH // 2 ; Y = 1 + HEIGHT // 2
    Cnv.create_image((X, Y), image=PhImg_Dessin)

    # Boucle principale
    F.mainloop()
```

Enfin la dernière boîte noire est la fonction `D.putpixel( (px, py), (`r`, `g`, `b`) )` qui, si `D` désigne un dessin, peint le pixel `(px, py)` de la couleur $(r, g, b) \in \{0\,;\,1\,;\,\ldots\,;\,255\}^3$.

Exercice 1 — Les couleurs de base. On donne ceci.

```
ROUGE = (255, 0, 0)
BLANC = (255, 255, 255)
```

Définir `NOIR`, `BLEU`, `VERT`, `JAUNE`, `MAGENTA`, `CYAN`, `VERT_FONCÉ`, `ORANGE`, `VIOLET`, `GRIS`, `ROSE` et `MARRON`.

Exercice 2

a) Créer un dessin et peindre en rouge le pixel de coordonnées (50, 50).

b) Dans quel sens sont dirigés les deux axes de coordonnées ?

Exercice 3 — Carré bleu sur fond noir.

a) Tester le programme ci-dessous.

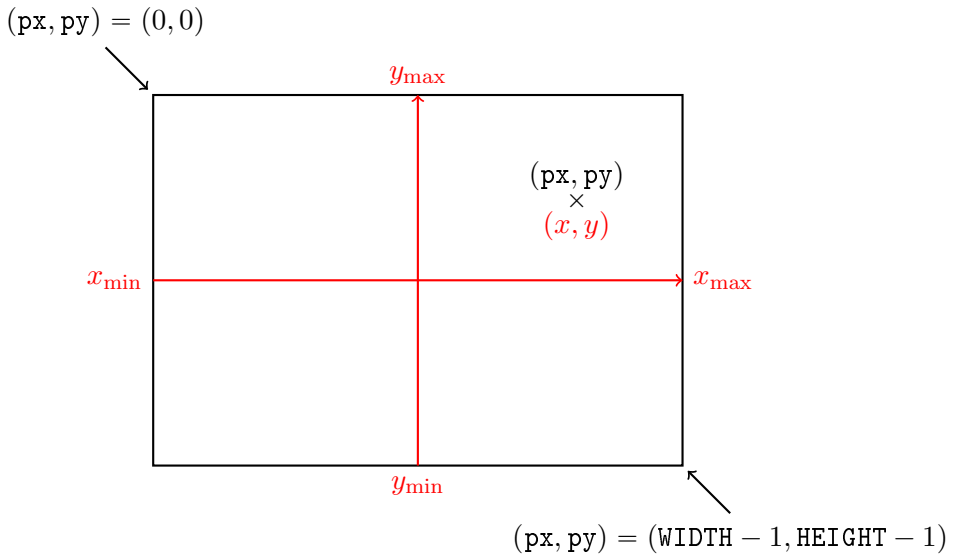
```
def Modèle_1() :

    # Création du dessin
    D = CréerDessin()

    # On peint chaque pixel un par un...
    for px in range(WIDTH) :
        for py in range(HEIGHT) :

            # ...en choisissant une couleur pour chacun
            if 100 <= px <= 200 and 100 <= py <= 200 :
                c = ROUGE
            else :
                c = BLANC

            # Là on peint le pixel de la couleur choisie
            D.putpixel( (px, py), c )

    # Et on affiche le résultat !
    Afficher(D)
```

b) En déduire un programme `CarréBleuSurFondNoir()` qui fait ce qu'on imagine.

Exercice 3 — Modifier le programme précédent pour qu'il peigne le carré dans un dégradé allant du magenta au vert.

Partie II — Coordonnées géométriques

Pour faire des dessins plus sophistiqués, il est préférable d'abandonner les pixels (qui outre la position baroque des axes sont obligatoirement des entiers) au profit de coordonnées plus habituelles, et avec lesquelles on pourra faire tous les calculs qu'on veut (de la trigonométrie par exemple).



On introduit donc deux nouveaux axes, définis par leurs intersections avec les bords de la fenêtre : chaque point aura donc deux jeux de coordonnées : les pixels $(\texttt{px}, \texttt{py})$ et des coordonnées (que nous appellerons *géométriques*) (x, y).

```
x_min = -2
x_max =  2
y_min = -2
y_max =  2
```

La première chose à faire est d'écrire les programmes qui permettent de passer d'un système de coordonnées à l'autre. Voici d'abord le sens qui nous sera utile.

```
def PixelVersPoint(px, py) :
    x = x_min + px / (WIDTH - 1) * (x_max - x_min)
    y = y_max - py / (HEIGHT - 1) * (y_max - y_min)
    return (x, y)
```

Et à titre indicatif voici la conversion inverse, que nous n'utiliserons pas aujourd'hui.

```
def PointVersPixel(x, y) :
    px = int( (x - x_min) / (x_max - x_min) * (WIDTH - 1) )
    py = int( (y_max - y) / (y_max - y_min) * (HEIGHT - 1) )
    return (px, py)
```

Exercice 4 — Justifier les formules des deux programmes ci-dessus.

Exercice 5 — Carré vert sur fond vert.

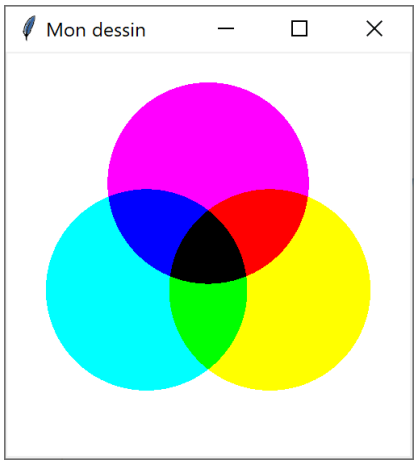
a) Tester le programme ci-dessous.

```
def Modèle_2() :

    # Création du dessin
    D = CréerDessin()

    # On peint chaque pixel un par un
    for px in range(WIDTH) :
        for py in range(HEIGHT) :

            # On détermine les coordonnées géométriques
            (x, y) = PixelVersPoint(px, py)

            # On choisit la couleur
            if -1 <= x <= 1 and -1 <= y <= 1 :
                c = ROUGE
            else :
                c = BLANC

            # Là on peint le pixel de la couleur choisie
            D.putpixel( (px, py), c )

    # Et on affiche le résultat !
    Afficher(D)
```

b) En déduire un programme `CarréVertSurFondVert()` qui réalise la même figure, mais en utilisant deux nuances de vert.

Exercice 6 — Écrire un programme qui réalise un dégradé bidirectionnel sur toute la fenêtre : du rouge au cyan sur le bord du haut, du rouge au vert sur le bord gauche.

Exercice 7 — Synthèse soustractive. Écrire le programme qui produit le dessin ci-dessous.

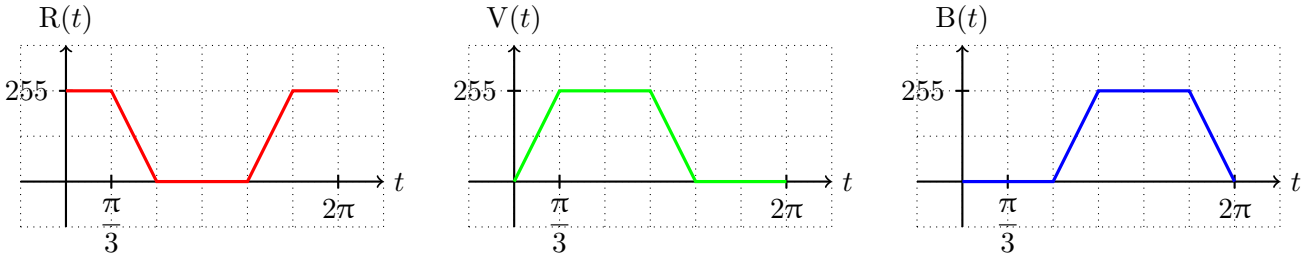


Partie III — Les couleurs

Exercice 8 — Couleurs saturées. On dit qu'une couleur (r, g, b) est *saturée* lorsque sa saturation

$$\mathrm{sat}(r, g, b) = \max(r, g, b) - \min(r, g, b)$$

est maximale, c'est-à-dire ici égale à 255. On va associer à chaque réel t une couleur saturée ; pour cela on définit trois fonctions R, V et B, toutes 2π-périodiques, et dont les courbes (sur la période principale) sont les suivantes.



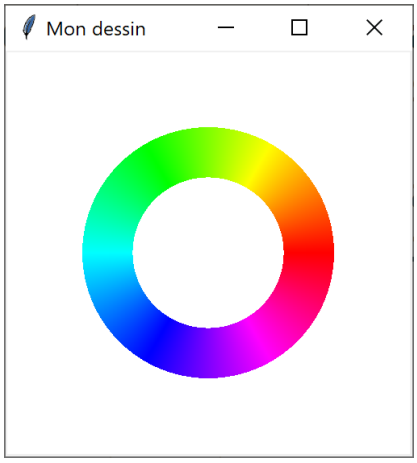
a) Que représente, en Python, `t % (2 * π)` ?

b) Écrire les programmes `Rouge(t)`, `Vert(t)` et `Bleu(t)` qui calculent les trois fonctions ci-dessus. Pour coller à la convention de TKinter, on leur fera renvoyer des `int`.

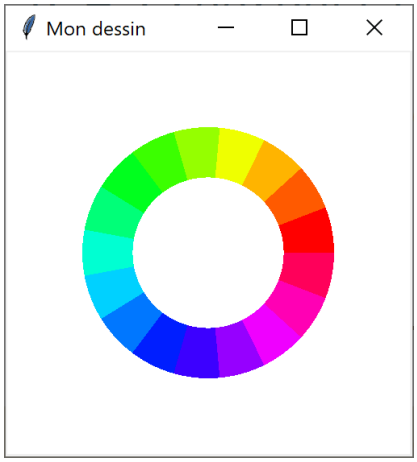
c) Écrire ensuite le programme `Saturée(t)` qui renvoie la couleur $(\mathrm{R}(t), \mathrm{V}(t), \mathrm{B}(t))$.

d) En déduire finalement le programme qui réalise le dessin ci-dessous. On pourra utiliser la petite fonction que voici.

```
from numpy import angle as arg
```



Exercice 9 — Proposer une variante `CercleChromatiqueDiscret(N)` du programme précédent, permettant d'obtenir le dessin ci-dessous (N représentant évidemment le nombre de portions).



Exercice 10 — Couleurs et nombres complexes. Soit z un nombre complexe. Notons t son argument, on sait associer à t une couleur saturée $(\hat{r}, \hat{g}, \hat{b})$. Nous allons assombrir ou éclaircir cette couleur en fonction du module de z.

Si $|z| \leqslant 1$, posons $\ell = |z|$ et

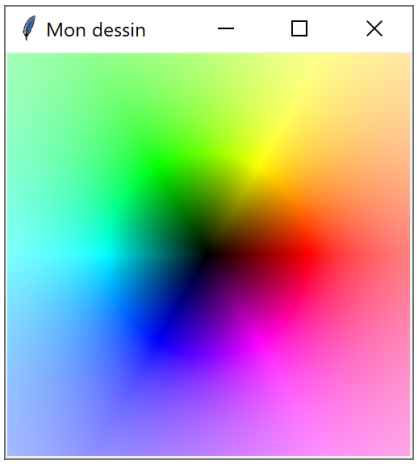
$$r = \lfloor 255 \times \ell\hat{r} \rceil\,, \quad g = \lfloor 255 \times \ell\hat{g} \rceil \quad \text{et} \quad b = \left\lfloor 255 \times \ell\hat{b} \right\rceil.$$

Et si $|z| \geqslant 1$, posons $m = 1 - 1/|z|$ et

$$r = \lfloor 255m + (1-m)\hat{r} \rceil\,, \quad g = \lfloor 255m + (1-m)\hat{g} \rceil \quad \text{et} \quad b = \left\lfloor 255m + (1-m)\hat{b} \right\rceil.$$

a) Écrire le programme `CouleurAssociée(z)` qui renvoie le triplet (r, g, b) associé à un nombre complexe z selon ces règles.

b) En déduire un programme qui peint chaque pixel du plan en fonction de l'affixe à laquelle il se trouve.



Exercice 11 — La fonction exponentielle.

a) Reprendre l'exercice précédent, en peignant cette fois-ci le pixel d'affixe z avec la couleur associée à e^z (on pourra utiliser `exp` du module NumPy).

b) Observer le dessin sur une fenêtre assez large (-10 à 10 sur les deux axes, par exemple). Quelles propriétés de la fonction exponentielle « voit »-t-on ?