00_pyqt5_interfaces

May 30, 2017

Interfaces graphiques avec QT5

1 Première interface : code minimal

Dans cet exemple, on dérive une classe de QWidget. Cette classe correspondra à la fenêtre principale. On redéfinira toujours au minimum la fonction membre **init**(), qui contient : - la déclaration des éléments graphiques de la fenêtre (boutons, zones de texte, ...) - la fonction show(), qui permet l'affichage de la fenêtre

```
In [ ]: from PyQt5.QtWidgets import QWidget

        class fenetre(QWidget):
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('First UI')
                self.show()
```

Lorsque l'on execute le code ci-dessus, il ne se passe rien. Il faut en effet créer un objet de la classe "fenetre". Il est de plus nécessaire de lancer la boucle infinie ("feedback loop") qui permet de récupérer les interaction de l'utilisateur avec la les différents éléments de la fenêtre. Cela se fait avec quelques lignes de code :

```
In [ ]: import sys
        from PyQt5.QtWidgets import QApplication

        if __name__=='__main__':
            app = QApplication.instance()
            if app is None:
                # On crée une nouvelle application uniquement si aucune n'est déjà
                # Cette vérification n'est nécessaire que dans le cas ou l'on veut
                # une interface graphique sous jupyter notebook
                app = QApplication(sys.argv)

            f1 = fenetre()
            app.exec_()     # démarrage de la "feedback loop"
            # sys.exit(app.exec_())  => à utiliser dans une application lancée depu
```

1.1 Fenêtre principale

Dans l'exemple précédent, la fenêtre principale est dérivée de la classe QWidget. Cette classe ne dispose pas de barre de menu ni de barre d'état. On utilisera ce type de fenêtre principale pour de petites applications ou comme sous-fenêtre. Dans la suite, nous utiliserons essentiellement ce type de fenêtre principale pour conserver un code aussi simple que possible.

Si l'on souhaite développer une application plus grosse, utilisant des menus, il est nécessaire de faire dériver la fenêtre principale de la classe QMainWindow. Nous n'entrerons pas dans le détail de toutes les possibilités offertes par cette classe. Les éléments qui peuvent être insérés dans cette fenêtre sont les suivants : - barre de menus - barre d'outils - fenêtre accrochables (dock widgets) - fenêtre centrale - barre de status

Voici un exemple. Le détail du code n'a pas d'importance à ce stade, l'exemple est simplement là pour vous présenter les possibilités de la classe QMainWindow !

```
In [ ]: import sys
        from PyQt5.QtWidgets import QWidget, QMainWindow, QApplication, QPushButtor
        from PyQt5.QtGui import QIcon
        from PyQt5.QtCore import Qt

        class fenetreComplete(QMainWindow):
            """fenetre principale"""
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('Using a main window')

                # actions
                sortie = QAction(QIcon('exit.png'), '&Exit', self)
                sortie.setShortcut('Ctrl+Q')
                sortie.setStatusTip('Exit application')
                sortie.triggered.connect(self.close)
                whoami = QAction(QIcon(), '&About', self)
                whoami.triggered.connect(self.about_box)

                # barre de menus
                menubar = self.menuBar()
                fileMenu = menubar.addMenu('&File')
                fileMenu.addAction(sortie)
                fileMenu.addAction(whoami)

                #barre d'outils
                toolbar = self.addToolBar('t1')
                toolbar.addAction(sortie)

                # widget centrale
                cen = centre()
                self.setCentralWidget(cen)
```

```
            # dock widget
            doc = dock("d1")
            self.addDockWidget(Qt.LeftDockWidgetArea,doc)

            # barre de status
            self.statusBar().showMessage('Hello World !')

            self.show()

        def about_box(self):
            QMessageBox.about(self, "MainWindow with PyQt5", "version 0.0.0 alp

    class centre(QWidget):
        """fenetre centrale"""
        def __init__(self):
            super().__init__()

    class dock(QDockWidget):
        """dock window"""
        def __init__(self, titre):
            super().__init__(titre)
            lab = QLabel(self)
            lab.setText("dock me !")
            lab.setFixedHeight(100)


    if __name__=='__main__':
        app = QApplication.instance()
        if app is None:
            app = QApplication(sys.argv)
        f1 = fenetreComplete()
        app.exec_()
```

2 Exemples d'éléments d'interface

Pour l'insant, nous avons créé une fenêtre vide, ce qui n'a pas beaucoup d'intérêt ! Pour la remplir, on instancie des objets d'interface (boutons, zones de texte, zone d'image, ...) qui dérivent tous de la classe QWidget et sont accessibles dans le module PyQt5.QtWidgets. ## Bouton Pour créer un bouton, on instancie un objet de la classe QPushButton. Deux arguments sont intéressants à initialiser : - le texte qui apparaitra sur le bouton (par défaut : '' => rien) - le QWidget parent du bouton (par défaut : None) Pour spécifier l'action à effectuer lorsque l'on clique sur le bouton, QT utilise le principe du signal/slot. Il faut alors connecter le signal 'clicked' émis par le bouton lorsque l'utilisateur clique dessus à une fonction qui définit l'action à effectuer.

```
In [ ]: import sys
        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox
```

```
class fenetre_bouton(QWidget):
    def __init__(self):
        super().__init__()
        self.resize(500, 500)
        self.setWindowTitle('First UI')

        b1 = QPushButton("Push me !", self)
        b1.clicked.connect(self.b1_clicked)    #Attention indiquer le nom d
        b1.move(100,200)

        self.show()

    def b1_clicked(self):
        QMessageBox.about(self, "Button pushed", "keep calm and close this


if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
    f1 = fenetre_bouton()
    app.exec_()
```

2.1 Messagebox

On a utilisé un objet QMessageBox pour ouvrir une fenêtre qui délivre un message d'information à l'utilisateur. Différents types de fenêtre peuvent être utilisés : - about - information - warning - critical - question => dans ce cas, l'objet retourne une valeur en fonction du bouton sur lequel l'utilisateur a cliqué : QMessageBox.Yes, QMessageBox.No. D'autres valeurs de retour telles que QMessageBox.Cancel, etc... sont définies. On peut en effet créer des QMessageBox "à la carte", mais cela nécessite quelques lignes de code !

2.2 Exercice 1

Testez le rendu des différents types de QMessageBox.

Déplacez le bouton sur la fenêtre en utilisant la méthode move(dx, dy).

Modifiez le code pour demander à l'utilisateur s'il souhaite désactiver le bouton servant à appeller le QMessageBox. S'il répond oui, désactivez le bouton en utilisant la méthode setEnabled de l'objet QPushButton.

Note : Les méthodes "move" et "setEnable" sont des méthodes héritée de QWidget, donc utilisables sur tout type d'élément d'interface. C'est le cas pour un certain nombre de méthodes, ce qui simplifie l'apprentissage et la lecture du code.

```
In [ ]: # Exercice 1
```

2.3 QLabel

Le QLabel (QWidgets.QLabel) est un élément d'interface relativement polyvalent, qui peut accueillir du texte ou une image - Pour l'affichage de texte, on utilise la méthode text() pour

récupérer le texte et setText() pour écrire - Pour l'affichage d'images, on passe par un objet QPicture (optimisé pour les accès disques et la modification des pixels) ou QPixmap (optimisé pour l'affichage sur écran) que l'on associe au QLabel avec la méthode setPicture() ou setPixmap(). On n'entrera pas dans les détails ici, référez vous à la documentation pour plus d'informations...

2.4 QLineEdit

QWidgets.QLineEdit est un élément incontournable pour récupérer de petites entrées de texte de l'utilisateur (sur une seule ligne). Pour la version multi-ligne, utiliser QWidgets.QTextEdit. Les principales méthodes du QLineEdit sont : - setText() et text() pour modifier/accéder au contenu - insert() pour modifier le texte en passant par un processus de validation - setValidator() permet d'associer un objet QtGui.QValidator, qui vérifie que les données entrées par l'utilisateur répondent bien à certains critères. - un signal editingFinished est émis lorsque l'utilisateur appuie sur entrée ou que le QLineEdit perd le focus (voir chapitre suivant sur les signaux !)

Note sur les QValidator : on utilise généralement cette fonctionnalité pour vérifier que l'utilisateur a entré un entier ou un flottant valide, auquel il également est possible de fixer une limite haute et basse. Il faut alors utiliser la classe dérivée QIntValidator ou QDoubleValidator. Voir l'exemple ci-dessous

```
In [ ]: import sys
        from PyQt5.QtGui import QDoubleValidator
        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox

        class fenetre_bouton(QWidget):
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('Line edit Example')

                self.l1 = QLineEdit(self)
                self.l1.setValidator(QDoubleValidator(0,1,10))
                self.l1.move(100,100)

                b1 = QPushButton("Push me !", self)
                b1.clicked.connect(self.b1_clicked)
                b1.move(100,200)


                self.show()

            def b1_clicked(self):
                if self.l1.hasAcceptableInput():
                    valeur = float(self.l1.text())
                    QMessageBox.about(self, "Entered number is valid", "squared val
                else:
                    QMessageBox.critical(self, "Entered number is not valid", "plea
```

```
if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
    f1 = fenetre_bouton()
    app.exec_()
```

2.5 Autres éléments d'interface

Il existe encore d'autres éléments d'interface, auxquels on ne s'intéressera pas ici en détail, mais dont voici une liste non exhaustive : - QComboBox : liste déroulante (+ fonctionnalités de QLineEdit éventuellement) - QCheckBox : liste à cocher. - QSpinBox : liste circulaire, souvent utilisée pour faire défiler des entiers (la liste est couplée à des boutons + et -) - QSlider : Barre de déplacement (pour fixer une valeur numérique)

2.6 Fenêtres d'ouverture / sauvegarde de fichiers

Il est souvent nécessaire de lire les données depuis un fichier ou de sauvegarder les données générées par le programme. Dans ce cas, l'utilisateur doit pouvoir choisir l'emplacement du fichier à lire/écrire. Le moyen le plus simple pour réaliser cela est l'utililsation de boites de dialogue dédiées via la classe QWidgets.QFileDialog.

La clase QFileDialog permet de créer des boites de dialogue sur mesure, mais deux méthodes permettent de créer des boites de dialogue clé en main qui sont les plus rapides à mettre en oeuvre : - getOpenFileName() crée une boite de dialogue pour l'ouverture de fichiers - getSaveFileName() crée une boite de dialogue pour l'enregistrement de données dans un fichier Dans les deux cas, ces fonctions retournent le chemin complet (répertoire + fichier) du fichier sélectionné par l'utilisateur.

Dans les deux cas, la fonction renvoie le chemin du fichier ainsi que le filtre sélectionnés par l'utilisateur, sous forme d'un tuple.

Note : en remplacement des boites de dialogue d'ouverture/sauvegarde de fichiers, on peut utiliser un QTreeView. Cette Widget affiche une arborescence de fichiers dans votre fenêtre principale. Cela peut être utile si l'interface est destinée à permettre l'ouverture fréquente de différents fichiers car cela évite d'avoir le popup de la boite de dialogue à chaque fois. L'inconvénient est qu'il faut prévoir une certaine place dans la fenêtre principale pour que le treeview soit utilisable confortablement.

2.7 Exercice 2 :

Ecrire un programme contenant un bouton "ouvrir fichier" et un QLabel permettant d'afficher le texte présent de le ficher sélectionné par l'utilisateur dans le QLabel.

```
In [ ]: # Exercice 2
        import sys
        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox

        class fenetre(QWidget):
            """fenetre principale"""
            def __init__(self):
                super().__init__()
```

```
            self.resize(500, 500)
            self.setWindowTitle('Text reader')

            # QLabel pour l'affichage du fichier
            #### A COMPLETER ####

            # bouton pour lancer l'ouverture du fichier à lire
            b1 = QPushButton("lecture", self)
            b1.clicked.connect(self.b1_clicked)
            b1.move(100,200)

            self.show()

        def b1_clicked(self):
            # modification d'une option par défaut pour éviter le plantage sous
            options = QFileDialog.Options()
            options |= QFileDialog.DontUseNativeDialog
            path = QFileDialog.getOpenFileName(self, "ouvrir fichier", options=
            #### A COMPLETER ####

if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
        print("OK !")
    f1 = fenetre()
    app.exec_()
```

3 Organiser les éléments : layouts

On a vu qu'il est possible de fixer la position des éléments au sein de la fenêtre en précisant leur position grâce à la méthode move(). Cependant, il faut ajouter du code pour recalculer la position de l'élément si l'utilisateur modifie la taille de la fenêtre. Par ailleurs, lorsque l'interface contient de nombreux élément, le code devient difficile à modifier (par exemple, si l'on veut ajouter un élément au milieu de l'interface, il faut décaler tous les suivants...)

3.1 types de layout

Pour positionner les éléments les uns par rapport aux autres et par rapport à la fenêtre, on utilise donc des layouts. Il en existe 3 types principaux, situés dans le module QtWidgets. - QVBoxLayout => éléments placés verticalement (en colonne) - QHBoxLayout => éléments placés horizontalement (en ligne) - QGridLayout => éléments placés en grille

L'utilisation des layouts est simple : - on déclare le layout. - on insère des éléments dans le layout grâce à la méthode addWidget()

La méthode AddWidget() des QHBoxLayout et QVBoxLayout requièrent uniquement le nom du widget à ajouter. Par contre, pour le QGridLayout, cette méthode attend des paramètres supplémentaires : - un entier indiquant la position verticale (numéro de ligne) dans la grille (1er

élément = 0) - un entier indiquant la position horizontale (numéro de colonne) dans la grille (1er élément = 0) - (optionnel) un entier indiquant le nombre de lignes utilisées pour placer l'élément (si omis => 1) - (optionnel) un entier indiquant le nombre de colonnes utilisées pour placer l'élément (si omis => 1)

Par ailleurs, il est possible d'insérer des layouts dans un layout, grâce à la méthode addLayout(). Cela permet de gérer facilement le placement au sein d'une interface contenant beaucoup d'éléments. Lors de la déclaration du layout, on peut indiquer un parent. Il faut préciser que le parent est la fenêtre principale dans le cas du layout de plus haut niveau, mais rien (ou None) pour les niveaux hiérarchiques inférieurs.

Voyons cela à travers un exemple. Examinez le comportement du positionnement des éléments lorsque l'on modifie la taille de la fenêtre.

Note : il existe également un 4ème type de layout qui peut être intéressant : le QFormLayout. Il permet de créer plus rapidement un ensemble de lignes composées d'un texte et d'un QLineEdit. La méthode addRow() qui prend un str et un objet QLineEdit en arguments crée automatiquement le QLabel qui contient le texte passé en premier paramètre. Pour les applications scientifiques, c'est un objet intéressant pour les interfaces ou l'utilisateur doit entrer un certain nombre de valeurs de paramètres.

```
In [ ]: import sys

        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox

        class fenetre(QWidget):
            """fenetre principale"""
            def __init__(self):
                super().__init__()
                self.setWindowTitle('Vertical layout')

                # bouton 1
                b1 = QPushButton("b1", self)
                b1.clicked.connect(self.b1_clicked)
                # bouton 2
                b2 = QPushButton("b2", self)
                b2.clicked.connect(self.b2_clicked)
                # bouton 3
                b3 = QPushButton("b3", self)
                b3.clicked.connect(self.b3_clicked)

                # layout
                vlay = QVBoxLayout(self)    # vlay = layout de plus haut niveau
                vlay.addWidget(b1)
                vlay.addWidget(b2)
                vlay.addWidget(b3)

                self.show()

            def b1_clicked(self):
                QMessageBox.about(self, "Button 1 pressed","")
```

```
        def b2_clicked(self):
            QMessageBox.about(self, "Button 2 pressed","")

        def b3_clicked(self):
            QMessageBox.about(self, "Button 3 pressed","")

if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
        print("OK !")
    f1 = fenetre()
    app.exec_()
```

3.2 séparateurs extensibles

Pour controller le comportement des éléments lors du redimensionnement de la fenêtre, on peut utiliser des séparateurs extensibles. Ces éléments sont ajoutés au layout grâce à la méthode addStretch() qui prend en argument un entier. Cet entier correspond à la vitesse d'expansion de la zone et n'est utile que lorque vous utilisez plusieurs séparateurs extensibles. Note : Si on utilise 0 comme argument de addStretch(), le séparateur n'est pas extensible

L'ajout d'un séparateur extensible va avoir pour effet de "repousser" les éléments de l'interface, puisqu'il occupe tout l'espace disponible dans la fenêtre qui n'est pas occupé par un élément. addSQtretch() n'est pas disponible pour un QGridLayout

3.3 Exercices :

1) Utilisez les séparateurs extensibles dans la fenêtre de code ci-dessous pour :

- que les boutons de l'exemple précédent restent en haut de la fenêtre.
- que les boutons de l'exemple précédent restent au centre de la fenêtre, le troisième bouton étant "collé" en bas.
- tester l'effet de l'argument de addStretch() lorque plusieurs séparateurs sont utilisés.

2) Utilisez un QGridLayout pour former afficher les interfaces suivantes :

- grille de 2x3 boutons.
- grille de 3x3 avec 1 bouton par case sur les 2 premières colonnes et un seul bouton sur la 3ème colonne
- testez l'utilisation de addStretch() dans le cas du QGridLayout

3) Layouts imbriqués

- Reprendre la seconde question de l'exercice 2, mais en utilisant uniquement des QVBoxLayout et QHBoxLayout imbriqués.
- Utilisez un QGridLayout pour former une grille de 2x2 boutons. Cette grille doit rester au centre de la fenêtre sans modification de la taille des boutons lorsque la fenêtre principale est redimensionnée.

```
In [ ]: # Exercices sur les layouts
```

3.4 Taille et alignement des éléments

Par défaut, les éléments prennent toute la place disponible dans la "case" du layout. Cela n'est pas le cas si l'on fixe la taille de l'élément. Pour fixer la taille d'un élément, on peut utiliser la méthode setFixedHeight(hauteur), ou setFixedWidth(largeur) ou setFixedSize(largeur, hauteur).

Une fois la taille fixée, l'alignement au sein du layout indiquera comment l'élément s'affiche. Par défaut, l'alignement est à droite et centré verticalement. On peut préciser l'aligement des éléments d'un layout lors de sa création en ajoutant l'arguement alignement=xxx. La valeur de l'alignement peut être - Qt.AlignRight (droite) - Qt.AlignLeft (gauche) - Qt.AlignTop (haut) - Qt.AlignBottom (bas) - Qt.AlignHCenter (centré horizontalement) - Qt.AlignVCenter (centré verticalement) - Qt.AlignCenter (centré horizontalemet et verticalement)

Un exemple ci dessous :

```
In [ ]: import sys

        from PyQt5.QtCore import Qt
        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox

        class fenetre(QWidget):
            """fenetre principale"""
            def __init__(self):
                super().__init__()
                self.setWindowTitle('Size and alignment')

                # bouton 1
                b1 = QPushButton("b1", self)
                b1.clicked.connect(self.b1_clicked)
                b1.setFixedSize(200,50)
                # bouton 2
                b2 = QPushButton("b2", self)
                b2.clicked.connect(self.b2_clicked)
                b2.setFixedWidth(400)
                # bouton 3
                b3 = QPushButton("b3", self)
                b3.clicked.connect(self.b3_clicked)
                b3.setFixedHeight(200)
                #bouton 4
                b4 = QPushButton("b4", self)
                b4.clicked.connect(self.b3_clicked)
                b4.setFixedWidth(400)

                # layout
                glay = QGridLayout(self)
                glay.addWidget(b1,0,0,Qt.AlignLeft|Qt.AlignTop)
                glay.addWidget(b2,0,1,Qt.AlignLeft|Qt.AlignBottom)
                glay.addWidget(b3,1,0,Qt.AlignRight)
                glay.addWidget(b4,1,1,Qt.AlignRight)
```

```
            self.show()

        def b1_clicked(self):
            QMessageBox.about(self, "Button 1 pressed","")

        def b2_clicked(self):
            QMessageBox.about(self, "Button 2 pressed","")

        def b3_clicked(self):
            QMessageBox.about(self, "Button 3 pressed","")

    if __name__=='__main__':
        app = QApplication.instance()
        if app is None:
            app = QApplication(sys.argv)
            print("OK !")
        f1 = fenetre()
        app.exec_()
```

4 Les signaux QT

Les signaux correspondent à une techique permettant de faire communiquer deux objets entre eux recommandée en python. Nous allons nous intéresser particulièrement aux signaux fournis par QT pour la communication entre deux éléments d'une interface graphique. Cependant, voyons dans un premier temps l'utilisation de signaux génériques en python ## Les signaux en python Les signaux sont gérés dans le module "signal". La fonction faisant le lien entre le signal et la fonction à exécuter lorsque le signal est reçu est "signal.signal(). Cette fonction prend deux arguments : le signal à traiter et le nom de la fonction qui doit effectuer le traitement. Dans l'exemple ci-dessous, on crée un signal de type signal.SIGALRM qui se déclenche au bout de 5 secondes grâce à l'instruction signal.alarm(5). Ce signal permet de sortir d'une fonction qui prend trop de temps (programme qui peut rentrer dans une boucle infinie, problème de connection sur un serveur distant, ...)

```
In [ ]: import sys
        import signal

        def handler(a,b):
            print('signal reçu : {} -- {}'.format(a,b))
            raise RuntimeError("program took too long !")

        signal.signal(signal.SIGALRM, handler)
        signal.alarm(5)

        try:
            while(1):
                pass
        except RuntimeError:
```

```
        pass

    print('fin du programme normale !')
```

4.1 les signaux avec QT

QT propose sa propre classe de signaux dédiés aux interfaces graphiques. - la déclaration d'un signal se fait en déclarant un objet de type QtCore.pyqtSignal - l'émission des signaux est obtenue en redéfinissant les fonctions correspondant à l'action de l'utilisateur, qui sont toutes définies dans la classe QWidget (dont dérivent tous les objets d'interface) - le lien entre le signal et la fonction a exécuter est fait grâce à la fonction "connect"

Dans la pratique, la déclaration du signal et son émission sont réalisés au sein de l'objet d'interface. Par contre, le lien avec la fonction à exécuter ainsi que cette fonction sont définis au sein de la fenêtre principale. On a déjà vu la connection d'un signal à une fonction dans le cas du bouton. Dans ce cas, on utilise un signal "clicked" défini par QT. C'est la manière la plus courante d'utiliser les signaux.

Cependant, on souhaite dans certain cas modifier le comportement de certains éléments d'interface pour qu'ils puissent envoyer des informations non prévues par défaut. Il faut alors créer soi-même le signal et gérer son émission. Par exemple, la classe "QLabel", qui permet d'afficher des textes ou des images, n'envoie pas de signal lorsque l'on clique dedans. On va ajouter cette fonctionnalité dans une classe "clickLabel" dérivant de QLabel

```
In [ ]: import sys
        from PyQt5.QtCore import pyqtSignal, QEvent
        from PyQt5.QtWidgets import QWidget, QApplication, QLabel, QMessageBox

        class fenetre(QWidget):
            """ la fenetre principale """
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('signals in PyQt')

                l1 = clickLabel("Click me !", self)
                l1.signal1.connect(self.l1_clicked)

                self.show()

            def l1_clicked(self, ev):
                QMessageBox.about(self, "label clicked", "at position : {}".format

        class clickLabel(QLabel):
            """QLabel dans lequel le clic souris est pris en charge"""
            signal1 = pyqtSignal(QEvent)      # variable globale de classe

            def mousePressEvent(self, ev):    # redéfinition de la fonction de QWidg
                self.signal1.emit(ev)
```

```
if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
    f1 = fenetre()
    app.exec_()
```

Dans certains cas, on peut vouloir connecter le signal émis par différents objets à une même fonction destinataire. Dans ce cas, il peut être intéressant de savoir quel objet a émis le signal. On peut connaître ce dernier grâce à la méthode sender(), fournie par la classe dans lequel la connection est réalisée (généralement, la fenêtre principale). Voici un exemple de l'utilsation de sender() : une interface possédant deux boutons, dont les signaux clicked sont tous les deux liés à la fonction b_clicked(). Cette fonction peut retrouver sur quel bouton l'utilisateur à cliqué grâce à la méthode sender().

```
In [ ]: import sys
        from PyQt5.QtCore import pyqtSignal, QEvent
        from PyQt5.QtWidgets import QWidget, QApplication, QPushButton, QMessageBox

        class fenetre(QWidget):
            """ la fenetre principale """
            def __init__(self):
                super().__init__()
                self.resize(150, 200)
                self.setWindowTitle('signals in PyQt - sender')

                b1 = QPushButton("ONE", self)
                b1.clicked.connect(self.b_clicked)
                b2 = QPushButton("TWO", self)
                b2.clicked.connect(self.b_clicked)
                b2.move(0,100)

                self.show()

            def b_clicked(self):
                if self.sender().text() == "ONE":
                    QMessageBox.about(self, "ONE clicked", "now try with button 2")
                else:
                    QMessageBox.about(self, "TWO clicked", "now try with button 1")

        if __name__=='__main__':
            app = QApplication.instance()
            if app is None:
                app = QApplication(sys.argv)
            f1 = fenetre()
            app.exec_()
```

4.2 Dessin dans un QLabel

Pour mettre en pratique les signaux, on se propose de réaliser une application qui permet de dessiner des rectangles ! Mais avant de passer à la réalisation, il faut quelques notions sur le dessin dans QT.

Le dessin dans une fenêtre peut être réalisé de différentes façons. La plus courante consiste à redéfinir la méthode "paintEvent" de QWidget. On crée alors un objet QPainter, qui représente en quelque sorte le peintre et son pinceau. Cet objet QPainter contient tout ce qu'il faut pour dessiner : - des primitives pour le dessin de formes de base (rectangle, ellipse, ...) - QPen = définit comment sont tracés les traits ainsi que le contour des formes - QBrush = définit comment sont remplies les formes

En outre, d'autres objets interviennent pour la description des dessins à effectuer : - QColor décrit une couleur (par exemple grâce aux paramètres R,G,B) - QPoint décrit les coordonnées d'un point en 2 dimensions (x,y) - QRect décrit les coordonnées d'un rectange (par exemple à l'aide de 2 QPoints : haut-gauche et bas-droite)

Ci-dessous est présenté un exemple de tracé d'un rectangle dans un QLabel. A titre d'exercice, essayez d'effacer le bord noir autour du rectangle !

```
In [ ]: import sys
        from PyQt5.QtCore import Qt, QPoint, QRect
        from PyQt5.QtWidgets import QWidget, QApplication, QLabel
        from PyQt5.QtGui import QPainter, QColor, QBrush, QPixmap

        class fenetre(QWidget):
            """ la fenetre principale """
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('"Draw me a rectangle"')
                l1 = QLabel(parent=self)
                l1.setFixedSize(400,400)
                self.show()

            def paintEvent(self, event):
                antoine = QPainter(self)

                pinceau = QBrush()
                pinceau.setStyle(Qt.SolidPattern)
                pinceau.setColor(QColor(255,0,0))
                antoine.setBrush(pinceau)

                point_hg = QPoint(10,10)
                point_bd = QPoint(100,200)
                antoine.drawRect(QRect(point_hg, point_bd))
                antoine.end()


        if __name__=='__main__':
```

```
app = QApplication.instance()
if app is None:
    app = QApplication(sys.argv)
f1 = fenetre()
app.exec_()
```

4.3 Exercice 3

Dans la fenêtre ci-dessous, écrivez le code d'une application permettant de répondre au cahier des charges suivant : - La fenêtre principale comprend une zone de dessin basée sur un QLabel - Avec le bouton gauche de la souris, l'utilisateur peut tracer un rectangle bleu. Un coin du rectangle est situé à l'endroit ou il a appuyé sur le bouton de la souris, et le coin opposé à l'endroit où il a relaché le bouton.

```
In [ ]: # EXERCICE 3 : tracé de rectangles
```

5 Graphes matplotlib

Matplotlib est une librairie couramment utilisée pour le tracé de graphiques. L'utilisation standard de matplotlib génère une fenêtre séparée qui contient le graphique.

Voici un exemple d'interface qui utilise matplotlib :

```
In [ ]: %matplotlib

        import sys

        import matplotlib.pyplot as plt
        import numpy as np

        from PyQt5.QtWidgets import QWidget, QApplication, QVBoxLayout, QPushButtor

        class fenetre(QWidget):
            """ la fenetre principale """
            def __init__(self):
                super().__init__()
                self.resize(500, 500)
                self.setWindowTitle('sinusoidal plot')
                a=QPushButton("launch plot", self)
                a.clicked.connect(self.launch)

                self.show()

            def launch(self):
                #remplissage du graphe
                x=np.linspace(0,10,1000)
                y=np.sin(x)
                plt.plot(x,y)
                plt.show()
```

```
if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
    f1 = fenetre()
    app.exec_()
```

Si l'on souhaite insérer le graphique généré par matplotlib au sein de l'interface, la procédure est légèrement plus complexe : - on crée un objet Figure, qui correspond à un containeur permettant l'affichage de plusieurs graphs (plot). Lors de la création, on spécifie la taille (en inch) - on crée un objet FigureCanvasQTAgg, qui correspond à l'emplacement sur lequel on va afficher la figure. Cet objet dérive de QWidget et pourra donc être inséré dans un layout grâce à la méthode addWidget() - on crée enfin un (ou plusieurs) graphes dans la Figure grâce à la méthode add_subplot()

Pour utiliser le graphique ainsi créé, on utilise les mêmes fonctions que dans le cas d'un graphe non intégré : - graph.plot(x,y) => ajout d'une courbe - graph.cleat() => efface toutes les courbes

Par contre, une commande supplémentaire est nécessaire pour forcer le graphique à se réafficher après avoir utilisé graph.plot() : - canevas.draw() => force la mise à jour du graphique

Voici un exemple d'application. Vous pouvez tester le comportement lorsque le canevas est inséré directement dans la fenêtre principale ou dans un layout.

In [ ]:
```
import sys

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as canevas
from matplotlib.figure import Figure
import numpy as np

from PyQt5.QtCore import Qt, QPoint, QRect
from PyQt5.QtWidgets import QWidget, QApplication, QVBoxLayout, QPushButton

class fenetre(QWidget):
    """ la fenetre principale """
    def __init__(self):
        super().__init__()
        self.resize(500, 500)
        self.setWindowTitle('sinusoidal plot')

        # definition du canevas
        fig = Figure((4, 4))
        can = canevas(fig)
        can.setParent(self)

        #définition et remplissage du graphe
        sin_graph = fig.add_subplot(111)
        x=np.linspace(0,10,1000)
        y=np.sin(x)
        sin_graph.plot(x,y)
```

```
            # insertion du canevas dans un layout et affichage
            lay = QVBoxLayout(self)
            lay.addWidget(can)

            self.show()

if __name__=='__main__':
    app = QApplication.instance()
    if app is None:
        app = QApplication(sys.argv)
    f1 = fenetre()
    app.exec_()
```

5.1 Exercice 4

1) Créez une interface contenant 3 boutons (sinus, cosinus et reset). Cette interface devra permettre d'afficher une courbe de sinus, de cosinus, ou d'effacer le graphique. Le graphique sera situé dans la même fenêtre que les boutons.

2) Ajoutez la possibilité pour l'utilisateur de régler la valeur minimale et maximale sur l'axe des abscisses.

3) Ajoutez un bouton "open" et un bouton "save" pour ouvrir ou sauvegarder le graphe au format 2 colonnes.

```
In [ ]: # Exercice 4
```

6 Projet plotter

On souhaite réaliser une interface graphique permettant d'afficher des courbes en 2 dimensions et de les fitter. Dans ce projet, on se propose de construire l'interface en ajoutant des fonctionnalités au fur et à mesure. Vous procéderez ainsi par versions, chaque nouvelle version étant démarrée par copie de la version précédente dans une nouvelle cellule.

Pour stocker les données, on utilisera un objet QTableWidget. Cet objet à l'avantage de pouvoir représenter les données graphiquement sous forme d'un tableau (que l'on peut en plus éditer si besoin !). Les principales fonctionnalités du QTableWidget sont résumées ci-après. Pour plus d'informations, référez vous à la documentation sur internet. - Imports : il faut importer QTableWidget et QTableWidgetItem depuis PyQt5.QtWidgets. - Constructeur : on peut spécifier le nombre de colonnes et de lignes, ainsi que le QWidget parent. - Entrée des données : les données sont forcément de type string. Pour insérer une nouvelle cellule dans le tableau, on utilise la méthode setItem(i,j,e). 'i' et 'j' correspondent aux coordonnées (ligne/colonne) de la cellule et 'e' est un objet de type QTableWidgetItem (= cellule du tableau). Cette dernière classe possède un constructeur dans lequel on peut passer en paramètre un float ou un string. une insertion d'un cellule en (3,1) contenant la valeur 'z' ressemblera donc à : table.setItem(3, 1, QTableWidgetItem(z)) - Récupération des données : se fait grâce à item(i,j).text(). la méthode 'item(i,j)' de QTableWidget retourne un QTableWidgetItem et text() retourne le texte de la cellule. Dans notre cas, on stockera des données de type float, il faudra donc transformer le résultat en float.

6.1 Version 1 :

- initialisation de la fenêtre principale avec un layout contenant deux éléments : le graphe à afficher et un QTableWidget permettant l'affichage des points du graphe
- lors de l'initialisation de le fenêtre principale, les données affichées sont une courbe sinusoïdale

Le code devra comprendre deux méthodes pour la classe de fenêtre principale : - fill(x,y), qui prend en paramètres deux listes ou numpy.ndarray. La méthode permet le remplissage de la QTableWidget (1ère colonne => x, 2nde colonne => y) - plot(), qui affiche le contenu de la QTableWidget dans le graphe.

6.2 Version 2 :

- interface permettant de fixer le nombre de points, l'abscisse minimal et maximale, et de mettre à jour la table et le graphe.
- possibilité d'afficher une courbe via un QLineEdit dans lequel l'utilisateur tape une formule mathématique. Utiliser la fonction eval() pour évaluer le string de cette LineEdit comme s'il s'agissait de code python.
- possibilité de sauvegarder la courbe dans un fichier au format 2 colonnes.

Le code devra comprendre deux méthodes supplémentaires pour la classe de la fenêtre principale : - update(), qui récupère les données des lineEdit pour a) calculer le vecteur x d'apres xmin, xmax et le nombre de points, b) calculer le vecteur y d'après la formule entrée par l'utilisateur et c) mettre à jour le QTableWidget et le graphe. - save(), qui sauvegarde les données de la QTableWidget dans un fichier 2 colonnes.

7 Projet fitter :

L'objectif de ce projet est de fournir une interface permettant le 'fit' d'un fichier de données.

On repartira de la version 1 du projet 'plotter'. On aura dans cette version une table à 3 colonnes (x, y_données et y_fit). la colonne y_données sera remplie grâce à l'ouverture d'un fichier. La colonne y_fit correspondra aux données issues de la formule.

7.1 Version 1

Dans un premier temps, on se concentre sur une interface permettant d'afficher les données issues d'un fichier ainsi que celles issues d'une formule. Les abscisses utilisées pour calculer la formule seront les mêmes que celles des données issues du fichier. La courbe correspondant aux données et celle correspondant à la formule seront affichées sur le même graphe (mais avec une couleur de trait différente)

L'interface devra comporter les éléments suivants, en plus de ceux déjà présents dans la version 1 du projet 'plotter' : - un bouton pour charger le fichier de données ainsi qu'un affichage du nom du fichier de données utilisé actuellement - une LineEdit pour que l'utilisateur puisse entrer la formule de fit. - un QShortcut permettant de lancer la méthode 'update_fit' lorsque l'utilisateur appuie sur la touche entrée ('Return').

Vous devrez implémenter les méthodes suivantes : - load_data(), permettant de charger le fichier de données et de l'afficher (dans la table et le graphe) - update_fit(), pour mettre à jour la

table des données et le graphe une fois la formule éditée par l'utilisateur. - séparer la fonction fill() en une fonction fill_data(x,y) et une fonction fill_fit(y). - mise à jour de la fonction plot() pour afficher les deux courbes

7.2 Version 2

On ajoute à présent la fonctionnalité de fit. Pour commencer, on implémentera un fit "à la main", c'est à dire que l'utilisateur modifie les paramètres jusqu'à obtenir une adéquation entre les données et la formule qui lui convient. On implémentera la procédure de minimisation automatique dans la version 3.

L'idée est que l'utilisateur puisse écrire une formule contenant des paramètres libres. La valeur et le nom de ces paramètres pourra être fixé par l'utilisateur. Pour gérer les paramètres libres de la formule, on utilisera une seconde QTableWidget contenant 2 colonnes : - le nom du paramètre - la valeur du paramètre

Au départ, cette table contiendra une seule ligne. L'utilisateur pourra modifier le nombre de lignes grâce à un QSpinBox.

L'interface de la version 2 devra donc incorporer les éléments suivants : - la table des paramètres de fit - le QSpinBox pour ajuster le nombre de ligne. On utilisera le signal valueChanged émis par le QSpinBox pour appeller la méthode param_number_change().

Vous devrez implémenter les méthodes suivantes : - param_number_change(), pour mettre à jour le nombre de lignes de la table des parametres en fonction de la valeur du SpinBox. Utilisez la méthode value() du QSpinBox pour connaître sa valeur courante. - modification de la fonction update_fit, qui doit à présent lire les données de la table des paramètres. Pour cela, on utilisera la fonction exec(). Par exemple, exec('a=1') a le même effet que si vous aviez mis la ligne de code a=1 dans votre programme.

7.3 Version 3

Dans cette version, il sera possible de réaliser un fit sur la courbe de données grâce à la formule fournie par l'utilisateur. Les paramètres libres du fit sont ceux indiqués dans la table des paramètres. On ajoutera une colonne à cette table pour indiquer à l'utilisateur la valeur des paramètres à l'issue du fit.

Pour réaliser le fit, on implémentera la méthode des moindres carrés. Pour cela, on utilisera la fonction minimize() fournie par le module scipy.optimize. Cette fonction requiert 3 arguments : - le nom de la fonction à minimiser. Cette fonction doit prendre en argument un np.ndarray contenant les parmètres libres de la minimisation et doit retourner un float. - le vecteur des paramètres initiaux (de type numpy.ndarray) - le nom de la méthode de minimisation à utiliser. Dans notre cas, on indiquera : "method='nelder-mead' " La méthode minimize retourne un objet. La propriété 'x' de cet objet contient le vecteur des paramètres permettant d'obtenir le minimum de la fonction. Par exemple :

```
In [ ]: resultat = minimize(param_initiaux, fonction_a_minimiser, method='nelder-me
        param_optimises = resultat.x   # de type np.ndarray
```

L'interface sera modifiée de la manière suivante : - ajouter une troisième colonne à la table des paramètres. Les éléments de cette colonne ne doivent pas pouvoir être modifiés (pour cela utiliser : mon_item.setFlags(mon_item.flags() ^ ~Qt.ItemIsEnabled). - ajouter un bouton 'launch fit' pour démarrer la procédure de fit

Vous devrez implémenter les méthodes suivantes : - modification de la méthode param_number_change pour que les cellules ajoutées sur la 3ème colonne ne puissent pas être éditées. - error_square_sum(). Cette fonction sera celle appelée par minimioze(), elle doit donc prendre un numpy.ndarray en paramètre et retourner un float. Dans cette fonction, on calcule le résultat de la formule en utilisant les paramètres fournis grâce à la variable d'entrée. On fait alors la différence au carré entre le vecteur issu de la formule et celui des données (que l'on va lire dans la table de données). Pour faire la somme des carrés utiliser np.sum(np.square(y_data-y_ref)). - fit_data(), pour lancer la procédure de fit des données : créer le vecteur des paramètres initiaux contenus à l'aide du contenu de la table des paramètres puis lancer l'appel à la fonction minimize(). Vous ferez en sorte que les paramètres optimisés issus du fit soient affichés dans la 3ème colonne de la table de fit et que le graphe soit mis à jour avec le résultat du fit.

Vous pouvez à présent passer à la phase de test des capacités de minimisation de votre programme. Pour cela, utilisez le projet 'plotter' pour créer un fichier de données. Pour ajouter du réalisme, vous pouvez utiliser la fonction numpy.random.rand(n), qui crée un ndarray de n nombres aléatoires compris entre 0 et 1.