PyDraw

Killian HARROIS,Anthony Voisin, Houmame Lachache,Afdali Omar,Antonin BÔ

January 2025

Contents

1 Syntaxe de PyDraw

1.1 Structure de base

1.1.1 Type de données supportés

PyDraw prend en charge les types suivants :

- `int` : Nombres entiers.
- `float` : Nombres à virgule.
- `bool` : Valeurs booléennes (true ou false).
- `string` : Chaînes de caractères.
- `pen` : Objet graphique manipulable qui sert à dessiner.

1.1.2 Retour à la ligne

Chaque instruction doit se terminer par un point-virgule (';').

1.2 Déclarations

1.2.1 Variables

Les variables sont déclarées avec leur type suivi de leur nom et éventuellement d'une valeur initiale :

```
int entier = 56;
float deci = 3.14;
int i = 0;
```

1.2.2 Fonctions

Une fonction est déclarée avec le mot-clé `func`, suivi de son type de retour, du nom et des paramètres entre parenthèses :

```
func void test(int a, float b) {
    // Corps de la fonction
}
```

1.3 Opérations

1.3.1 Opérations supportées

PyDraw permet les opérations suivantes :

- `+` : Addition.
- `-` : Soustraction.
- `*` : Multiplication.
- `/` : Division.

1.3.2 Affectation et incrémentation

- `=` : Affecte une valeur.
- `++` : Incrémente (`j = j + 1`).
- `-` : Décrémente (`j = j - 1`).

1.3.3 Opérateurs de comparaison

- `==` : Égalité.
- `!=` : Inégalité.
- `<=`, `>=` : Infériorité ou égalité, supériorité ou égalité.
- `<`, `>` : Infériorité stricte, supériorité stricte.

1.3.4 Opérateurs logiques

- `&` : ET logique.
- `|` : OU logique.

1.4 Structures de contrôle

1.4.1 Conditions

Les confitions utilisent les mots-clés `if`, `elseif`, et `else` :

```
if(entier == 55) {
    // Instructions
} elseif(deci == 3.14) {
    // Instructions
} else {
    // Instructions
}
```

1.4.2 Boucle

- `repeat` : Boucle avec initialisation, condition et incrémentation :

```
repeat(i, i < 10, i++) {
    // Instructions
}
```

Mots-clés pour contrôler les boucles :

- `skip` : Passe à l'itération suivante.
- `leave` : Quitte la boucle.

1.5 Objets `pen`

1.5.1 Déclaration

Un `pen` est initialisé avec la fonction `cursor`, qui prend les coordonnées initiales :

```
pen stylo1 = cursor(100, 200);
```

1.5.2 Attributs

Les `pen` possèdent les attributs suivants :

- `color` : (=defineColor(string code hexadecimal) cet attribut change la couleur du pen. Il faut passer par la fonction defineColor car SDL a besoin d'un code RGB et cette fonction traduit le code hexadecimal en RGB.
- `thickness` : (int >= 1) Epaisseur du pen.
- `rotation` : (float) Change l'angle vers lequel pointe le pen
- `penDown` : (int 1 ou 0) Lève ou baisse le pen (1 = stylo baissé qui dessine / 0 = stylo levé qui ne dessine pas)

1.5.3 Méthodes

- `walk(int n>=0)` : Avance d'une distance donnée.
- `goTo(int x, int y)` : Va à une position sans dessiner.
- `circle(int r>=1)` : Dessine un cercle de rayon r.

Pour utiliser une méthode : [nomPen].[nomMethode](parametres);
Exemple : superStylo.circle(250);

1.5.4 Fonctions incluses

Ces fonctions permettent d'interagir directement avec la zone de dessin :

- `fillColor(int x, int y, string c)` : Se rend au pixel de position (x,y) et change la couleur de tous les pixels adjacents qui sont de la même couleur, en la couleur passee en parametre (code hexadecimal)
- `translation(int x, int y, int l, int h, int distance, int precision > 0)` : Se rend au pixel de position (x,y), selectionne une zone de longeur l et hauteur h, effectue une translation de distance pixel en prenant en compte la rotation. Plus la precision est elevee, plus la translation effectuera de copie colle de la zone selectionnee.
- `rotateArea(int x, int y, int l, int h, int rotation)` : Se rend au pixel (x,y), selectionne une zone de longueur l et hauteur h et effectue une rotation de cette zone.
- `waitKey()` : fait une pause dans l'execution du script. Pour reprendre le script, cliquer sur une touche du clavier

- `clearMatrix(string c)` : nettoie toute la zone de dessin en un fond de la couleur passee en paramètre (code hexadecimal).

2 IDE

Dans cette partie, nous allons expliquer les fonctionnalitées de l'IDE. Nous verrons aussi à travers certains bouts de codes comment marchent ces fonctionnalitées.

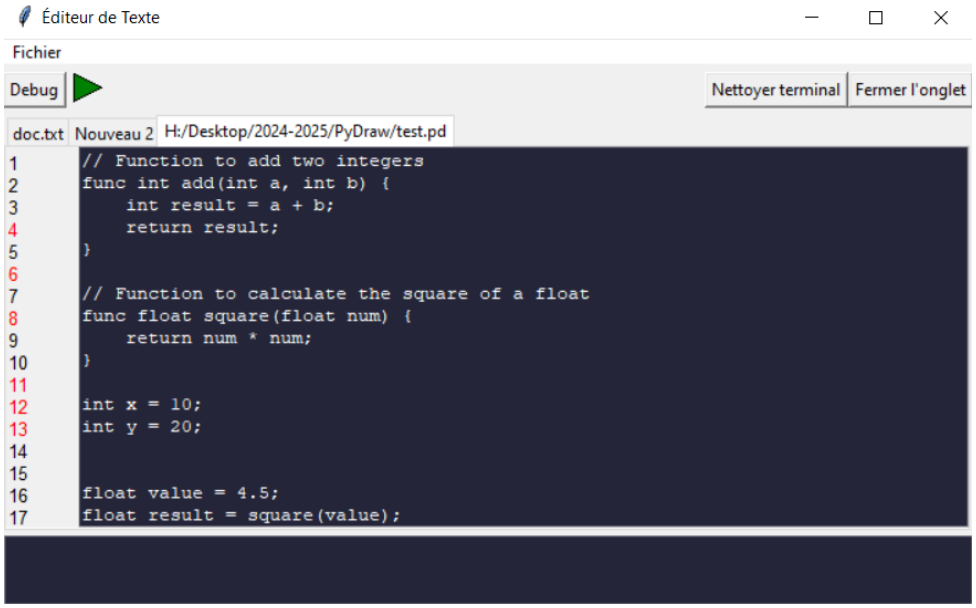


Figure 1: IDE.

Tout d'abord, observons les import utilisées pour réaliser cette interface :

- `tkinter` : c'est une bibliothèque qui permet de réaliser des interfaces graphiques sur python
- `platform` : ceci nous permet de détecter l'OS de la machine (nécessaire pour certaines opérations)
- `compiler.main` : ceci est le compilateur que nous avons créer qui va vérifier si il y a des erreurs dans le code PyDraw écrit par l'utilisateur

2.1 Interface utilisateur

2.1.1 Barre de menu

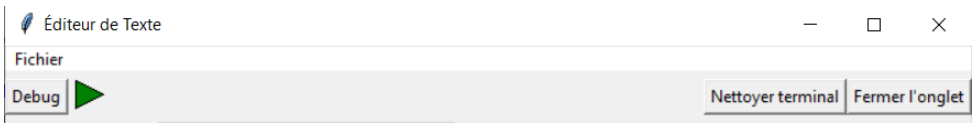


Figure 2: Barre de menu de l'IDE.

La barre de menu se situe sur la partie supérieure de l'IDE. Elle est composée de plusieurs boutons qui permettent d'effectuer diverses actions.

Le menu déroulant "fichier" : En cliquant sur le bouton, un menu déroulant apparaît avec les boutons suivants :

- `Nouveau` : Créer une nouvelle zone de code.
- `Ouvrir` : Permet à l'utilisateur d'ouvrir un fichier .pd, déjà dans son ordinateur.
- `Enregistrer` : Permet de sauvegarder les modifications effectuées sur le fichier actuel.
- `Enregistrer sous` : Permet d'enregistrer le fichier actuel sur l'ordinateur de l'utilisateur.
- `Changer de thème` : Permet de passer du thème sombre au thème clair et inverse.

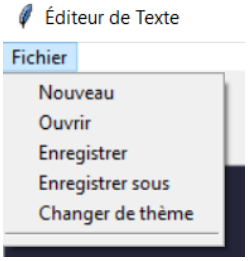


Figure 3: Menu déroulant "fichier".

Debug : En cliquant sur ce bouton, le programme s'éxécute en mode debug (plus de détail en bas).

Run (triangle vert) : Ce bouton sert à exécuter le code.

Nettoyer terminal : Ce bouton sert à nettoyer le terminal.

Fermer l'onglet : Ce bouton sert à fermer le script actuellement selectionné.

2.1.2 Terminal

Le terminal sert à afficher des messages d'erreurs si le code est incorrect, ou bien un message indiquant qu'il n'y a pas de problèmes.

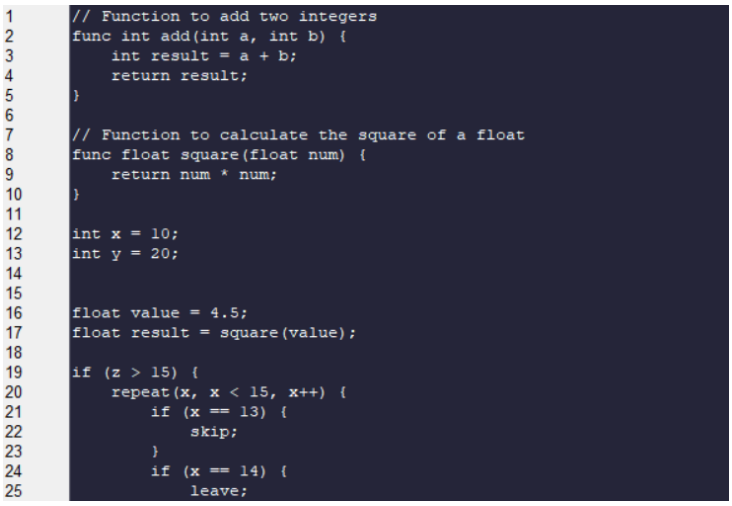
2.1.3 Zone d'édition du code

C'est dans cette zone que l'utilisateur doit écrire le code PyDraw. Si l'utilisateur exécute un code qui comporte des erreurs, les lignes erronées seront soulignés.

2.2 Gestion des fichiers

2.2.1 Fonctionnalitées

L'IDE prend en charge plusieurs fonctionnalitées relatives à la gestion de fichier :



```
// Function to add two integers
func int add(int a, int b) {
    int result = a + b;
    return result;
}

// Function to calculate the square of a float
func float square(float num) {
    return num * num;
}

int x = 10;
int y = 20;


float value = 4.5;
float result = square(value);

if (z > 15) {
    repeat(x, x < 15, x++) {
        if (x == 13) {
            skip;
        }
        if (x == 14) {
            leave;
```

Figure 4: Zone d'édition du code

Fichier de documentation : Au lancement de l'IDE, un fichier de documentation est affiché. Il sert à guider l'utilisateur sur l'utilisation du langage PyDraw. Ce fichier ne peut pas être modifié ni fermé par l'utilisateur.

Ouvrir un fichier : L'utilisateur peut ouvrir un fichier .pd déjà enregistré sur son ordinateur. En cliquant sur "ouvrir", une nouvelle page va s'ouvrir, permettant à l'utilisateur de choisir un fichier .pd.

Nouveau fichier : En cliquant sur "nouveau", un nouveau script va se créer. Par défaut, ce script s'appellera "Nouveau", mais il est possible de changer son nom lors de sa sauvegarde.

Sauvegarde d'un fichier : L'utilisateur peut soit faire "enregistrer sous" pour sauvegarder un nouveau fichier, ou simplement "enregistrer" pour sauvegarder les modifications apportées à un fichier déjà enregistré.

En cliquant sur "enregistrer sous", une nouvelle fenètre va s'ouvrir pour permettre à l'utilisateur de sélectionner l'emplacement et le nom du fichier qu'il souhaite enregistrer.

En cliquant sur "enregistrer", cela va simplement sauvegarder les modifications apportées au fichier actuellement affiché. Si l'utilisateur clique sur "enregistrer" pour un nouveau fichier qu'il n'a pas encore enregistré sur son ordinateur, cela va aussi afficher une nouvelle page pour d'abord lui permettre d'enregistrer le fichier sur son ordinateur (comme si il faisait "enregistrer sous").

```python
def save_file(self):
    #select the current fenester
    current_tab = self.notebook.index(self.notebook.select())
    if self.open_files[current_tab]['file_path']:
        with open(self.open_files[current_tab]['file_path'], 'w') as file:
            #save the file
            file.write(self.open_files[current_tab]['text_widget'].get(1.0, tk.END))
    #if it's a new file --> do "save as" instead
    else:
        self.save_file_as()
```

Figure 5: Fonction save_file

Ajout de la nouvelle fenêtre sur l'interface : Lorsque l'utilisateur veut ouvrir ou créer un fichier, il faut ensuite afficher la nouvelle fenêtre sur l'interface pour qu'il puisse la visualiser et intéragir avec. C'est pourquoi, dans chacune des fonctions du code Python qui servent à ouvrir une nouvelle fenêtre (fichier de documentation, nouveau fichier, ouvrir un fichier déjà existant) il faut :

- `créer une nouvelle fenêtre dans l'IDE`
- `ajouter le numéro des lignes`
- `récolter les commandes de l'utilisateur` : Tkinter ne détecte pas automatiquement les actions de l'utilisateur sur l'interface (scroll, clic avec la souris ...).

```python
# add a new window
text_frame = tk.Frame(self.notebook)
text_widget = tk.Text(text_frame, wrap='word', bg='#252539', fg='white')  # Couleurs du thème
text_widget.pack(side='right', fill='both', expand=True)

# add line numbers
line_numbers = tk.Canvas(text_frame, width=50)
line_numbers.pack(side='left', fill='y')

# collect all of the user's input
text_widget.bind('<KeyRelease>', lambda event: self.update_line_numbers(text_widget, line_numbers))
text_widget.bind('<Configure>', lambda event: self.update_line_numbers(text_widget, line_numbers))
text_widget.bind('<MouseWheel>', lambda event: self.on_mouse_wheel(event, text_widget, line_numbers))
text_widget.bind('<Button-4>', lambda event: self.on_mouse_wheel(event, text_widget, line_numbers))
text_widget.bind('<Button-5>', lambda event: self.on_mouse_wheel(event, text_widget, line_numbers))
line_numbers.bind('<MouseWheel>', lambda event: self.on_mouse_wheel(event, text_widget, line_numbers))

self.notebook.add(text_frame, text=new_file_name)
self.open_files.append({'text_widget': text_widget, 'line_numbers': line_numbers, 'file_path': None, 'frame': text_frame, 'closable': True})

## after loading the file, update line number
self.update_line_numbers(text_widget, line_numbers)
```

Figure 6: Bloc de code qui ajoute la nouvelle fenêtre

Synchroniser les lignes du code avec les numéros de lignes : La zone d'édition du code et la zone avec les numéros de lignes sont 2 zones différentes, par conséquent lors du scroll dans l'une des 2, il faut synchroniser l'autre afin que les lignes restent bien allignées avec leur numéro.

Comme on peut le voir dans la figure 6, lorsque l'utilisateur scroll, cela va appeller la fonction "on_mouse_wheel".

Dans cette fonction, on utilise la bibliothèque "platform" afin de détecter l'OS de l'utilisateur. En effet, la gestion du scroll ne fonctionne pas de la même façon sur Linux que sur Windows.

```python
def on_mouse_wheel(self, event, text_widget, line_numbers):
    if platform.system() == "Linux":
        # for Linux
        if event.num == 4:
            text_widget.yview_scroll(-1, 'units')
        elif event.num == 5:
            text_widget.yview_scroll(1, 'units')
    else:
        # for others OS
        text_widget.yview_scroll(int(-1 * (event.delta / 120)), 'units')

    # Syncrhonize script line with line numbers
    line_numbers.yview_moveto(text_widget.yview()[0])
    self.update_line_numbers(text_widget, line_numbers)
```

Figure 7: Fonction on_mouse_wheel

Ensuite, à l'aide d'une fonction "update_line_numbers", on met à jour les numéros affichés en fonction du scroll de l'utilisateur.

2.3 Mode debug

L'utilisateur peut passer en mode debug. Le mode debug permet d'effectuer des pauses lors de l'éxécution de son script. Voici comment l'utiliser :

L'utilisateur doit cliquer sur les numéros de lignes où il souhaite faire une pause pendant l'éxécution. Une fois qu'une ligne est sélectionnée, son numéro devient rouge.

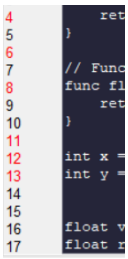


Figure 8: Numéros rouges pour le mode debug

Pour lancer l'éxécution en mode debug, l'utilisateur doit cliquer sur le bouton "Debug". Le code va alors s'éxécuter en effectuant des pauses lorsque les lignes séléctionnées sont éxécutés. Pour reprendre l'éxécution lors d'une pause, l'utilisateur doit appuyer sur n'importe quelle touche du clavier.

```python
def debug(self):
    # select current window
    current_tab = self.notebook.index(self.notebook.select())
    current_file = self.open_files[current_tab]['file_path']
    if current_file :
        if current_file != "doc.txt": # verify that the current window isn't the documentation file
            # run the run function in debug mode (parameter 1)
            underlined_lines = [line_num for line_num, underline in self.line_underlignes[current_file].items() if underline]
            self.run_action(1)
```

Figure 9: Fonction debug

En cliquant sur le bouton debug, cette fonction va séxécuter, et elle va appeller la fonction principale "run_action" qui permet d'éxécuter le script PyDraw. Cependant, elle va passer en paramètre 1 afin de donner l'information que nous éxécutons en mode debug.

2.3.1 Exécution du script

```python
def run_action(self,debug = 0):
    # select current window
    self.save_file()
    current_tab_index = self.notebook.index(self.notebook.select())
    current_tab = self.open_files[current_tab_index]
    file_path = current_tab['file_path']
    text_widget = current_tab['text_widget']
    liste_key = [] #stock the errors returned by the compiler
    on_lines = [] #stock the debug lines
    if debug != 0: # if debug mode, we send to the compiler the lines selected by the user
        on_lines_str = [line_num for line_num, state in self.line_states[file_path].items() if state]
        on_lines = [int(x) for x in on_lines_str]
    if file_path :
        if file_path != "doc.txt": # checking if we're not in the documentation file
            result = main.main(file_path,on_lines) #sending the file path and the debug lines to the compiler
            if result != None :
                for key in result:
                    liste_key.append(key)
                    # underline the lines with an error
                    self.line_underlignes[file_path][key] = True
                    self.update_underlignes(text_widget, file_path)
                    # print in the terminal
                    if key !=0 :
                        self.terminal.insert(tk.END, f"LIGNE : {key}, ERROR : {result[key]}\n")
                    else :
                        self.terminal.insert(tk.END, "Compilation successful! C code has been generated \n")
                        self.c_compiling()
                #remove the underline
                for i in liste_key :
                    self.line_underlignes[file_path][key] = False
            else :
                self.terminal.insert(tk.END, "Compilation successful! C code has been generated \n")
                self.c_compiling()
                for i in liste_key :
                    self.line_underlignes[file_path][key] = False
                self.update_underlignes(text_widget, file_path)
```

Figure 10: Fonction run_action

Cette fonction est appellée lorsque l'utilisateur exécute le code ou lance le mode débug.

- `Récupérer les informations sur la fenètre actuelle`
- `Si le mode débug est activé, envoyer les lignes à débugger au compilateur`
- `Vérifier que ce n'est pas le fichier documentation, et lancer le compilateur`
- `Souligner les lignes qui comportent des erreurs`
- `S'il n'y a pas d'erreur, lancer le fichier "c_compiling" qui va faire le dessin`

3 Main

Dans cette partie, nous explorerons le fonctionnement d'un compilateur et ses principales composantes, comme le lexeur, le parseur, et la gestion des erreurs. Chaque section détaillera un aspect spécifique avec des extraits de code pour illustrer les concepts.

Bibliothèque utilisée : `Re` La bibliothèque `re` nous est proposée avec ses différentes fonctions, permettant essentiellement de rechercher, modifier ou supprimer des expressions via des expressions régulières (regex).

3.1 Définition des Tokens

Dans ce bloc de code, la liste `TOKENS` est définie pour spécifier les différents types de tokens nécessaires à l'analyse lexicale (ou tokenization) dans un langage donné. Chaque élément de cette liste représente un type de token accompagné d'une expression régulière (regex) permettant de le reconnaître. Voici une explication des principaux tokens définis :

`COMMENT` : Capture les commentaires sur une seule ligne, qui commencent par `//` et se terminent à la fin de la ligne (`r"//.*"`).

`KEYWORD` : Identifie les mots-clés spécifiques au langage, comme `int`, `float`, `if`, ou encore des fonctions spécifiques comme `goTo` ou `initSDL`. Ces mots sont entourés de `_` pour garantir qu'ils sont isolés (par exemple, pour éviter de confondre `int` avec `integer`).

`NUMBER` : Représente les nombres, incluant les entiers (positifs et négatifs) et les nombres décimaux. Le préfixe `-?` gère les signes négatifs, et la partie décimale est capturée via `(.+)?`.

`OPERATOR` : Reconnaît les opérateurs courants, comme les opérateurs arithmétiques (`+`, `-`, `*`, `/`), de comparaison (`<`, `>`, `=`) ou logiques (`|`, `||`).

`SYMBOL` : Capture les symboles de ponctuation utilisés dans la syntaxe, comme `{`, `}`, `(`, `)`, `;`, ou `,`.

`BOOL` : Identifie les valeurs booléennes (`true` et `false`).

`STRING` : Représente les chaînes de caractères, qui sont encadrées par des guillemets doubles (`"`).

`PEN_ATTRIBUTE` : Définit les attributs spécifiques à un objet `pen`, comme `color`, `thickness`, `positionX`, etc., montrant un aspect orienté objet dans le langage.

`IDENTIFIER` : Reconnaît les noms de variables ou de fonctions. Un identifiant commence par une lettre ou un souligné (`_`), suivi de lettres, chiffres ou soulignés supplémentaires.

`WHITESPACE` : Capture les espaces blancs, tabulations et sauts de ligne pour les ignorer ou les traiter séparément.

3.2 TOKENIZE

La fonction `tokenize` transforme le code source en une liste de tokens en identifiant les segments correspondant aux motifs définis dans `TOKENS`. Elle ignore les espaces et commentaires, associe chaque token à son type et son numéro de ligne, et lève une erreur si un caractère invalide est rencontré. Une exception personnalisée, `SyntaxErrorWithLine`, est utilisée pour signaler ces erreurs avec des informations supplémentaires, comme le numéro de ligne où l'erreur s'est produite.

3.3 Analyse Syntaxique

Les fonctions définies dans la classe `Parser` permettent de gérer l'analyse syntaxique et la vérification des déclarations dans un programme. Voici une explication des principales fonctions :

`find_variable_type(name)` : Retourne le type d'une variable déclarée. Vérifie dans les variables locales ou globales. Si la variable est de type `pen`, elle est interprétée comme `int`. Lève une exception si la variable n'est pas trouvée.

`find_variable(name)` : Vérifie si une variable existe et retourne son nom si elle est déclarée. Sinon, lève une exception.

`is_variable_declared(name)` : Vérifie si une variable est déclarée dans la portée locale ou globale. Retourne `True` si trouvée, sinon `False` .

`is_function_declared(name)` : Vérifie si une fonction est déclarée. Retourne `True` si trouvée, sinon `False` .

`declare_variable(name, var_type)` : Déclare une nouvelle variable dans la portée locale ou globale. Lève une exception en cas de redéclaration.

`get_line()` : Retourne le numéro de ligne du jeton actuel pour faciliter la gestion des erreurs.

`declare_function(name, return_type, params)` : Déclare une nouvelle fonction avec son nom, type de retour, et paramètres. Lève une exception si la fonction est déjà déclarée.

`find_function(name)` : Retourne les détails d'une fonction (type de retour et paramètres) si elle est déclarée, sinon lève une exception.

`consume(self, expected_type)` : Vérifie que le token actuel correspond au type attendu (`expected_type`). Si c'est le cas, avance au prochain token et retourne la valeur du token consommé. Si le type ne correspond pas ou si la fin des tokens est atteinte de manière inattendue, une exception est levée avec des détails sur l'erreur.

`parse_variable_declaration(self)` : Analyse une déclaration de variable. Vérifie d'abord le type de la variable (mot-clé comme `int`) et son nom. Ensuite, gère deux cas possibles : - Une déclaration seule, terminée par un point-virgule. - Une déclaration avec initialisation, utilisant = pour affecter une valeur. La variable est ensuite ajoutée au contexte approprié.

`parse_pen_declaration(self)` : Analyse une déclaration spécifique au type `pen`. Vérifie que le nom de la variable n'existe pas déjà, puis attend une initialisation avec `cursor(x, y)`. Les coordonnées `x` et `y` peuvent être des nombres ou des variables existantes. Après validation, la déclaration est enregistrée avec le type `pen`.

`parse_number_or_variable(self)` : Analyse un token pouvant représenter un nombre ou une variable. Si c'est un nombre, retourne sa valeur et son type (`int` ou `float`). Si c'est une variable, vérifie qu'elle est déclarée et de type numérique (`int`, `float`, ou `pen`) avant de retourner son nom et son type. Une exception est levée si ce n'est ni un nombre ni une variable valide.

`parse_pen_method(self, pen_name)` : Analyse l'appel de méthodes spécifiques sur un objet `pen`. Vérifie que l'objet est déclaré, identifie la méthode appelée, et parse ses paramètres. Retourne un nœud AST représentant l'appel.

`parse_pen_call(self, pen_name)` : Analyse l'accès aux attributs d'un objet `pen`. Vérifie que l'objet est déclaré et retourne un nœud AST représentant l'attribut accédé.

`parse_pen_attribute(self, pen_name)` : Analyse l'affectation d'un attribut spécifique à un objet `pen`. Vérifie l'attribut, parse la valeur assignée, et retourne un nœud AST représentant cette affectation.

`parse_function_call(self, name=None)` : Analyse un appel de fonction. Vérifie que la fonction est déclarée, parse les arguments et vérifie leur type. Retourne un nœud AST représentant l'appel.

`parse_function(self)` : Analyse une déclaration de fonction. Parse son type de retour, son nom, ses paramètres et son corps. Vérifie qu'il n'y a pas de redéclaration et retourne un nœud AST.

`parse_repeat(self)` : Analyse une boucle `repeat`. Parse l'initialisation, la condition, l'incrémentation, et le corps de la boucle. Retourne un nœud AST représentant la boucle.

`parse_condition(self)` : Analyse une structure conditionnelle `if/elseif/else`. Parse la condition principale et ses branches, puis retourne un nœud AST.

`parse_condition_methode(self)` : Analyse une méthode de contrôle (`skip`, `leave`, `break`, `wait`). Retourne un nœud AST représentant cette méthode.

`parse_increment(self)` : Analyse une opération d'incrémentation (++, -). Vérifie que la variable est déclarée et retourne un nœud AST.

`parse_short_operation(self)` : Analyse une opération courte (`x++`, `x-`). Vérifie la déclaration de la variable et retourne un nœud AST.

`parse_assignment(self)` : Analyse une affectation de valeur à une variable. Vérifie la déclaration de la variable et le type de la valeur assignée. Retourne un nœud AST.

`parse_expression_condition(self)` : Analyse une expression conditionnelle (ex : `x > y  z < w`). Parse les termes et opérateurs, et retourne l'expression analysée.

`parse_term_condition(self)` : Analyse un terme dans une condition (ex : une variable, un nombre, ou une chaîne). Vérifie son type et retourne le terme analysé.

`parse_expression(self, expected_type=None)` : Analyse une expression (ex : une opération mathématique). Parse les termes et vérifie leur type. Retourne l'expression et son type.

`parse_term(self, expected_type)` : Analyse un terme dans une expression (ex : un nombre ou une variable). Vérifie le type attendu et retourne le terme analysé.

`parse_sdl_function(self)` : Analyse un appel de fonction SDL (ex : `initMatrix`). Parse ses arguments et retourne un nœud AST.

`parse_statement(self)` : Analyse une instruction unique. Identifie son type (déclaration, condition, boucle, etc.), appelle la fonction appropriée, et retourne un nœud AST.

3.4 Generation du code C

Le Compilateur appel la fonction `generate_c_code(ast, lst, current_line)` : Cette fonction génère du code C à partir de l'AST (Abstract Syntax Tree). Elle traite chaque nœud de l'AST en fonction de son type et génère la représentation correspondante en C. Voici une explication des différents types de nœuds gérés :

`var_decl` : Génère une déclaration de variable. Si la variable est initialisée, inclut sa valeur dans la déclaration. *Exemple :* `int x = 5;`.

`pen_decl` : Génère une déclaration pour un objet `pen`, avec initialisation de ses coordonnées via `createPen(x, y)`. *Exemple :* `PEN myPen = createPen(10, 20);`.

`pen_method` : Génère un appel de méthode sur un objet `pen`. Les paramètres de la méthode sont inclus. *Exemple :* `myPen.move(10, 20);`.

`function_call` : Génère un appel de fonction standard en C avec ses arguments. *Exemple :* `myFunction(arg1, arg2);`.

`sdl_function_call` : Génère un appel de fonction SDL (ex : `initMatrix`). Les arguments sont inclus dans l'appel. *Exemple :* `initMatrix();`.

`function` : Génère une définition de fonction avec ses paramètres et son corps. Appelle récursivement `generate_c_code` pour traiter le corps de la fonction. *Exemple :*

```
int myFunction(int x, int y) {
    // Corps de la fonction
```

```
}
```

`return` : Génère une instruction `return` avec la valeur de retour spécifiée. *Exemple :* `return x + y;`.

`assignment` : Génère une instruction d'affectation. *Exemple :* `x = 5;`.

`methode` : Génère une méthode de contrôle, comme `WAIT`. *Exemple :* `WAIT;`.

`short_operation` : Génère des opérations courtes comme l'incrémentation (`x++`) ou la décrémentation (`x-`). *Exemple :* `x++;`.

`repeat` : Génère une boucle `for`, avec initialisation, condition, et incrémentation. Appelle récursivement `generate_c_code` pour traiter le corps de la boucle. *Exemple :*

```
for (int i = 0; i < 10; i++) {
    // Corps de la boucle
}
```

`if` : Génère une structure conditionnelle `if/else if/else`. Appelle récursivement `generate_c_code` pour traiter chaque bloc conditionnel. *Exemple :*

```
if (x > 0) {
    // Corps du if
} else if (x < 0) {
    // Corps du else if
} else {
    // Corps du else
}
```

`pen_attribute` : Génère une affectation d'attribut pour un objet `pen`. Si l'attribut est `color`, il inclut l'appel à une fonction comme `defineColor`. *Exemple :* `myPen.color = defineColor("red");`.

`method_call` : Génère un appel de méthode sur un objet, incluant les paramètres si nécessaire. *Exemple :* `goTo(myPen, 10, 20);`.

—

En cas de type de nœud non reconnu, une erreur est levée. La fonction retourne le code C généré sous forme de chaîne.

3.5 Gestion des fichiers : lecture et écriture

`read_file(filename)` : Lit le contenu d'un fichier spécifié par `filename`. La fonction ouvre le fichier en mode lecture (`"r"`) avec l'encodage UTF-8, lit tout le contenu, et le retourne sous forme de chaîne. Elle utilise le mot-clé `with` pour s'assurer que le fichier est automatiquement fermé après la lecture.

`write_file(filename, content)` : Écrit une chaîne de caractères dans un fichier spécifié par `filename`. La fonction ouvre le fichier en mode écriture (`"w"`) avec l'encodage UTF-8, écrit le contenu donné, et le ferme automatiquement grâce à `with`.

`import sys, ast as at` : `sys` permet d'interagir avec le système, comme lire les arguments de la ligne de commande. `ast` (importé sous l'alias `at`) est utilisé pour analyser ou manipuler des expressions Python, notamment les structures de données passées comme arguments.

3.6 main

`main(input_file, line_numbers=None)` : Fonction principale du programme qui coordonne les étapes de compilation du code source en langage `PyDraw` vers du code C.

`Paramètres :`

- `input_file` : Le chemin du fichier source contenant le code `PyDraw`.
- `line_numbers` : Une liste optionnelle de numéros de ligne, utilisée pour effectuer des actions spécifiques lors de la génération de code.

`Étapes principales :`

1. Lit le contenu du fichier source à l'aide de la fonction `read_file`.
2. Effectue l'analyse lexicale avec `tokenize`, suivie de l'analyse syntaxique via la classe `Parser`.
3. Génère l'AST (Abstract Syntax Tree) en appelant la méthode `parse()` du parser.
4. Ajoute un `header` et un `footer` spécifiques pour générer un fichier `main.c` avec des appels aux bibliothèques `pyDraw`.
5. Appelle `generate_c_code` pour transformer l'AST en code C et l'écrit dans le fichier de sortie défini (`./c/src/main.c`).
6. En cas d'erreur syntaxique (`SyntaxErrorWithLine`), retourne un message contenant le numéro de ligne et l'erreur détectée.
7. Gère les exceptions générales en retournant une description de l'erreur.

`Utilisation en ligne de commande :`

- Si le script est exécuté directement (`if __name__ == "__main__"`), il prend en entrée le chemin du fichier source et, optionnellement, une liste de numéros de ligne.
- Exemple : `python main.py input.py [1, 2, 3]`.

`Exemple d'utilisation dans un autre script :`

- Importation : `import main`.
- Appel : `main.main("test.txt", [1, 2, 3])`.

4 Code c et implémentation de SDL

Dans cette section, nous allons présenter les différentes fonctionnalités utilisées dans le programme du code c, notamment l'initialisation, la gestion des objets graphiques, et les opérations de dessin. Chaque fonction sera détaillée pour expliquer son rôle et son impact sur l'exécution du programme.

4.1 La Manipulation des matrix

Dans cette section, nous abordons les fonctions essentielles permettant de gérer et de manipuler une matrice graphique, élément central pour représenter et modifier les pixels d'une image. Ces fonctions incluent l'initialisation, la réinitialisation et la modification des couleurs des pixels. Elles constituent la base des opérations graphiques réalisées par le programme, en garantissant une gestion efficace de la mémoire et des données associées à chaque pixel. Passons maintenant à l'explication détaillée des principales fonctions utilisées pour manipuler cette matrice.

`initMatrix()` : Cette fonction initialise une matrice bidimensionnelle représentant une grille graphique de dimensions `WIDTH` x `HEIGHT`. Chaque élément de la matrice correspond à un pixel de type `SDL_Color`, alloué dynamiquement en mémoire à l'aide de `malloc`. Les pixels sont initialisés avec des valeurs par défaut (`DEFAULTCOLOR_BG_R`, `DEFAULTCOLOR_BG_G`, `DEFAULTCOLOR_BG_B`, `DEFAULTCOLOR_BG_A`). Si une allocation mémoire échoue, le programme s'arrête immédiatement avec `exit(1)`.

`clearMatrix(hex)` : Cette fonction remplit la matrice entière avec une couleur spécifiée par le code hexadécimal `hex`. La couleur est convertie en structure `SDL_Color` à l'aide de la fonction `defineColor`. Chaque pixel de la matrice est ensuite mis à jour avec cette couleur, et la fonction `renderMatrix()` est appelée pour actualiser l'affichage graphique.

4.2 Creation du Pen

`initPen()` : Initialise un objet `PEN` avec des valeurs par défaut. La position du `PEN` est centrée dans l'écran (`WIDTH/2`, `HEIGHT/2`), l'épaisseur (`thickness`) est initialisée à 1, et la direction (`rotation`) à 0. La couleur du `PEN` est définie par les constantes `DEFAULTCOLOR_PEN_R`, `DEFAULTCOLOR_PEN_G`, `DEFAULTCOLOR_PEN_B`, et `DEFAULTCOLOR_PEN_A`. La fonction retourne un objet `PEN` prêt à être utilisé.

`createPen(x, y)` : Crée un objet `PEN` en utilisant `initPen()` et met à jour ses coordonnées initiales avec `x` et `y`. Retourne un objet `PEN` positionné.

`goTo(pen, x, y)` : Déplace un `PEN` directement aux coordonnées (`x, y`) sans dessiner. La position du `PEN` est mise à jour et retournée.

`walk(pen, length)` : Fait avancer un `PEN` d'une distance spécifiée (`length`) dans la direction actuelle (`rotation`).

- Calcule la position finale (`endX, endY`) à l'aide de fonctions trigonométriques (`cosf`, `sinf`) basées sur l'angle de rotation.
- Détermine une boîte englobante pour la ligne tracée et vérifie qu'elle reste dans les limites de l'écran (`WIDTH` x `HEIGHT`).
- Pour chaque pixel dans cette boîte englobante, calcule si le point se trouve sur la ligne ou à proximité (en tenant compte de l'épaisseur du `PEN`) :
  - Si le point est proche du début, de la fin, ou le long de la ligne, il est colorié avec la couleur actuelle du `PEN`.
- La fonction met à jour la position du `PEN` à (`endX, endY`), actualise la matrice graphique avec `renderMatrix()`, et retourne le `PEN`.

4.3 Gestion des couleurs et remplissage de la matrice

`pop(stack)` : Supprime et retourne l'élément au sommet de la pile `stack` en réduisant sa taille.

`isEmpty(stack)` : Vérifie si la pile `stack` est vide (`true` si `size == 0`).

`freeStack(stack)` : Libère la mémoire utilisée par la pile `stack`.

`fill(x, y, color2Change, colorReplace)` : Remplace une couleur `color2Change` par `colorReplace` dans la matrice à partir du point (`x, y`). Utilise une pile pour gérer les positions à traiter et applique un algorithme de remplissage en profondeur (*flood-fill*). Libère la pile après traitement.

`fillColor(x, y, hex)` : Convertit une couleur hexadécimale (`hex`) en `SDL_Color` avec `defineColor`, et appelle `fill` pour appliquer le remplissage. Met à jour l'affichage avec `renderMatrix`.

`push(stack, pos)` : Ajoute une position (`pos`) à la pile `stack`. Si la capacité est atteinte, double la taille de la pile avec `realloc`.

4.4 Manipulations graphiques avancées

`circle(pen, radius)` : Dessine un cercle de rayon `radius` autour de la position du `PEN`, en respectant l'épaisseur et la couleur définies. Met à jour l'affichage.

`rotateArea(x, y, width, height, rotation)` : Effectue une rotation d'une zone rectangulaire autour de son centre en utilisant une matrice temporaire pour la transformation. Actualise la matrice principale et l'affichage.

`copyPaste(x, y, width, height, x1, y1)` : Copie une zone rectangulaire définie et la colle à une nouvelle position.

`copy(x, y, width, height)` : Enregistre une zone rectangulaire dans une matrice temporaire pour une utilisation ultérieure.

`paste(x, y)` : Colle la matrice temporaire à une position donnée dans la matrice principale.

`cut(x, y, width, height, x1, y1)` : Coupe une zone et la déplace à une nouvelle position, en remplaçant l'ancienne par une couleur blanche.

`translation(x, y, width, height, length, rotation, precision)` : Déplace une zone rectangulaire selon un vecteur donné (`length`, `rotation`) avec un contrôle sur la précision des incréments.

4.5 Gestion SDL et rendu graphique

`waitKey()` : Attend qu'une touche soit pressée ou qu'un événement `SDL_QUIT` soit déclenché. En cas de fermeture, libère les ressources (fenêtre, renderer, et matrice).

`closeEventSDL()` : Attend un événement `SDL_QUIT` pour fermer l'application. Libère les ressources graphiques et la mémoire allouée.

`renderMatrix()` : Rend chaque pixel de la matrice graphique sur l'écran en utilisant le renderer SDL. Met à jour l'affichage avec `SDL_RenderPresent`.

`initSDL()` : Initialise SDL, crée une fenêtre et un renderer. Si une étape échoue, affiche une erreur et quitte. Appelle `renderMatrix()` pour le premier rendu.

4.6 Fonctions diverses et utilitaires

`hex2rgb(hex)` : Convertit une couleur en format hexadécimal (`hex`) en une structure `SDL_Color`. Gère les formats RGB (`RRGGBB`) et RGBA (`RRGGBBAA`). Retourne la couleur ou affiche une erreur en cas de format invalide.

`compareSDLColors(color1, color2)` : Compare deux couleurs `SDL_Color` en vérifiant l'égalité de leurs composantes (`r`, `g`, `b`, `a`). Retourne 1 si elles sont identiques, sinon 0.

`defineColor(hexColor)` : Valide et convertit une couleur hexadécimale en une structure `SDL_Color`. Retourne une couleur valide ou une couleur noire par défaut en cas d'erreur.

`approxPosX(x)`, `approxPosY(y)` : Approximent une coordonnée `x` ou `y` au pixel le plus proche en arrondissant à l'entier supérieur si la partie décimale est supérieure à 0,5.

`approxPos(x, y)` : Calcule la position approximative (`x, y`) en appelant `approxPosX` et `approxPosY`. Retourne une structure `POS`.

`float2Rad(degrees)` : Convertit un angle en degrés en radians pour les calculs trigonométriques.

`pixelColor(x, y)` : Retourne la couleur (`SDL_Color`) du pixel à la position (`x, y`) dans la matrice.

4.7 main

Dans ce programme principal, nous démontrons les différentes fonctionnalités offertes par la bibliothèque `pyDraw`. Voici une vue d'ensemble des étapes et actions réalisées :

- Initialisation de la matrice et de l'environnement graphique (`initMatrix`, `initSDL`).
- Création de deux stylos (`PEN`), positionnés à des coordonnées différentes. Ces stylos sont utilisés pour dessiner des lignes et des formes géométriques à l'écran.
- Dessin de lignes et déplacement des stylos (`walk`, `goTo`), avec la possibilité de modifier leur orientation (`rotation`).
- Utilisation de la fonction `fillColor` pour remplir des zones spécifiques avec des couleurs définies en format hexadécimal.
- Définition d'une fonction `test` qui dessine un cercle avec un rayon donné autour de la position actuelle d'un `PEN`.
- Utilisation d'une boucle `for` pour réaliser des dessins répétitifs, avec des modifications conditionnelles de paramètres comme la couleur, l'épaisseur, ou l'état du stylo (activé/désactivé).
- Application d'opérations graphiques avancées :
  - Copier-coller des zones de l'écran (`copy`, `paste`, `copyPaste`).
  - Rotation d'une zone définie (`rotateArea`).
  - Translation incrémentale d'une zone selon un angle et une distance (`translation`).

- Effacement de la matrice avec des couleurs spécifiques (`clearMatrix`), et utilisation de la fonction `waitKey` pour attendre une interaction utilisateur.
- Fermeture propre de l'environnement graphique avec `closeEventSDL`.

Ce programme illustre les possibilités créatives et interactives offertes par `pyDraw`, en mêlant dessins géométriques, manipulations graphiques et gestion des interactions utilisateur.