Programowanie w językach skryptowych

PyQT – widżety i interakcja

Opracował:
dr inż. Wojciech Bieniecki
wbieniec@kis.p.lodz.pl
http://wbieniec.kis.p.lodz.pl
Instytut Informatyki Stosowanej

Standardowe widżety

QCheckBox –przycisk typu *CheckBox*

QSlider – suwak

QProgressBar – pasek postępu

QCalendarWidget – obiekt kalendarza

QPixmap – wyświetlanie obrazu w okienku

QLineEdit – *EditBox* – pozwala na wpisanie jednej linii tekstu

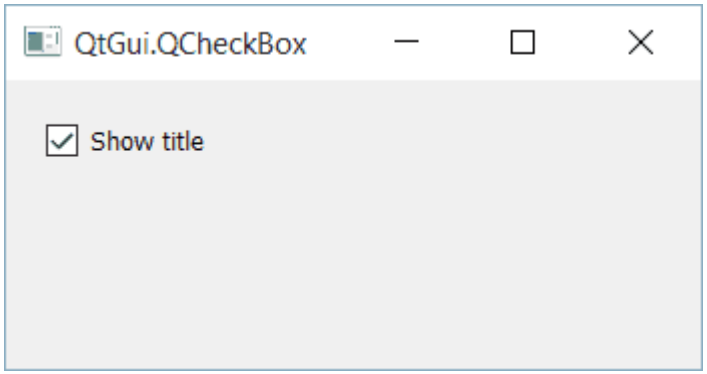
QSplitter – współdziała z Layoutem. Pozwala zmieniać proporcje poszczególnych widżetów.

QComboBox – lista rozwijana (drop down)

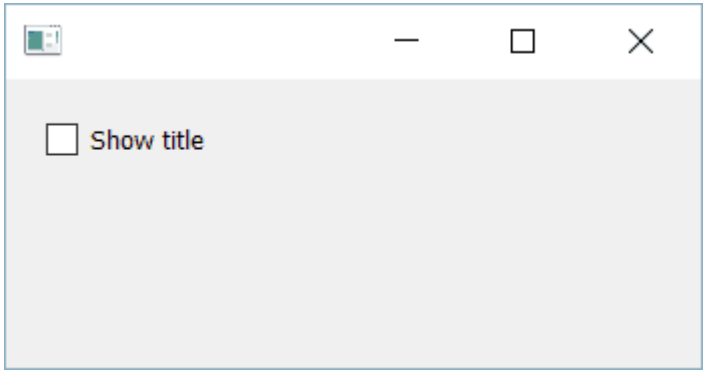
QPainter – tzw. *Canvas* (płótno). Obiekt, na którym można rysować linie, kształty geometryczne, pisać teksty przy użycia pędzla (*Brush*) oraz pióra (*Pen*)

Przykłady użycia widżetów QCheckBox

QtGui.QCheckBox – POLE WYBORU - ma dwa stany: włączony lub wyłączony.
Jest to pole z etykietą.
Używany do reprezentowania opcji, która mogę być włączona lub wyłączona.




```
# utworzenie obiektu i dodanie do okna
cb = QCheckBox('Show title', self)
cb.move(20, 20)
# przełączenie kontrolki
cb.toggle()
# powiązanie sygnału ze slotem
cb.stateChanged.connect(self.changeTitle)
```




```
def changeTitle(self, state):
# odczytanie stanu obiektu
  if state == QtCore.Qt.Checked:
    self.setWindowTitle('QtGui.QCheckBox')
  else:
    self.setWindowTitle('')
```

Przykład użycia QRadioButton

Elementy Radio Button powinny być przełączane.
Powinniśmy je ustawić w grupę i nadać właściwość przełączania.
Każdy przycisk w grupie będzie miał ID

```
options = ['Czerwony', 'Żółty','Zielony']
rb = [QRadioButton(o) for o in options]
bl = QVBoxLayout()
cbg = QButtonGroup(self)
cbg.setExclusive(True)
for id, ch in enumerate(options):
  rb = QRadioButton(ch)
  cbg.addButton(rb)
  cbg.setId(rb, id)
  bl.addWidget(rb)
```

Zdarzenie podłączamy pod grupę. Będzie można uzyskać informację o przycisku i bieżącym ID

```
cbg.buttonClicked.connect(self.nowyKolor)
```

```
def nowyKolor(self, button):
  print(button.text())
  print(self.sender().checkedId())
```

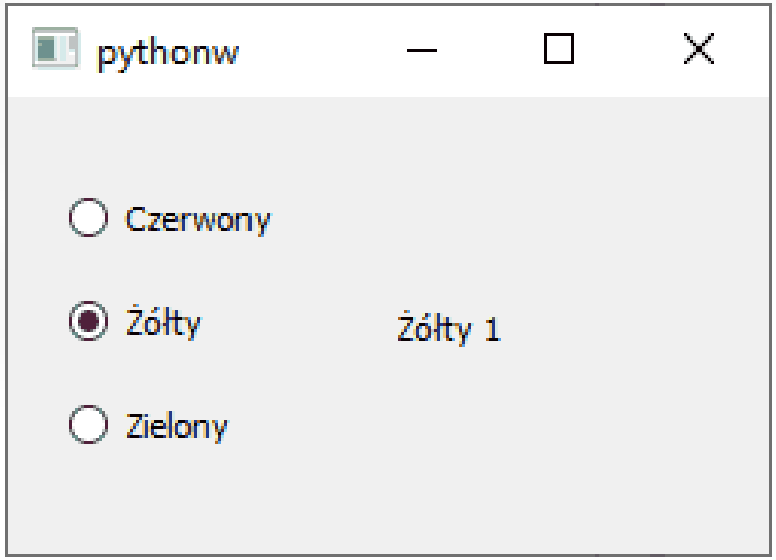
Przykład użycia QRadioButton

Cały kod

```
def createGroup(self):
  options = ['Czerwony', 'Żółty','Zielony']
  rb = [QRadioButton(o) for o in options]
  bl = QVBoxLayout()
  cbg = QButtonGroup(self)
  cbg.setExclusive(True)
  for id, ch in enumerate(options):
    rb = QRadioButton(ch)
    cbg.addButton(rb)
    cbg.setId(rb, id)
    bl.addWidget(rb)
  cbg.buttonClicked.connect(self.nowyKolor)
  w = QWidget()
  w.setLayout(bl)
  return w
```

```
def initUI(self):
  w = self.createGroup()
  hb = QHBoxLayout()
  hb.addWidget(w)
  self.label = QLabel("Kliknij kolor")
  hb.addWidget(self.label)
  self.setLayout(hb)
  self.setGeometry(300, 300, 250, 150)
  self.show()
```

```
def nowyKolor(self, button):
    self.label.setText(button.text() + " " +
str(self.sender().checkedId()))
```



Przykład QSlider + QLineEdit

Wartość możemy wpisywać z pola edycyjnego albo poprzez wybór suwakiem.

Przygotowanie QLineEdit wraz z walidacją (ograniczenie na wpisywanie liczb 0-255)

```
self.le = QLineEdit('0')
objValidator = QIntValidator(self)
objValidator.setRange(0, 255)
self.le.setValidator(objValidator)
self.le.textChanged.connect(self.le_change)
```

Przygotowanie QSlidera

```
self.sl = QSlider(Qt.Horizontal)
self.sl.setMaximum(255)
self.sl.setMinimum(0)
self.sl.valueChanged.connect(self.sl_change)
```

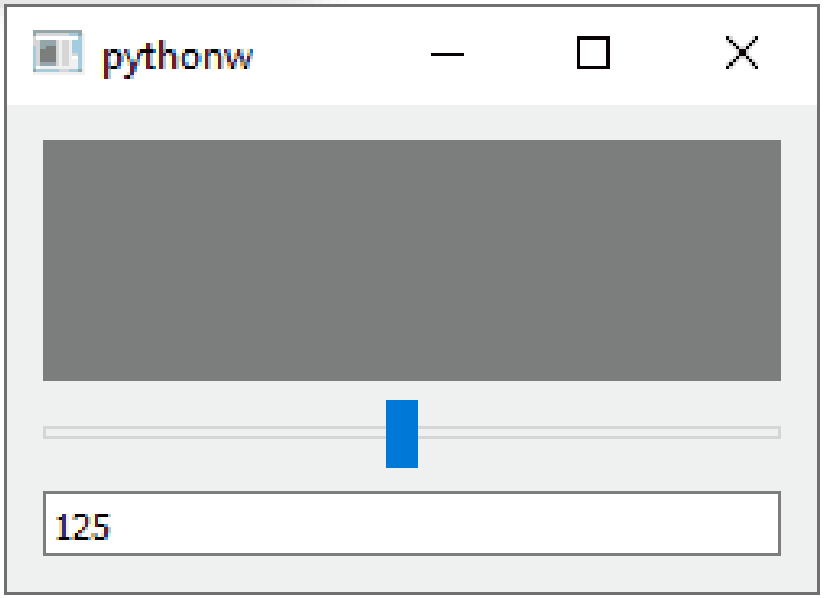
Funkcje obsługi zdarzeń

```
def le_change(self, value):
  if not value:
    return
  self.sl.setValue(int(value))
```

```
def sl_change(self, v):
  self.le.setText(str(v))
  self.prost.setStyleSheet("QWidget {background-color:rgb(%s,%s,%s)}" % (v,v,v))
```

Przykład Qslider c.d. – okno

```
self.prostokat = QLabel()
self.prostokat.setGeometry(QRect(0, 550, 150, 31))
vb = QVBoxLayout()
vb.addWidget(self.prostokat)
vb.addWidget(self.sl)
vb.addWidget(self.le)
self.setLayout(vb)
self.setGeometry(300, 300, 250, 150)
self.show()
```



Zdarzenia

Istnieje wiele rodzajów zdarzeń, z których każde reprezentuje inny typ interakcji - np. zdarzenia myszy lub klawiatury.

Występujące zdarzenia są przekazywane do procedury obsługi zdarzenia w widżecie, w którym nastąpiła interakcja.

Na przykład kliknięcie widżetu spowoduje wysłanie zdarzenia QMouseEvent do procedury obsługi zdarzeń .mousePressEvent w widżecie.

Ten moduł obsługi może przetworzyć zdarzenie, np. aby znaleźć źródło i miejsce zdarzenia.

Zdarzenia

Zdarzenie można przechwycić poprzez utworzenie klasy pochodnej i nadpisanie funkcji obsługi zdarzenia.

Można filtrować, modyfikować lub ignorować zdarzenia, przekazując je do normalnego modułu obsługi zdarzenia, wywołując funkcję klasy nadrzędnej za pomocą super()

```
class CustomButton(QButton):
  def keyPressEvent(self, e):
    super(CustomButton, self).keyPressEvent(e)
```

Jednak w przypadku dużej liczby przycisków problem się komplikuje …

Przechwycenie zdarzenia

```
class Okno(QWidget):

  def initUI(self):
    self.setGeometry(300, 300, 250, 150)
    self.setWindowTitle('Message box')
    self.show()

  def closeEvent(self, event):
    reply = QMessageBox.question(self, 'Message', "Na pewno?", QMessageBox.Yes
| QMessageBox.No, QMessageBox.No)

    if reply == QMessageBox.Yes:
      event.accept()
    else:
      event.ignore()
```

Zamknięcie „krzyżykiem” obiektu QWidget, powoduje wysłanie sygnału QCloseEvent.

Aby przechwycić ten sygnał należy przedefiniować (ang. override) metodę closeEvent

W przykładzie uruchamiamy MessageBox i odbieramy odpowiedź.

Odpowiedź TAK powoduje domyślne przesłanie sygnału a NIE - jego anulowanie

PyQt – sygnały i sloty

Charakterystyczną cechą Qt jest mechanizm sygnałów i slotów będący sposobem porozumiewania się elementów aplikacji.

Sygnał emitowany jest w przypadku wystąpienia określonej akcji (np. przyciśnięto przycisk).

Slot to funkcja połączona (za pomocą *QtCore.QObject.connect()*) z określonym sygnałem i jest wykonywana, gdy taki sygnał zostanie wyemitowany.

Połączenia sygnałów i slotów:

- sygnał może być połączony z wieloma slotam
- sygnał może być połączony z innym sygnałem
- slot może być połączony z wieloma sygnałami
- w PyQt sygnały są emitowane przez metodę *QtCore.QObject.emit()*
- połączenia mogą być bezpośrednie - synchroniczne lub kolejkowane – asynchroniczne
- można tworzyć połączenia między wątkami
- sygnały są rozłączane za pomocą metody *QtCore.QObject.disconnect()*

Metody obsługi sygnałów

Obsłużmy sygnał windowTitleChanged dla klasy QMainWindow

```
class MainWindow(QMainWindow):
  def __init__(self, *args, **kwargs):
    super(MainWindow, self).__init__(*args, **kwargs)
```

Przykład 1: Połączona funkcja będzie wywoływana przy każdej zmianie tytułu okna. Nowy tytuł zostanie przekazany do funkcji jako parametr

```
    self.windowTitleChanged.connect(self.onWindowTitleChange)
```

Przykład 2 Nowy tytuł jest odrzucany w lambda, a funkcja wywoływana jest bez parametrów.

```
    self.windowTitleChanged.connect(lambda x: self.my_custom_fn())
```

Metody obsługi sygnałów

Przykład 3: Połączona funkcja będzie wywoływana przy każdej zmianie tytułu okna. Nowy tytuł x jest przekazywany do funkcji i zastępuje parametr domyślny

```
  self.windowTitleChanged.connect(lambda x: self.my_custom_fn(x))
```

Przykład 4: To samo co powyżej, ale dodatkowe dane mogą zostać przekazywane z wnętrza lambda.

```
self.windowTitleChanged.connect(lambda x: self.my_custom_fn(x, 25))
```

Funkcje realizujące zdarzenie (sloty)

```
def onWindowTitleChange(self, s): #przedefiniowana funkcja
  print(s)# s jest nowym tytulem okna

def my_custom_fn(self, a="HELLLO!", b=5): #wlasna funkcja
  print(a, b) tu można wykorzystać własne parametry
```

Zdarzenia

Dzięki zastosowaniu sygnałów/slotów w większości przypadków można ominąć używanie zdarzeń

Weźmy pod uwagę zdarzenie .contextMenuEvent dla okna QMainWindow. Zdarzenie występuje, gdy wyświetlane jest menu kontekstowe i jest przekazywane zdarzenie o pojedynczej wartości typu QContextMenuEvent.

Aby przechwycić zdarzenie, po prostu nadpisujemy metodę obiektową

```
def contextMenuEvent(self, event):
    print("Context menu event!")
```

Czasem trzeba przechwycić zdarzenie, ale nadal wywołać domyślną procedurę obsługi zdarzeń. To umożliwia propagację zdarzenia w górę hierarchii.

```
def contextMenuEvent(self, event):
    print("Context menu event!")
    super(MainWindow, self).contextMenuEvent(event)
```

Zdarzenia

Propagacja jest zdefiniowana inaczej niż w JavaFX.
Pewne Widżety mają domyślnie zdefiniowaną propagację do swoich rodziców a inne nie.

Kontrolę tego, czy propagacja nastąpi czy nie dokonujemy poprzez .accept() / .ignore()

```
class CustomButton(Qbutton):

    def event(self, e):
        e.accept() #propagacja jest ZABLOKOWANA
```

```
class CustomButton(Qbutton):

    def event(self, e):
        e.ignore() #propagacja jest WYMUSZONA
```

Atrybuty zdarzenia

Każdy rodzaj zdarzenia ma swoje atrybuty, które mogą być przetwarzanie w funkcji obsługi. Np. mysz

```
 def mouseMoveEvent(self, e):
   x = e.x()
   y = e.y()
   text = "x: {0},  y: {1}".format(x, y)
   self.label.setText(text)
```

Czasami wygodnie jest wiedzieć, który widżet jest nadawcą sygnału.
W tym celu stosujemy metodę .sender().

```
  btn1 = QPushButton("Button 1", self)
  btn2 = QPushButton("Button 2", self)
  btn1.clicked.connect(self.buttonClicked)
  btn2.clicked.connect(self.buttonClicked)

def buttonClicked(self):
  sender = self.sender()
  self.statusBar().showMessage(sender.text() + ' pressed')
```

Emisja sygnału

Obiekty utworzone z QObject mogą emitować sygnały. Możemy tworzyć własne sygnały (analogia do obserwatorów i obserwowanych wartości w Javie)

Niech koło będzie obserwowalnym obiektem.

```
import sys
from PyQt5.QtWidgets import *
from PyQt5.QtGui import *
from PyQt5.QtCore import QObject, pyqtSignal, pyqtSlot


class Circle(QObject):
  #tworzenie dwoch sygnalow z parametrami
    resized = pyqtSignal(int)
    moved = pyqtSignal(int, int)

    def __init__(self, x, y, r):
        QObject.__init__(self)
        self._x = x  #prywatne zmienne
        self._y = y  #dostep bedzie
        self._r = r  #poprzez wlasciwosci
```

Emisja sygnału

```
#ciag dalszy definicji Circle
    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, new_x):
        self._x = new_x
        self.moved.emit(new_x, self._y)

    @property
    def y(self):
        return self._y

    @y.setter
    def y(self, new_y):
        self._y = new_y
        self.moved.emit(self._x, new_y)
```

```
@property
    def r(self):
        return self._r

    @r.setter
    def r(self, new_r):
        self._r = new_r
        self.resized.emit(new_r)
```

- Circle dziedziczy po QObject, więc może emitować sygnały.
- Sygnały są tworzone z podpisem slotu, do którego zostaną podłączone.
- Ten sam sygnał może być emitowany w wielu miejscach.

Emisja sygnału

Zdefiniowano sygnały przenoszące dane. Zostaną utworzone sloty przyjmujące te dane.

```
@pyqtSlot(int, int)
def on_moved(x, y):
    print('Circle was moved to (%s, %s).' % (x, y))

# A slot for the "resized" signal, accepting the radius
@pyqtSlot(int)
def on_resized(r):
    print('Circle was resized to radius %s.' % r)
```

W dowolnym miejscu tworzymy obiekt koła i przypinamy sygnały do slotów

```
c = Circle(5, 5, 4)
c.moved.connect(on_moved)
c.resized.connect(on_resized)
c.x += 1
c.r += 1
```

KONIEC