Kurs rozszerzony języka Python

Środowisko GTK+, dekoracja kodu, dynamiczna kompilacja

Marcin Młotkowski

17 listopada 2017

Plan wykładu

Plan wykładu

Biblioteki okienkowe w Pythonie

- curses: interfejs tekstowy
- Tkinter (Tk interface): biblioteka okienkowa Tk + Tix (Tk extension)
- Pygtk, pygnome: API do środowiska Gtk/Gnome
- PyQT: API do QT
- wxWindows
- OpenGL
- PyWin32

GTK+/GNU

GTK+

The **G**IMP **T**ool**k**it

GNOME

GNU Network Object Model Environment

Moduły towarzyszące

Moduły współistniejące z GTK+

- GObject
- ATK
- Pango
- Cairo
- Glade

Środowisko GTK+

Elementy składowe

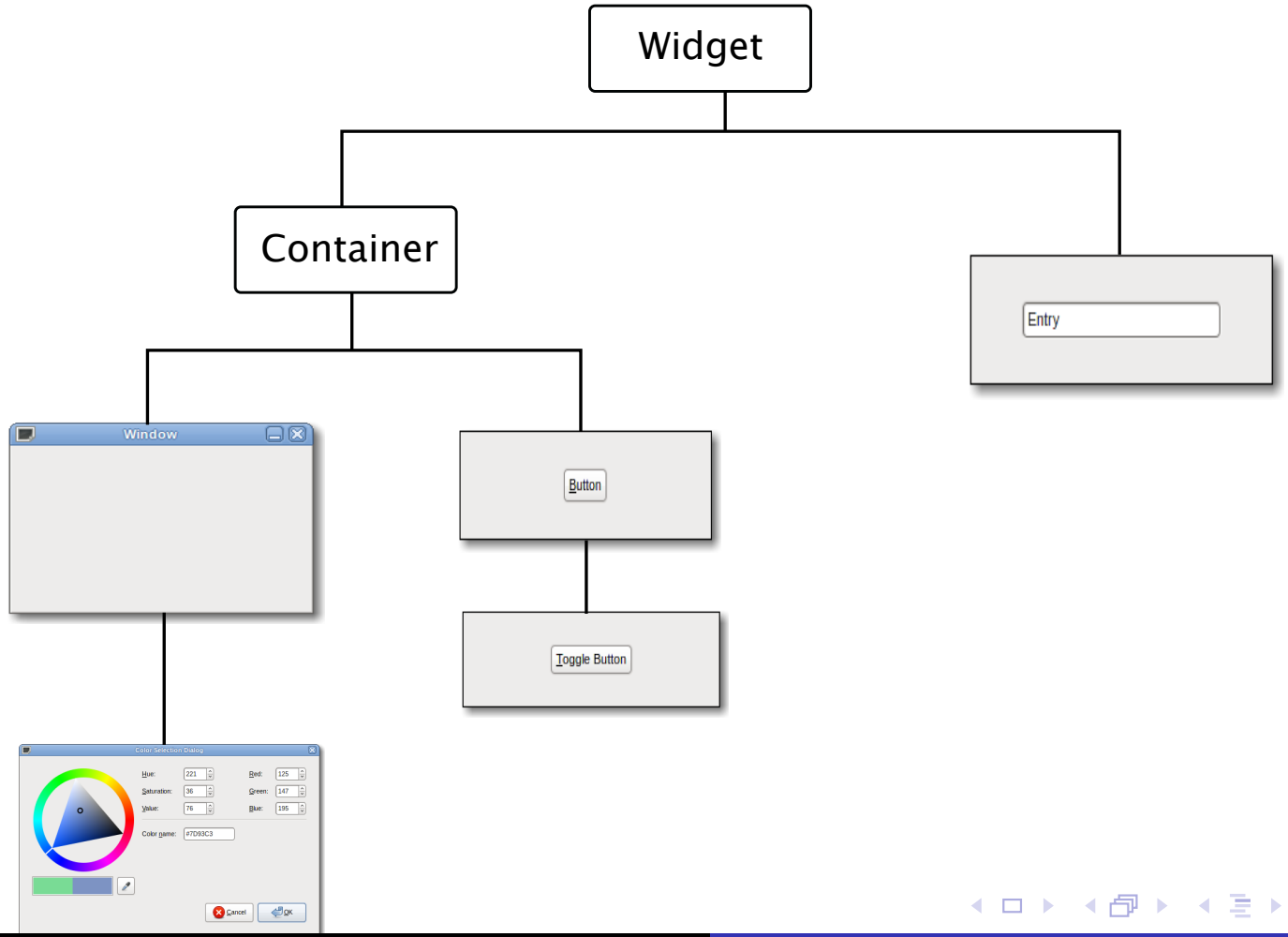
- okna;
- kontrolki;
- zdarzenia

Okna i kontrolki

Rola kontrolki (bardzo nieformalnie):

- kontrolka ma być (najlepiej ładna);
- kontrolka czasem ma reagować, np. na kliknięcie myszą;
- kontrolka czasem może zawierać inne kontrolki.

Hierarchia kontrolek (uproszczone)



GTK+ i Python

Biblioteka `PyGTK`

Współczesność (przeszłość?)

PyGTK obecnie łączy Pythona 2.* i GTK+ 2.0, zapowiadany jest koniec tej biblioteki.

GTK+ i Python

Biblioteka `PyGTK`

Współczesność (przeszłość?)

PyGTK obecnie łączy Pythona 2.* i GTK+ 2.0, zapowiadany jest koniec tej biblioteki.

Współczesność i przyszłość

Biblioteka `PyGObject`: łączy GTK+ 3.0 i Pythona 2.6/3(3.1)

PyGTK: podstawowe elementy

Biblioteka się nazywa `gtk`.

PyGTK: podstawowe elementy

Biblioteka się nazywa `gtk`.

W bibliotece są odpowiednie klasy `Window`, `Entry`, `Button` etc.

Plan wykładu

Przykładowa aplikacja

Specyfikacja

Aplikacja powinna:

- rysować zadane figury;
- podpisywać rysunki;
- kończyć pracę po kliknięciu klawisza na przycisk koniec.

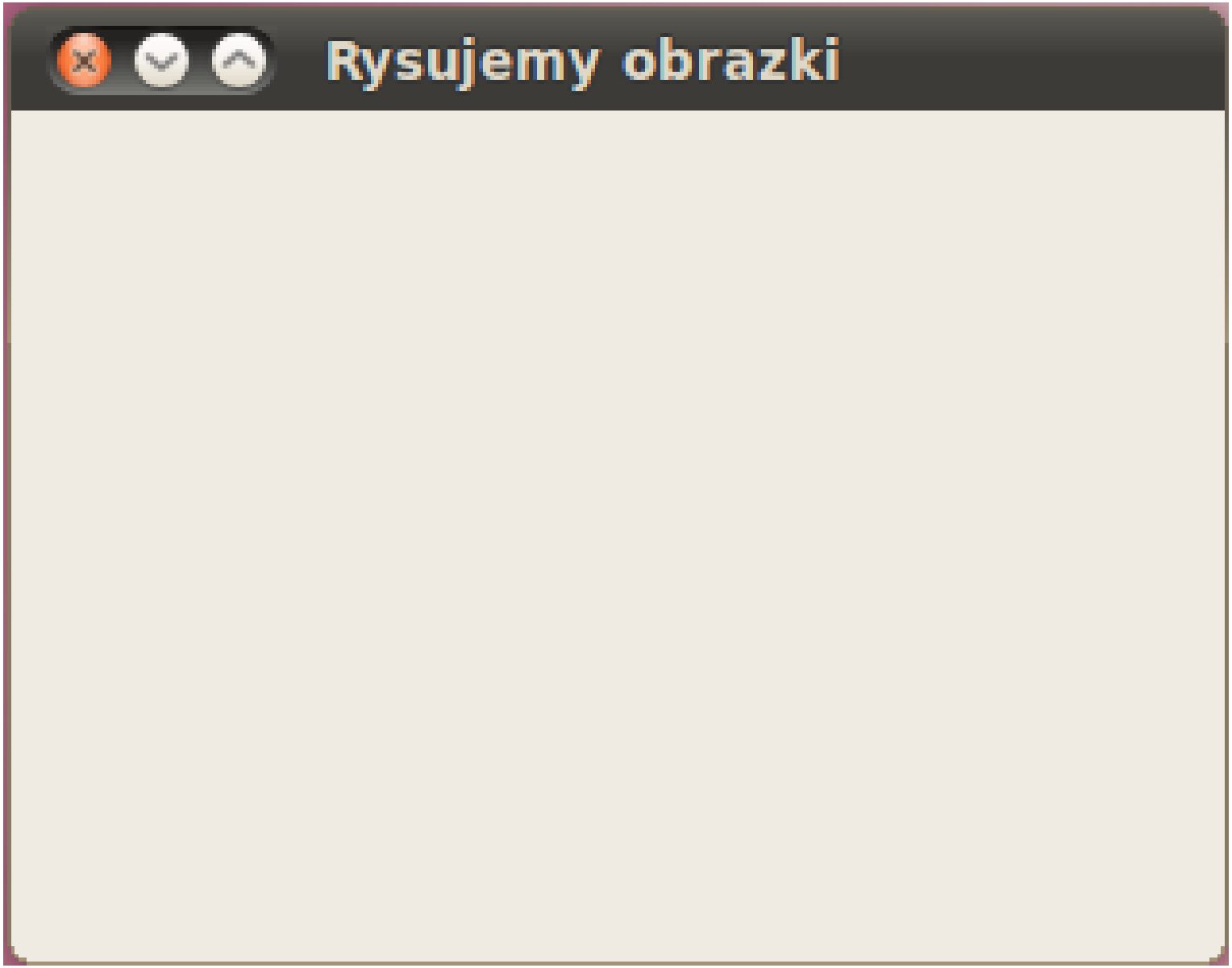
Budowanie okna

```
class Rysownik(gtk.Window):
    def __init__(self):
        super(Rysownik, self).__init__()
        self.set_title("Rysujemy obrazki")
        self.set_position(gtk.WIN_POS_CENTER)
        self.connect("destroy", gtk.main_quit)
        self.kontrolki()
        self.show_all()
```

Uruchomienie aplikacji

```
r = Rysownik()
gtk.main()
```

Efekt



Dokładanie kontrolek

- `Window` jest kontrolką "widzialną";
- `Window` jest też kontenerem, można wstawić element;
- do `Window` można wstawić tylko jeden element.

Pudełka

Do układania elementów służą pudełka pionowe i poziome.

Pudełka

Do układania elementów służą pudełka pionowe i poziome.

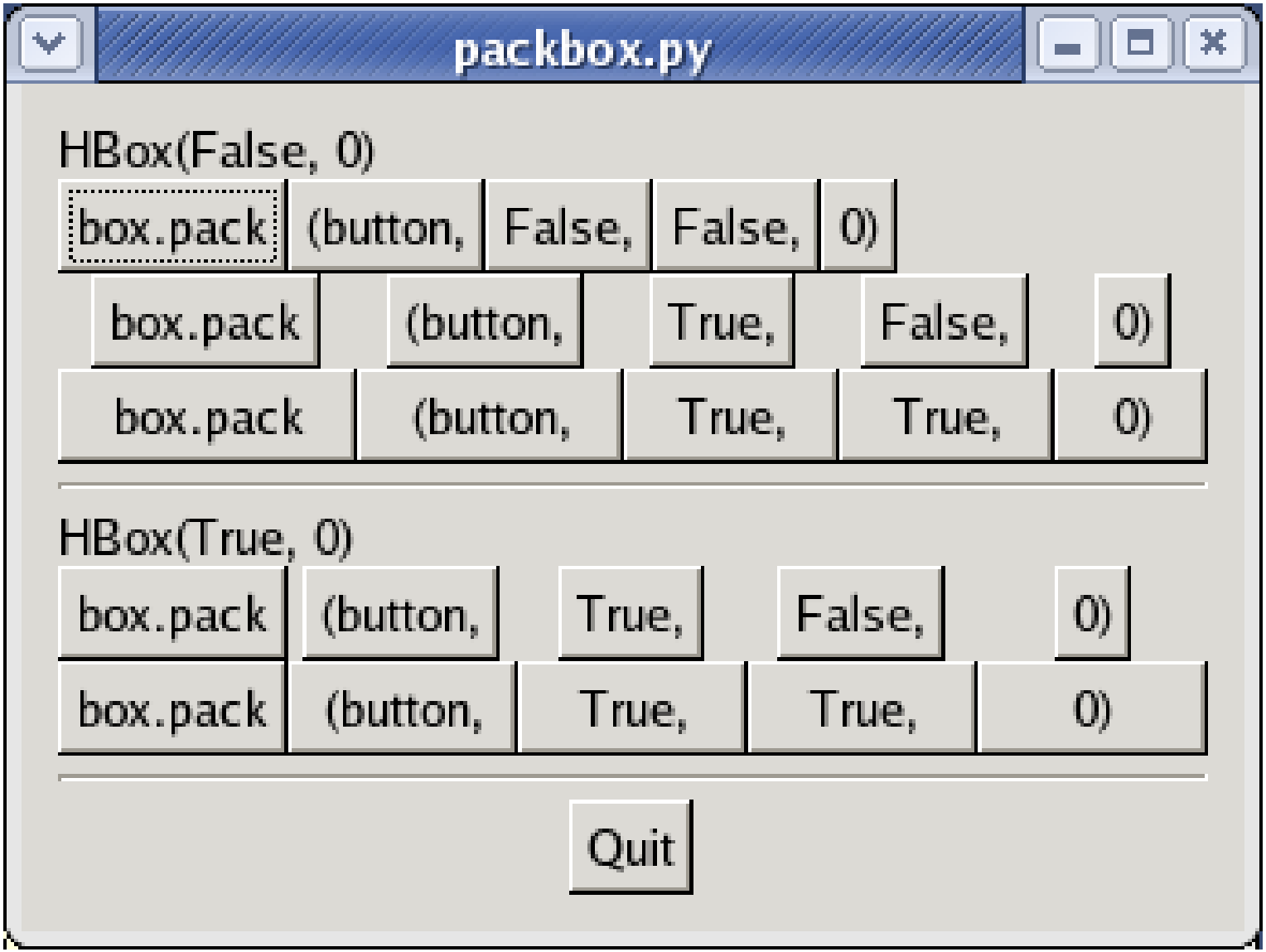
gtk.Box: gtk.VBox i gtk.HBox

.pack_start(*Widget*, *expand*, *fill*, *padding*)
.pack_end(*Widget*, *expand*, *fill*, *padding*)

gdzie

- expand(bool): kontrolki włożone do pudełka mają wypełniają całe pudełko, wypełnieniem jest pusta przestrzeń;
- fill (bool): kontrolki wypełniają całą przestrzeń, ale przy okazji powiększane są kontrolki;
- padding (int): dodatkowy odstęp od sąsiada

Ilustracja pakowania



Parę uwag dodatkowych

Pudełka można pakować w pudełka (pionowe w poziome, poziome w pionowe etc)

Parę uwag dodatkowych

Pudełka można pakować w pudełka (pionowe w poziome, poziome w pionowe etc)

Alternatywa: gtk.Table

"Kratka" komórek, do których wkłada się kontrolki.

Wprowadzanie tekstu

gtk.Entry

- .get_text(): pobranie tekstu z bufora kontrolki;
- .set_text("text"): wyczyszczenie bufora i wstawienie tekstu;
- .insert_text("insert", pos): wstawienie tekstu od *pos*.

Przyciski

gtk.Button('tekst')

```
endb = gtk.Button("Koniec")
endb.connect("clicked", lambda x: self.destroy())
```

Sygnały (ang. *signals*)

Sygnał w GTK+

Informacja, że zaszło jakieś zdarzenie, np. kliknięcie przycisku, likwidacja jakiejś kontrolki.

Sygnały (ang. *signals*)

Sygnał w GTK+

Informacja, że zaszło jakieś zdarzenie, np. kliknięcie przycisku, likwidacja jakiejś kontrolki.

- Sygnały są związane z kontrolkami.
- Sygnały mają swoje nazwy.

Funkcje zwrotne (ang. *callbacks*)

Funkcje zwrotne to są funkcje wywoływane jako reakcja na sygnały.

Funkcje zwrotne (ang. *callbacks*)

Funkcje zwrotne to są funkcje wywoływane jako reakcja na sygnały.

Postać funkcji zwrotnej

def funkcja_zwrotna(*kontrolka*, *dane*)

Funkcje zwrotne (ang. *callbacks*)

Funkcje zwrotne to są funkcje wywoływane jako reakcja na sygnały.

Postać funkcji zwrotnej

def funkcja_zwrotna(*kontrolka*, *dane*)

Łączenie kontrolek, sygnałów i funkcji zwrotnych

kontrolka.connect("nazwa sygnału", *funkcja zwrotna*, *dane*)

Są jeszcze zdarzenia: **Events**.

Są jeszcze zdarzenia: **Events**.

Ale nie będę o nich mówił.

Co to jest

Cairo

Biblioteka 2D, z której można korzystać w GTK+ i Pythonie.

W bardzo wielkim skrócie

Miejsce do malowania

Kontrolka `gtk.DrawingArea()`.

Rysowanie na ekranie

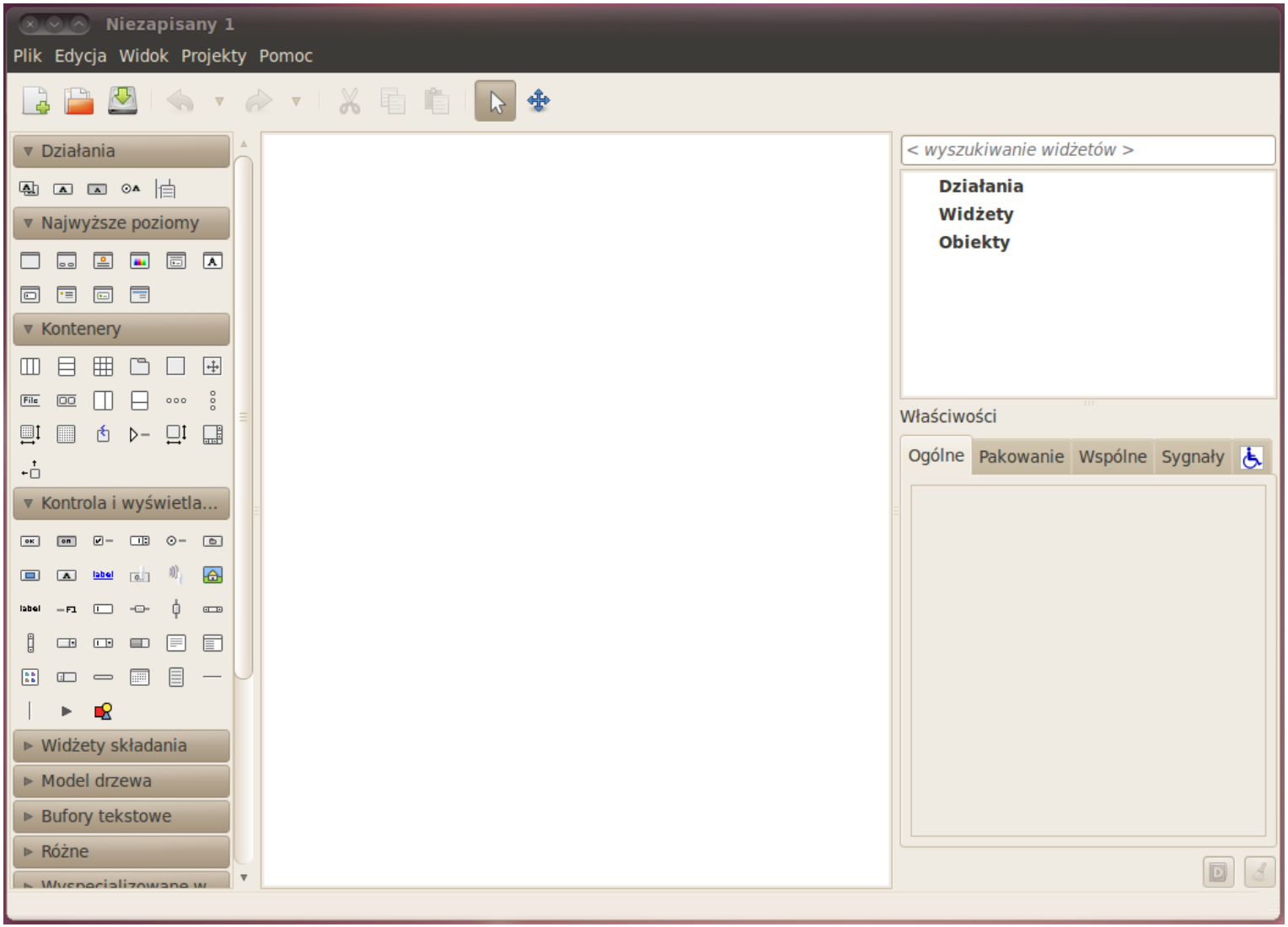
- najpierw tworzy się ścieżkę za pomocą tzw. CairoContext: cairo_context = drawing_area.window.cairo_create()
- następnie rysuje się po ścieżce: cairo_context.stroke()

Co to jest

Glade to graficzne narzędzie do projektowania interfejsów dla środowiska GTK+/GNOME.

Schemat działania (Glade-3)

- Glade produkuje plik XML, w którym jest opisany interfejs;
- Aplikacja "wczytuje" ten plik i buduje interfejs;
- Glade-3 jest niezależny od języka.


Niezapisany 1
Plik Edycja Widok Projekty Pomoc
Działania
Najwyższe poziomy
Kontenery
Kontrola i wyświetla...
Widżety składania
Model drzewa
Bufory tekstowe
Różne
< wyszukiwanie widżetów >
Działania
Widżety
Obiekty
Właściwości
Ogólne
Pakowanie
Wspólne
Sygnały

Użycie projektu

GtkBuilder

Biblioteka budująca z XML interfejs graficzny.

Użycie projektu

GtkBuilder

Biblioteka budująca z XML interfejs graficzny.

libglade

Poprzednia biblioteka, używa innego, niekompatybilnego XML'a.

Użycie projektu

GtkBuilder

Biblioteka budująca z XML interfejs graficzny.

libglade

Poprzednia biblioteka, używa innego, niekompatybilnego XML'a.

Konwersja plików

`gtk_builder_convert`

Budowanie okna

Ważne

Trzeba pamiętać, że kontrolki mają swoje nazwy.

Budowanie okna

Ważne

Trzeba pamiętać, że kontrolki mają swoje nazwy.

```
builder = gtk.Builder()
builder.add_from_file("wyklad.glade")
window = builder.get_object("okno")
window.show()
gtk.main()
```

Podłączanie sygnałów

Ważne

Podczas budowania interfejsu trzeba wskazać, jakim sygnałom odpowiadają jakie procedury obsługi (handlery).

Podłączanie sygnałów

Ważne

Podczas budowania interfejsu trzeba wskazać, jakim sygnałom odpowiadają jakie procedury obsługi (handlery).

```
builder = gtk.Builder()
builder.add_from_file("wyklad.glade")
builder.connect_signals({ "on_window_destroy" : gtk.main_quit })
window = builder.get_object("okno")
window.show()
gtk.main()
```

Podłączanie menu

Łatwe

Tak samo jak w przypadku innych sygnałów.

Podłączanie menu

Łatwe

Tak samo jak w przypadku innych sygnałów.

Dokładniej:

- trzeba w Glade wskazać procedurę obsługi (wpisać jej nazwę) dla sygnału 'activated';
- powiązać nazwę z prawdziwą procedurą:

```
builder.connect_signals({ "on_window_destroy" : gtk.main_quit,
                          "on_menu_koniec" : lambda widget : akcja(widget) })
```

Bardziej obiektowo

```
class Rysownik:
    def __init__(self):
        builder = gtk.Builder()
        builder.add_from_file("wyklad.glade")
        self.window = builder.get_object("okno")
        builder.connect_signals(self)

    def on_window_destroy(self, widget, data=None): pass

    def koniec(self, widget): pass

rysunek = Rysownik()
rysunek.window.show()
gtk.main()
```

Gazpacho

Inne narzędzie (napisane w PyGTK) do projektowania interfejsów graficznych, produkuje pliki zgodne z GtkBuilder.

Gazpacho

Inne narzędzie (napisane w PyGTK) do projektowania interfejsów graficznych, produkuje pliki zgodne z GtkBuilder.

Kiwi

Z projektem Gazpacho jest związana biblioteka Kiwi (napisana w Pythonie), która w założeniu ma być "lepszym GTK+".

Plan wykładu

Wprowadzenie

- badanie stanu obiektu
- badanie stanu obliczeń

Funkcja standardowa dir()

Co robi dir (przypomnienie)

Zwraca listę dostępnych nazw. Jeśli nie podano argumentu, to podaje listę symboli w lokalnym słowniku.

Funkcja standardowa dir()

Co robi dir (przypomnienie)

Zwraca listę dostępnych nazw. Jeśli nie podano argumentu, to podaje listę symboli w lokalnym słowniku.

>>> dir(1)

['__abs__', '__add__', '__and__', '__class__', '__cmp__', ...]

Analiza biegu programu

- dir() — lista zmiennych lokalnych
- locals() — lokalny słownik zmiennych
- globals() — globalny słownik: klasy, funkcje etc.

Dynamiczne wykonywanie programu

Instrukcja exec (Python 2.*)

```
exec "print 2 + 2"
exec open("foo.py")
```

Funkcja eval()

```
print eval('2*2')
```

Dynamiczne wykonywanie programu

Instrukcja exec (Python 2.*)

```
exec "print 2 + 2"
exec open("foo.py")
```

Funkcja eval()

```
print eval('2*2')
```

Python 3.*

exec nie jest już instrukcją, jest funkcją:

exec(*string_lub_kod*, *globals*, *locals*)

Dynamiczna kompilacja kodu

Wbudowana funkcja compile(*źródło*, *typ źródła*, *typ kodu*) zwraca obiekt reprezentujący skompilowany kod.

Przykład

```
x = 10
code = compile("print x\nprint 2*x", "<string>", "exec")
exec code
```

Problem z kontekstem wywołania

Skompilowanie kodu

```
x = 10
code = compile("print x\nprint 2*x", "<string>", "exec")
wykonaj(code)
```

```
def wykonaj(kod):
    exec kod
```

Przekazanie kontekstu wywołania

Skompilowanie kodu

```
x = 10
code = compile("print x\nprint 2*x", "<string>", "exec")
wykonaj(code)
```

```
def wykonaj(code):
    loc = {"x" : 5}
    exec code in loc
```

Plan wykładu

Rozszerzanie właściwości funkcji

```
def szalenie_skomplikowana_funkcja(arg1, arg2, arg3):
    ...
```

Śledzenie wywołania funkcji

Chcemy śledzić wywołania zaimplementowanych funkcji, tj. informacje o wywołaniu oraz informacja o argumentach wywołania. Bez ingerowania w te funkcje.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Schemat rozwiązania

Rozwiązanie 1.

```
def log_foo(*args):
    print "Wywoływana funkcja: foo z argumentami", args
    return foo(*args)
```

Co z tym zrobić 1.

Zamiast *foo* używamy *log_foo*.

Co z tym zrobić 2.

foo = log_foo

Uniwersalna funkcja opakowująca inne funkcje

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Uniwersalna funkcja opakowująca inne funkcje

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Zastosowanie

foo = log(foo)

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Dekoratory

```
def log(fun):
    def opakowanie(*args):
        print "funkcja:", fun.__name__, "argumenty", args
        return fun(*args)
    return opakowanie
```

Zastosowanie

```
@log
def foo(args):
    ...
```

Dekoratory standardowe

Dekorowanie programów wielowątkowych

```
from threading import Lock
my_lock = Lock()

@synchronized(my_lock)
def critical1(): ...

@synchronized(my_lock)
def critical2(): ...
```

Implementacja dekoratora

```
def synchronized(lock):
    def wrap(f):
        def new_function(*args, **kw):
            lock.acquire()
            try:
                return f(*args, **kw)
            finally:
                lock.release()
        return new_function
    return wrap
```