QComboBox クラス

概要

`QComboBox` は、ユーザーが複数の選択肢から一つを選択できるドロップダウンリストウィジェットです。省スペースで多くの選択肢を提供でき、フォーム、設定画面、フィルタリング機能などで広く使用されます。編集可能・編集不可の両方のモードをサポートします。

基本的な使用方法

```python
from PySide6.QtWidgets import QApplication, QComboBox, QVBoxLayout, QWidget
import sys

app = QApplication(sys.argv)
window = QWidget()
layout = QVBoxLayout()

# 基本的なQComboBoxの作成
combo = QComboBox()
combo.addItems(["選択肢1", "選択肢2", "選択肢3"])
layout.addWidget(combo)

window.setLayout(layout)
window.show()
sys.exit(app.exec())
```

主要なプロパティとメソッド

アイテムの追加・削除

メソッド	説明	例
`addItem(text, userData)`	アイテムを追加	`combo.addItem("新しい項目")`
`addItems(texts)`	複数のアイテムを追加	`combo.addItems(["項目1", "項目2"])`
`insertItem(index, text)`	指定位置にアイテムを挿入	`combo.insertItem(1, "挿入項目")`
`removeItem(index)`	指定インデックスのアイテムを削除	`combo.removeItem(2)`
`clear()`	すべてのアイテムを削除	`combo.clear()`

選択状態の操作

メソッド	説明	例
`setCurrentIndex(index)`	インデックスで選択を設定	`combo.setCurrentIndex(1)`
`setCurrentText(text)`	テキストで選択を設定	`combo.setCurrentText("項目2")`
`currentIndex()`	現在の選択インデックスを取得	`index = combo.currentIndex()`
`currentText()`	現在の選択テキストを取得	`text = combo.currentText()`
`currentData()`	現在の選択データを取得	`data = combo.currentData()`

アイテム情報の取得

メソッド	説明	例
`count()`	アイテム数を取得	`total = combo.count()`
`itemText(index)`	指定インデックスのテキストを取得	`text = combo.itemText(0)`
`itemData(index)`	指定インデックスのデータを取得	`data = combo.itemData(0)`
`findText(text)`	テキストでインデックスを検索	`index = combo.findText("項目名")`

編集機能

メソッド	説明	例
`setEditable(editable)`	編集可能モードを設定	`combo.setEditable(True)`
`isEditable()`	編集可能かどうかを確認	`editable = combo.isEditable()`
`setInsertPolicy(policy)`	挿入ポリシーを設定	`combo.setInsertPolicy(QComboBox.InsertAtTop)`
`insertPolicy()`	現在の挿入ポリシーを取得	`policy = combo.insertPolicy()`

挿入ポリシー

編集可能モードでのアイテム挿入動作を制御します：

ポリシー	説明
`NoInsert`	新しいアイテムを挿入しない
`InsertAtTop`	リストの先頭に挿入
`InsertAtCurrent`	現在の位置に挿入
`InsertAtBottom`	リストの末尾に挿入
`InsertAfterCurrent`	現在の位置の後に挿入
`InsertBeforeCurrent`	現在の位置の前に挿入
`InsertAlphabetically`	アルファベット順に挿入

主要なシグナル

シグナル	説明	使用例
`currentIndexChanged(index)`	選択インデックスが変更された時	`combo.currentIndexChanged.connect(on_index_changed)`
`currentTextChanged(text)`	選択テキストが変更された時	`combo.currentTextChanged.connect(on_text_changed)`
`activated(index)`	ユーザーがアイテムを選択した時	`combo.activated.connect(on_activated)`
`editTextChanged(text)`	編集テキストが変更された時	`combo.editTextChanged.connect(on_edit_changed)`

実用的な使用例

1. 基本的な選択リスト

```
# 国選択
country_combo = QComboBox()
countries = ["日本", "アメリカ", "イギリス", "フランス", "ドイツ"]
country_combo.addItems(countries)

# 選択変更の処理
def on_country_changed(text):
    print(f"選択された国: {text}")

country_combo.currentTextChanged.connect(on_country_changed)
```

2. データ付きアイテム

```
# 言語選択（表示名とコードを分離）
language_combo = QComboBox()
languages = [
    ("日本語", "ja"),
    ("English", "en"),
    ("Français", "fr"),
    ("Deutsch", "de")
]

for display_name, code in languages:
    language_combo.addItem(display_name, code)

# 選択されたコードを取得
def get_selected_language_code():
    return language_combo.currentData()
```

3. 編集可能なコンボボックス

```
# 検索履歴付きの検索ボックス
search_combo = QComboBox()
search_combo.setEditable(True)
search_combo.setInsertPolicy(QComboBox.InsertAtTop)

# 既存の検索履歴
search_history = ["PySide6", "Qt Framework", "Python GUI"]
search_combo.addItems(search_history)

def perform_search():
    query = search_combo.currentText()
    if query and query not in search_history:
        search_combo.insertItem(0, query)
        search_history.insert(0, query)
    print(f"検索: {query}")

search_combo.activated.connect(perform_search)
```

4. 動的なアイテム更新

```
category_combo = QComboBox()
item_combo = QComboBox()

# カテゴリデータ
categories = {
    "果物": ["りんご", "バナナ", "オレンジ"],
    "野菜": ["にんじん", "じゃがいも", "玉ねぎ"],
    "肉類": ["牛肉", "豚肉", "鶏肉"]
}

category_combo.addItems(list(categories.keys()))

def update_items(category):
    item_combo.clear()
    if category in categories:
        item_combo.addItems(categories[category])

category_combo.currentTextChanged.connect(update_items)
# 初期値の設定
update_items(category_combo.currentText())
```

5. カスタムアイテム表示

```
from PySide6.QtGui import QIcon

# アイコン付きコンボボックス
status_combo = QComboBox()

statuses = [
    ("オンライン", "online.png"),
    ("取り込み中", "busy.png"),
    ("離席中", "away.png"),
    ("オフライン", "offline.png")
]

for text, icon_path in statuses:
    status_combo.addItem(QIcon(icon_path), text)
```

フィルタリング機能の実装

コンプリーター付きコンボボックス

```
from PySide6.QtWidgets import QCompleter
from PySide6.QtCore import Qt

# 大量のデータがある場合のフィルタリング
data_combo = QComboBox()
data_combo.setEditable(True)

# 大量のアイテム
large_dataset = [f"アイテム{i:03d}" for i in range(1, 1001)]
data_combo.addItems(large_dataset)

# オートコンプリート機能
completer = QCompleter(large_dataset)
completer.setCaseSensitivity(Qt.CaseInsensitive)
data_combo.setCompleter(completer)
```

スタイリングとカスタマイズ

CSS スタイルシートの例

```
combo.setStyleSheet("""
    QComboBox {
        border: 2px solid #cccccc;
        border-radius: 5px;
        padding: 5px;
        background-color: white;
        selection-background-color: #007acc;
    }
    QComboBox:hover {
        border-color: #007acc;
    }
    QComboBox::drop-down {
        subcontrol-origin: padding;
        subcontrol-position: top right;
        width: 20px;
        border-left: 1px solid #cccccc;
    }
    QComboBox::down-arrow {
        width: 10px;
        height: 10px;
    }
    QComboBox QAbstractItemView {
```

```
        border: 1px solid #cccccc;
        background-color: white;
        selection-background-color: #007acc;
    }
""")
```

ベストプラクティス

1. 適切なデフォルト選択

```
# 良い例：最も一般的な選択肢をデフォルトに
priority_combo = QComboBox()
priority_combo.addItems(["低", "中", "高", "緊急"])
priority_combo.setCurrentText("中")  # 一般的な優先度をデフォルト
```

2. 空の状態の処理

```
# プレースホルダー的な項目を追加
combo.addItem("-- 選択してください --")
combo.setCurrentIndex(0)

# 実際の処理では最初の項目を除外
def get_valid_selection():
    if combo.currentIndex() > 0:
        return combo.currentText()
    return None
```

3. 大量データの効率的な処理

```
# 遅延読み込みのような機能
def populate_combo_on_demand():
    if combo.count() == 0:  # まだデータが読み込まれていない
        # データベースやAPIからデータを取得
        data = fetch_data_from_source()
        combo.addItems(data)

# フォーカス時にデータを読み込み
combo.focusInEvent = lambda event: populate_combo_on_demand()
```

注意事項

1. **パフォーマンス**: 大量のアイテムがある場合は、仮想化やページングを検討
2. **ユーザビリティ**: 選択肢が多い場合は、検索機能やグループ化を提供
3. **アクセシビリティ**: キーボードナビゲーションを考慮
4. **データ整合性**: 動的にアイテムを変更する際は、現在の選択状態を適切に管理

関連するクラス

- **QListWidget**: リスト表示ウィジェット
- **QCompleter**: オートコンプリート機能
- **QAbstractItemModel**: アイテムモデルの基底クラス
- **QStandardItemModel**: 標準的なアイテムモデル

参考リンク

- Qt公式ドキュメント - QComboBox
- PySide6公式ドキュメント