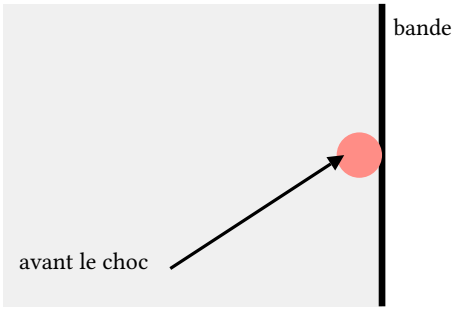
Simuler un gaz avec Pyxel

Doc. 3,4

Objectif

Construire pas à pas avec Pyxel la simulation d'un gaz : chaque particule est un tuple, le gaz est une liste de tuples passée en paramètre aux fonctions.

Q1. Sur un billard, une bille frappe la bande de droite. Le schéma montre sa vitesse juste avant le choc.



Dessiner la flèche de sa vitesse juste après le choc. Lequel des deux mouvements, horizontal ou vertical, est modifié par le choc ?

Dans un gaz, les particules avancent en ligne droite et rebondissent sur les parois comme la bille sur la bande. Le programme de départ est fourni dans Capytale. Quand une question commence par « Sans exécuter le programme », on écrit sa réponse avant de vérifier.

§1. Une particule décrite par un tuple

Une particule est décrite par cinq entiers, rangés dans un tuple, toujours dans le même ordre :

indice	0	1	2	3	4
élément	`x`	`y`	`vx`	`vy`	`couleur`

`x` et `y` sont les coordonnées du centre, en pixels. `vx` et `vy` sont les nombres de pixels parcourus horizontalement et verticalement à chaque image. Un écran n'a pas de demi-pixel : toutes ces valeurs sont des entiers.

```python
import pyxel


# Constantes
LARGEUR = 160    # largeur de la fenêtre, en pixels
HAUTEUR = 120    # hauteur de la fenêtre, en pixels
RAYON = 3        # rayon d'une particule, en pixels
```

```
# Type d'une particule : (x, y, vx, vy, couleur)
Particule = tuple[int, int, int, int, int]

def dessiner_particule(p: Particule) -> None:
    """Dessine la particule p = (x, y, vx, vy, couleur)."""
    x, y, couleur = p[0], p[1], p[4]
    pyxel.circ(x, y, RAYON, couleur)

def update(gaz: list[Particule]) -> None:
    """Fait avancer le gaz d'une image."""
    pass

def draw(gaz: list[Particule]) -> None:
    """Dessine les particules du gaz."""
    pyxel.cls(0)
    dessiner_particule(gaz[0])

# Programme principal
pyxel.init(LARGEUR, HAUTEUR, title="Gaz parfait")
particules = [(36, 60, 2, -1, 8)]     # le gaz : une seule particule pour l'instant
pyxel.run(lambda: update(particules), lambda: draw(particules))
```

La ligne `Particule = tuple[...]` donne un nom au type d'une particule, que l'on utilise dans les annotations des fonctions.

`pyxel.run` attend deux fonctions sans paramètre. `lambda: update(particules)` est justement une fonction sans paramètre, écrite sur une ligne : à chaque image, elle appelle `update` en lui passant la liste `particules`.

Règle de ce programme

Une fonction peut lire les **constantes**, écrites en majuscules (`LARGEUR`, `RAYON`…), car leur valeur ne change jamais. Elle ne lit aucune autre variable du programme principal : la liste des particules lui est toujours passée en paramètre.

Q2. Lancer le programme. Avec `particules = [(36, 60, 2, -1, 8)]` :

1. que vaut `particules[0][2]` ? Quelle grandeur représente-t-il ?
2. que faut-il écrire pour obtenir la couleur de la particule ?
3. la particule va-t-elle monter ou descendre quand elle bougera ? Justifier à l'aide du repère de Pyxel (Doc. 2,5).

Q3. Modifier `particules` pour obtenir une particule verte (couleur 11), au centre de l'écran, qui se déplacera vers la gauche de 1 pixel par image. Vérifier, puis revenir à la valeur de départ.

§2. Faire avancer la particule

Pyxel affiche 30 images par seconde et appelle `update(particules)` avant chaque image. À chaque appel, on ajoute `vx` à `x` et `vy` à `y`.

Q4. **Sans exécuter le programme.** La particule part de $x = 36$ avec `vx = 2`. Elle touche le bord droit quand son centre atteint $x = 159 - 3 = 156$. Au bout de combien de temps ?

Un élève propose :

```python
def deplacer(p: Particule) -> None:
    p[0] = p[0] + p[2]
    p[1] = p[1] + p[3]
```

Q5. **Sans exécuter le programme**, prévoir ce qui se passe si on appelle `deplacer(particules[0])`. Expliquer.

La fonction doit donc fabriquer un nouveau tuple et le retourner. Pour récupérer d'un coup les cinq éléments de `p`, on le **déballe** :

```python
x, y, vx, vy, couleur = p      # x reçoit p[0], y reçoit p[1], etc.
```

Il faut autant de noms à gauche que d'éléments dans le tuple.

Q6. Écrire la fonction `deplacer` :

```python
def deplacer(p: Particule) -> Particule:
    """Retourne la particule p après une image de déplacement."""
```

Tester dans la console : `deplacer((36, 60, 2, -1, 8))` doit retourner `(38, 59, 2, -1, 8)`.

Q7. Dans `update`, remplacer `pass` par `gaz[0] = deplacer(gaz[0])`. **Sans exécuter le programme**, prévoir ce qui se passera au bout de 2 secondes, puis vérifier.

Q8. Un autre élève écrit dans `update` : `gaz = [deplacer(gaz[0])]`. Il n'obtient aucune erreur, mais la particule reste immobile. Expliquer, en s'aidant des fonctions `reinitialiser_1` et `reinitialiser_2` du Doc. 3,2.

§3. Rebondir contre les murs

Comme sur le billard, quand la particule touche le mur de droite, `vx` devient `-vx`. On complète `deplacer` : elle change d'abord la vitesse si un mur est touché, puis fait avancer la particule. Première version, qui ne traite que le mur de droite :

```python
def deplacer(p: Particule) -> Particule:
    """Retourne la particule p après une image : sa vitesse change
    si elle touche un mur, puis elle avance."""
    x, y, vx, vy, couleur = p
    if x + RAYON >= LARGEUR - 1:      # le bord droit du disque touche le mur
        vx = -vx
    return (x + vx, y + vy, vx, vy, couleur)
```

Q9. Compléter `deplacer` pour les murs de gauche, du haut et du bas, puis tester.

Coups de pouce

1. Pour les murs du haut et du bas, quelle composante de la vitesse change de signe ?
2. Le mur du haut est la ligne $y = 0$. Quel point du disque l'atteint en premier ?

Q10. **Sans ordinateur.** On part de `particules = [(158, 60, 1, 0, 8)]`. Compléter le tableau avec les valeurs de `x` et `vx` après chaque appel de `update`. Que fait la particule ? Vérifier à l'écran.

image	départ	1	2	3	4
x	158				
vx	1				

Q11. Modifier les quatre conditions pour qu'une particule ne rebondisse que si elle se dirige vers le mur qu'elle touche. Refaire le tableau sans ordinateur, puis vérifier.

À retenir

Un tuple n'est pas mutable : une fonction qui « modifie » une particule fabrique et retourne un nouveau tuple. Le déballage `x, y, vx, vy, couleur = p` donne un nom à chaque élément.

Lors d'un rebond sur une paroi, la composante de la vitesse perpendiculaire à la paroi change de signe.

Une fonction qui reçoit une liste en paramètre peut modifier ses éléments (`gaz[0] = ...`). Si elle affecte une nouvelle valeur au paramètre (`gaz = ...`), la liste d'origine ne change pas.

§4. Un gaz : une liste de particules

On veut maintenant 30 particules, tirées au hasard. `pyxel.rndi(a, b)` retourne un entier au hasard entre `a` et `b`, bornes comprises. Ajouter ces constantes au début du programme (8, 11 et 12 sont le rouge, le vert et le bleu) :

```python
NB_PARTICULES = 30          # nombre de particules du gaz
VITESSE_MAX = 2             # valeur maximale d'une composante de la vitesse
COULEURS = (8, 11, 12)      # couleurs possibles
```

Q12. Écrire la fonction `nouvelle_particule`, puis la tester plusieurs fois dans la console :

```python
def nouvelle_particule() -> Particule:
    """Retourne une particule entièrement visible, tirée au hasard.
    Chaque composante de la vitesse est comprise entre -VITESSE_MAX
    et VITESSE_MAX. La couleur est l'un des éléments de COULEURS."""
```

Coups de pouce

1. Un disque est entièrement visible si son centre est à au moins `RAYON` pixels de chaque bord (Doc. 2,5).
2. Pour la couleur, tirer au hasard l'**indice** d'un élément de `COULEURS`.

Q13. Dans le programme principal, remplacer `particules = [(36, 60, 2, -1, 8)]` par une liste de `NB_PARTICULES` particules construite **en compréhension** (Doc. 3,2). Pourquoi cette ligne doit-elle se trouver après la définition de `nouvelle_particule` ?

Q14. Modifier `draw` pour dessiner toutes les particules de `gaz`.

Pour faire bouger toutes les particules, un élève écrit :

```python
def update(gaz: list[Particule]) -> None:
    for p in gaz:
        p = deplacer(p)
```

Q15. **Sans exécuter le programme**, prévoir ce qu'on observe. Expliquer, puis corriger.

À retenir

Le gaz est une **liste de tuples**, construite en compréhension. Pour lire ses éléments, on la parcourt par éléments. Pour les remplacer, on la parcourt par indices : `gaz[i] = ...`.

§5. Pour aller plus loin : des collisions entre particules

Dans un gaz réel, les particules se heurtent aussi entre elles. Deux particules se touchent si la distance entre leurs centres est inférieure ou égale à `2 * RAYON`.

Q16. Ajouter `from math import sqrt` au début du programme, puis écrire :

```python
def distance(p1: Particule, p2: Particule) -> float:
    """Retourne la distance entre les centres de p1 et p2."""


def se_touchent(p1: Particule, p2: Particule) -> bool:
    """Retourne True si p1 et p2 se touchent, False sinon."""
```

Tester : `distance((0, 0, 1, 1, 8), (3, 4, 0, 0, 12))` doit retourner `5.0`.

Deux particules de même masse qui se heurtent de face **échangent leurs vitesses**. On adopte ce modèle pour toutes les collisions.

Q17. Écrire la fonction `choc` :

```python
def choc(p1: Particule, p2: Particule) -> tuple[Particule, Particule]:
    """Retourne les particules p1 et p2 après leur collision :
    elles ont échangé leurs vitesses."""
```

Tester : `choc((50, 60, 1, 0, 8), (55, 60, -1, 0, 12))` doit retourner `((50, 60, -1, 0, 8), (55, 60, 1, 0, 12))`.

Voici la fonction qui teste toutes les paires de particules de la liste `gaz`. Sa dernière ligne déballe le tuple retourné par `choc`, comme `a, b = b, a` (Doc. 3,1).

```python
def gerer_chocs(gaz: list[Particule]) -> None:
    """Fait échanger leurs vitesses aux particules de gaz qui se touchent."""
    for i in range(len(gaz)):
        for j in range(i + 1, len(gaz)):
            p1 = gaz[i]
            p2 = gaz[j]
            if se_touchent(p1, p2):
                gaz[i], gaz[j] = choc(p1, p2)
```

Q18.
1. Avec 3 particules, écrire tous les couples `(i, j)` testés.
2. Pourquoi la boucle intérieure commence-t-elle à `i + 1` et non à `0` ?

Q19. Ajouter l'appel `gerer_chocs(gaz)` à la fin de `update`. Observer longuement : certaines particules restent collées. Pourquoi ? Corriger, en s'inspirant de la correction des rebonds sur les murs.

> 🔥 **Coups de pouce**
>
> 1. Que fait `gerer_chocs` à l'image qui suit un échange de vitesses ?
> 2. Sur le mur, on n'a fait rebondir que les particules qui s'en approchaient. Comment savoir si deux particules se rapprochent ? On peut utiliser `deplacer` et `distance`.

Il reste un défaut : des particules rapides se traversent parfois.

Q20. **Sans ordinateur.** Deux particules arrivent de face, en diagonale : `(50, 50, 2, 2, 8)` et `(56, 56, -2, -2, 12)`. À chaque image, `update` les déplace, puis appelle `gerer_chocs`. Compléter le tableau (distances arrondies au dixième), puis expliquer pourquoi les particules se traversent.

image	départ	1	2	3
particule 1	(50, 50)			
particule 2	(56, 56)			
distance	8,5			
`se_touchent`	non			
`se_rapprochent`	oui			

Pour éviter ce chevauchement, il faut anticiper : **avant** de déplacer les particules, on regarde si elles vont se toucher à l'image suivante.

Q21. Modifier la condition de `gerer_chocs` pour tester si les particules se toucheront après leur déplacement, puis placer l'appel `gerer_chocs(gaz)` au début de `update`. Refaire le tableau précédent, puis vérifier à l'écran que les particules ne se traversent presque plus.

§6. Défis

Q22. **Défi A : le mélange de deux gaz.** Au départ, les particules rouges sont dans la moitié gauche de la boîte, les bleues dans la moitié droite. Donner à `nouvelle_particule` les paramètres `couleur`, `x_min` et `x_max`, puis construire la liste. Comparer la vitesse du mélange avec et sans collisions.

Q23. **Défi B : chauffer le gaz.** Chaque appui sur la touche ↑ double les deux composantes de la vitesse de toutes les particules. Que se passe-t-il après plusieurs appuis ?

§7. Bilan

À l'issue de cette activité, je sais…

- représenter un objet par un tuple et lire un élément par son indice ;
- expliquer pourquoi on ne peut pas modifier un élément d'un tuple, et écrire une fonction qui retourne un nouveau tuple ;
- déballer un tuple pour nommer ses éléments ;
- construire une liste de tuples en compréhension ;
- choisir entre un parcours par éléments, pour lire une liste, et un parcours par indices, pour la modifier ;
- passer une liste en paramètre à une fonction, et expliquer pourquoi la fonction peut modifier ses éléments mais pas la remplacer ;
- parcourir toutes les paires d'éléments d'une liste avec deux boucles imbriquées ;
- faire tourner un programme à la main pour trouver l'origine d'un comportement inattendu.