Powiew świeżości w KDE – SuperKaramba i Python

Python w rytmie Karamby

SuperKaramba umożliwia wprowadzenie interesujących zmian w pulpicie KDE. A przy pomocy języka programowania Pyhton możemy dokładnie określić stopień dodatkowej funkcjonalności, zgodnie z naszymi potrzebami.

HAGEN HÖPFNER

Gdy tylko użyjemy SuperKaramby [2] do umieszczenia zegara i odtwarzacza plików MP3 w tle KDE [1], natychmiast zaczniemy myśleć o czymś więcej. Mamy dobrą wiadomość – aby zwiększyć zakres i funkcjonalność interaktywnych funkcji kompozycji Karamby możemy użyć języka programowania Python [3]. Możemy na przykład tworzyć menu, modyfikować dynamicznie teksty czy też dodać obsługę przeciągania i upuszczania do swoich kompozycji pulpitu.

Główny pomysł jest jednak inny – kiedy pojawia się zdarzenie KDE (np. poruszenie myszą, uruchomienie programu lub wybór pozycji menu), program odpowiedzialny za to zdarzenie wysyła informację, która jest odbierana dzięki odpowiedniej funkcji. SuperKaramba potrafi dokonywać interpretacji tego typu zdarzeń. SuperKaramba Python **API** (czyli interfejs programowy aplikacji), opisany szczegółowo pod adresem [5], umożliwia nam dokładne określenie, co się ma zdarzyć w danej sytuacji.

Czarujący wąż

Jeżeli chcemy wykorzystać Pythona do zwiększenia funkcjonalności naszych kompozycji, omówionych pod adresem [1], nie musimy modyfikować istniejącego kodu. Wystarczy, że dodamy kolejny plik do katalogu z kompozycją. Plik będzie nosił tą samą nazwę co kompozycja, a rozpoznawany będzie dzięki swojemu rozszerzeniu *–.py*. Programiści SuperKaramby zalecają rozpoczęcie pracy z szablonem [6] i jego późniejszą modyfikację. Przykładową kompozycję znajdziemy pod adresem [4].

Rozpakowujemy dane poleceniem:

```
tar -C ~/.superkaramba -xjvf ➥
small_text_xmms.tar.bz2
```

w katalogu *~/.superkaramba* (możliwe, że będziemy musieli najpierw stworzyć katalog o tej nazwie). Po wykonaniu tego polecenia utworzony zostanie katalog kompozy-

cji pulpitu o nazwie *small_text_xmms*. Kopiujemy teraz do tego katalogu plik szablonu *template.py*. Pamiętajmy o zmianie nazwy szablonu, aby odpowiadała nazwie naszej kompozycji:

```
cp template.py ➥
~/.superkaramba➥
/small_text_xmms➥
/small_text_xmms.py
```

Z głównego okna SuperKaramby wybieramy polecenie *Open...* i otwieramy plik kompozycji *small_text_xmms.theme*, co spowoduje zamianę kodu źródłowego Pythona na pseudokod Pythona (Ramka 1). Jeżeli katalog *.superkaramba* jest niewidoczny w oknie dialogowym wyboru plików naciskamy klawisz [F8]. Jeżeli na ekranie kompilator wyświetla informacje o błędzie, zalecamy ręczne uruchomienie SuperKaramby przez wpisanie w Xterminalu polecenia *superkaramba* (będzie też konieczne podanie ścieżki dostępu).

Przykładowo, komunikat typu *Moje rozszerzenie Python zostało pomyślnie załadowane!*, można wywołać dopisując polecenie *print* w ostatniej linijce szablonu:

```
print „Moje rozszerzenie Python ➥
zostało pomyślnie załadowane!"
```

Daje nam to pewność, że plik Pythona został prawidłowo przetworzony

Niestety nie udało się to w przypadku naszej instalacji SuSE 9. Przypuszczamy że dystrybucja ustawia nieprawidłowo jakieś zmienne globalne, z których korzysta Python. Wpisanie poleceń w *konsolę* przed uruchomieniem SuperKaramby szybko rozwiązało ten problem:

```
export PYTHONPATH=/usr/lib➥
/python2.3/
export PYTHONHOME=/usr/lib➥
/python2.3/
```

Przeciąganie i upuszczanie

Naszym pierwszym zwiększeniem możliwości SuperKaramby będzie dołączenie możliwości przeciągania i upuszczania plików *.mp3*, w sposób opisany pod adresem [1]. Sposób ten umożliwia użytkownikom przeciąganie pliku z przeglądarki Konqueror do zintegrowanego odtwarzacza plików MP3. Obecnie chcielibyśmy aby odtwarzaniem plików zajmował się XMMS.

Na Listingu 1 pokazano nasze rozwiązanie. Jeżeli nie chcemy korzystać z pliku szablonu, zastępujemy plik *small_text_xmms.py* plikiem z Listingu 1.

Aby SuperKaramba mogła obsługiwać kompozycję pulpitu z funkcją przeciągania i upuszczania, musimy wykonać jego inicjalizację. Najlepszą metodą jest utworzenie obiektu zwanego **widget** SuperKaramby Transmituje on sygnał rozpoznawany przez jeden ze „wskaźników" SuperKaramby – jest to tzw. funkcja *initWidget()*, a więc wpisujemy:

```
def initWidget(widget):
```

co umożliwi zdefiniowanie odpowiedniej reakcji w następnej linijce naszej kompozycji:

```
karamba.acceptDrops(widget)
```

W tym przypadku chcemy aby widget akceptował porzucone zdarzenia.

Do obsługi sygnału SuperKaramba wykorzystuje *itemDropped()*. Funkcja oczekuje na dwie części informacji z odrzuconego sygnału, opisanego w Tabeli 1: wskaźnik widgeta oraz – w przypadku plików – lista nazw plików zmiennej *dropText*, w której każda linijka przedstawia pojedynczy plik.

Niestety przed każdą nazwą pliku musi pojawić się słowo kluczowe *file:*. Jako że XMMS nie obsługuje pełnych adresów URL typu *file:/ścieżka dostępu/do/plik.mp3*, lecz oczekuje na składnię typu */ścieżka dostępu/do/plik.mp3*, będziemy musieli usunąć zbędne informacje z *dropText* jeszcze przed przekazaniem ich do odtwarzacza multimedialnego.

Striptiz

Do tego zadania będziemy potrzebowali funkcji zmiany ciągów tekstowych, znajdujących się pod adresem [7] z *import string*. Kolejnym krokiem będzie zdefiniowanie pustej zmiennej *xmms_filename*=", gdzie będziemy przechowywać skróconą wersję listy plików. Następnie możemy wywołać *string.split()*, co spowoduje rozdzielenie listy adresów URL z *dropText* na elementy składowe adresu URL:

```
string.split(string.rstrip(str➥
```

Tabela 1: Sygnały wzorcowe SuperKaramby

Funkcja	Wywoływana przez
initWidget(widget)	Wywołanie obiektu SuperKaramby.
widgetUpdated(widget)	Aktualizacja kompozycji. Przedział czasu między aktualizacjami jest określony w pliku .theme.
widgetClicked(widget, x, y, button)	Kliknięcie myszą w obrębie kompozycji. x i y oznaczają współrzędne (względem kompozycji) miejsca, w którym nastąpiło kliknięcie. button określa klawisz myszy, który został naciśnięty.
widgetMouseMoved(widget, x, y, button)	Ruch myszy w obrębie kompozycji. x i y oznaczają współrzędne (względem kompozycji) miejsca, w którym nastąpiło kliknięcie; button określa numer klawisza oraz stan jego przytrzymania.
menuItemClicked(widget, menu, id)	Wybranie pozycji menu. Przekazuje obsługę menu (patrz uwagi w tekście) oraz pozycję menu, która została wybrana (id).
menuOptionChanged(widget, key, value)	Wybrana pozycja w menu konfiguracji kompozycji. key oznacza obsługę pozycji menu, a value nową wartość (true /prawda/ lub false /fałsz/).
meterClicked(widget, meter, button)	Kliknięcie wskaźnika wyświetlania. meter oznacza obsługę, button określa numer naciśniętego klawisza myszy.
commandOutput(widget, pid, output)	Wywołanie programu przez executeInteractive(), pod warunkiem, że powoduje to wyjście do stdout. pid oznacza identyfikator procesu programu, output zawiera tekst wyjściowy.
itemDropped(widget, dropText)	Obiekty (np. ikony) przeznaczone do operacji przeciągania i upuszczania w kompozycji. dropText oznacza tekst dla danego obiektu (np. jego URL, patrz część "Przeciąganie i upuszczanie").
startupAdded(widget, startup)	Uruchomienie nowej aplikacji przez KDE. Po zakończeniu procedury uruchamiania programu pojawia się sygnał startupRemoved(), a po nim taskAdded().
startupRemoved(widget, startup)	Patrz startupAdded().
taskAdded(widget, task)	Patrz startupAdded().
taskRemoved(widget, task)	Przerwanie pracy programu.
activeTaskChanged(widget, task)	Przeniesienie na pierwszy plan innej aplikacji.

```
(dropText)),'\n')
```

Jako separatora użyjemy znaku nowej linii *\n*. Kolejna funkcja, *str()*, umożliwia obsługę zawartości *dropText* jako ciągu testowego, a *string.rstrip()* usunie w tym czasie niestandardowe znaki z prawej strony

Teraz, kiedy mamy już poszczególne adresy URL, możemy użyć pętli *for* w celu usunięcia przedrostka *file:*. Będziemy je zapisywać pojedynczo w zmiennej *filename*, a następnie wywoływać

```
string.split(➥
filename, 'file:', 1)
```

w celu usunięcia przedrostka *file:* z nazwy pliku. Mówiąc bardziej szczegółowo, rozdzielimy zawartość *filename* na dwie części w pierwszym (*1*) przypadku separatora *file:*. Gdybyśmy tego nie zrobili, wszystkie pliki MP3 zawierające ciąg znaków *file:* zostałyby pocięte na drobne kawałeczki. A więc w rezultacie otrzymujemy listę z dwoma wpisami przechowywanymi w innej zmiennej o nazwie *temp*. Pierwszy element, *temp[0]*, zawiera wyłącznie pusty ciąg tekstowy powstały z pierwszej operacji rozdzielania adresu URL.

Z każdym powtórzeniem pętli dołączamy nazwę pliku do istniejącej zawartości zmiennej *xmms_filename*. Jako że nazwa pliku jest drugim elementem na naszej liście *temp*, dostęp do konkretnej nazwy możemy uzyskać przez *temp[1]*:

```
xmms_filename=xmms_filename + ➥
' \'' + string.rstrip(➥
temp[1]) + '\''
```

Kolejnym zadaniem będzie umożliwienie programowi XMMS odtwarzania plików MP3, których nazwy zawierają spacje. Do tego celu konieczne jest umieszczenie nazw plików w cudzysłowie. Niestety Python także korzysta z cudzysłowów typu ' do oddzielania ciągów znaków, a więc będziemy musieli opuścić ciąg znaków przy pomocy '\" – jest to dość złożona procedura. Jeżeli zmienna *xmms_filename* będzie rzeczywiście zawierała cudzysłowy użycie \ spowoduje, że program XMMS potraktuje znak ' dosłownie.

Po zachowaniu pierwszej listy w *xmms_filename*, możemy skorzystać z

```
karamba.execute('xmms'➥
 + xmms_filename)
```

aby przekazać do XMMS nakaz wywołania nazwy pliku MP3.

Menu proszę!

Stworzenie kolejnego menu, uzupełniającego standardowe rozwijane menu wywoływane prawym przyciskiem myszy to kolejny sposób rozbudowania możliwości kompozycji pulpitu. Może to być menu startowe, uruchamiane przez dwukrotne kliknięcie prawym lub środkowym klawiszem myszy i umożliwiające uruchamianie dodatkowych aplikacji. Jako że zdefiniowanie nieskończonej liczby obszarów reagujących na kliknięcie myszą w pliku *.theme* byłoby nieeleganckie, ograniczymy się tutaj do interfejsu API Pythona.

Aby szybko wypróbować ten interfejs, zastępujemy zawartość pliku *~/.superkaramba/small_text_xmms/small_text_xmms.py* instrukcjami zawartymi na Listingu 2. Z góry przepraszamy zawodowych programistów za sztuczkę ze zmiennymi globalnymi **global variables** oraz funkcjami *selectmenu* i *my_menu*, których użyliśmy tutaj dla uproszczenia.

Listing 1: Kompozycja obsługuje przeciąganie i upuszczanie

```
import karamba
import string

def initWidget(widget):
    karamba.acceptDrops(widget)

def itemDropped(widget, dropText):
    xmms_filename=""
    for filename in string.split(string.rstrip(str(dropText)),"\n"):
        temp=string.split(filename, "file:", 1)
        xmms_filename=xmms_filename + " \"" + string.rstrip(temp[1]) + "\""
    karamba.execute("xmms" + xmms_filename)

print "Moje rozszerzenie Python zostało pomyślnie załadowane!"
```

Rysunek 1: Python tworzy menu.

Funkcja *selectmenu* ma początkowo wartość *0* i będzie w późniejszym czasie reprezentować identyfikator obiektu menu. *my_menu=[[],[]]* tworzy strukturę dla listy dwuelementowej. Lista będzie wykorzystana później w programie Python do przypisania pozycjom menu przydzielonych identyfikatorów.

W odpowiedzi na otwierające się okno kompozycji, *initWidget()* dodaje do niej menu:

```
selectmenu=➥
karamba.createMenu(widget)
```

Wewnętrzna liczba przypisana przez system jest teraz przechowywana w *selectmenu* zmiennych globalnych. Określamy to nazwą *uchwytu (handle)*, gdyż liczba identyfikuje wewnętrznie odpowiednie menu. Nasza konstrukcja przechowuje menu z dwoma pozycjami *my_menu*. Pierwsza z nich...

```
karamba.addMenuItem(widget, ➥
selectmenu, 'konqueror', ➥
'konqueror.png')
```

...tworzy wpis dla *konqueror* w *selectmenu*. Ikona *konqueror.png*, która musi znajdować się w tym samym katalogu, będzie wizualizować nam pozycję menu. Druga pozycja menu *addMenuItem()* działa dokładnie tak samo i wykorzystana jest dla przeglądarki internetowej *opera*:

```
karamba.addMenuItem(widget, ➥
selectmenu, 'opera', ➥
'opera.png')
```

Oczywiście brakuje tutaj najważniejszej rzeczy – funkcjonalności. Aby przypisać funkcjonalność do właściwych pozycji menu, potrzebujemy sposobu rozróżnia-

nia poszczególnych pozycji. Możemy zachować przeglądarki *konqueror* lub *opera* przy pomocy pozycji *addMenuItem()* z *my_menu*. Zawartość zmiennych będzie wyglądała następująco, w zależności od uchwytu (handle):

```
(['konqueror', -22], ➥
['opera', -23])
```

Stworzyliśmy zatem menu, ale nie będzie ono wyświetlane od razu, lecz po podwójnym kliknięciu myszą w obszarze kompozycji SuperKaramba; a więc zawsze wtedy gdy wygenerowany zostanie sygnał *widgetClicked(widget, x, y, button)*.

Nie będziemy teraz oceniać współrzędnych *x-y* czy numeru klawisza myszy pozycji *button*, ale gdybyśmy to zrobili, moglibyśmy obsługiwać wiele różnych obszarów na wiele różnych sposobów w zakresie jednej kompozycji, czy też rozróżniać kliknięcia lewym i środkowym klawiszem myszy Moglibyśmy przypisać funkcję także do prawego klawisza myszy jednak kliknięcie prawym klawiszem myszy powoduje rozwinięcie menu SuperKaramby więc jest to możliwość czysto teoretyczna.

Zauważmy że kliknięcia myszą będą zawsze wykonywały akcję zdefiniowaną w pliku *.theme*. W naszym przypadku oznacza to, że kliknięcie zegara otworzy zarówno narzędzie ustawiania czasu jak i menu (Rysunek 1).

Do wyświetlenia menu na ekranie potrzebujemy uchwytu (handle), który przechowywany jest w zmiennej *selectmenu*. Aby odczytać tę zmienną z globalnej przestrzeni danych, możemy skorzystać z *global selectmenu*. Funkcja *karamba.popupMenu(widget, selectmenu, x, y)* umożliwi wyświetlenie menu o tych samych współrzędnych *(x, y)* co miejsce kliknięcia myszą. Jeżeli będzie brakować tych danych, menu pojawi się w lewym górnym rogu (0,0) kompozycji.

Listing 2: Interakcja za pomocą menu

```
import karamba

selectmenu=0
my_menu=[[],[]]

def initWidget(widget):
 global selectmenu
 global my_menu
 selectmenu=karamba.createMenu(widget)
 my_menu=[
 „konqueror", karamba.addMenuItem(widget, selectmenu, „konqueror",
„konqueror.png")],
 ['opera', karamba.addMenuItem(widget, selectmenu, 'opera', 'opera.png')]

def widgetClicked(widget, x, y, button):
 global selectmenu
 karamba.popupMenu(widget, selectmenu, x, y)

def menuItemClicked(widget, menu, id):
 global my_menu
 for index in my_menu:
 if index[1] == id:
 if index[0] == 'konqueror':
 karamba.execute('konqueror')
 elif index[0] == 'opera':
 karamba.execute('opera')

print 'Moje rozszerzenie Python zostało pomyślnie załadowane!'
```

Kiedy użytkownik wybiera pozycję menu, SuperKaramba uruchamia funkcję *menuItemClicked(widget, menu, id)*. Funkcja ta korzysta z *id* do przydzielenia uchwytu pozycji menu. Jako że system przydziela uchwyt (handle) przypadkowo, nie ma w zasadzie żadnego sposobu na jego odgadnięcie, problem można obejść, przechowując id w zmiennych globalnych: *my_menu* dla pozycji menu i *selectmenu* dla samego menu. Z przestrzeni danych globalnych będziemy potrzebować wyłącznie *main_menu*.

for index in my_menu: pozwala nam na przejście kolejno przez wszystkie elementy zmiennej *my_menu* i przypisanie każdej z list, które się tam znajdują, do zmiennej *index*. Podczas

SŁOWNICZEK

API: Interfejs programowy aplikacji definiuje wiele funkcji wewnętrznych oprogramowania, które mogą wywołać inne programy napisane w określony sposób. W naszym przypadku aplikacje Pythona mogą korzystać z funkcji umieszczonych w SuperKarambie.

Stdout: Standardowy strumień wyjściowy określa, dokąd aplikacje Linuksa powinny wysyłać swoje tekstowe dane wyjściowe. Zwykle miejscem tym jest ekran monitora, ale elementem wyjściowym może być równie dobrze plik lub drukarka. Poza stdout, system Linux posiada także strumień stderror, umożliwiający przesyłanie komunikatów o błędach, które mogą być także wyświetlane na ekranie oraz strumień stdin dla operacji wejściowych, oznaczających zwykle przyjmowanie znaków z klawiatury komputera.

Widget: Ogólny termin określający elementy sterujące interfejsu graficznego. Mogą to być okna, przyciski, menu, listy kontrolne czy zakładki.

Wielowątkowość: Programy zwykle nie są w stanie przeprowadzić kilku czynności w tym samym czasie. Przykładowo, kiedy program oczekuje na otwarcie pliku, nie może on wykonać żadnych innych czynności. Wątki dają programowi możliwość równoczesnej pracy nad różnymi zadaniami. Zamiast bezczynnego oczekiwania na otwarcie pliku, program wielowątkowy przypisałby inne zadanie do kolejnego wątku.

Zmienna globalna: Paradygmat programistyczny, obejmuje funkcje i zmienne obiektów.

Przykładowo, opisanie zmiennej dla koloru przycisku ułatwia tworzenie kolejnych przycisków w innych kolorach, gdyż zmienna przechowywana jest w obiekcie „przycisk". Zmienne globalne istnieją przez cały czas pracy programu. Jeśli przypiszemy kolor zielony jako kolor przycisku, wszystkie nowo utworzone przyciski będą miały taki kolor.

każdego przejścia musimy sprawdzać, czy sygnał pozycji menu odpowiada wartości przechowywanej w drugiej pozycji *index*:

```
if index[1] == id:
```

Po odnalezieniu pary *my_menu* odpowiadającej kliknietej pozycji, musimy sprawdzić, do którego wpisu o przeglądarkach odnosi się to kliknięcie. Jeżeli jest to przeglądarka Konqueror..

```
if index[0] == 'konqueror':
```

..., uruchomimy tą przeglądarkę wywołując polecenie *karamba.execute('konqueror')*; jeśli nie – użyjemy polecenia

```
elif index[0] == 'opera':
```

dzięki czemu upewnimy się, czy kliknięto przeglądarkę Opera.

Jeśli połączymy teraz funkcje zawarte na Listingu 1 i 2, tworząc jeden plik, możemy opuścić kilka linijek. Przykładowo, będziemy potrzebować tylko jednej wersji polecenia *import karamba* znajdującego się na początku oraz jednego polecenia *print* na końcu pliku. Musimy ponadto połączyć zawartość bloków *„def initWidget(widget):”*, w celu stworzenia jednolitej wersji. Jeżeli po załadowaniu kompozycji będzie brakowało jednej z funkcji, kasujemy plik *.pyc*, który jest automatycznie tworzony w katalogu kompozycji.

Granice królestwa Pythona

Mimo że obsługa tak skomplikowanych zadań, jak przeciąganie i upuszczanie Pytona SuperKarambie, jest niezwykle prosta i mimo tego, że interfejs API Pythona zapewnia szereg dodatkowych funkcji, dokładnie udokumentowanych pod adresem [5], pojawia się jednak kilka problemów z interfejsem API. Największym z ich jest fakt, że nie może on obsługiwać czujników dynamicznych (1]). Gdybyśmy chcieli dodać do naszej kompozycji funkcję obsługującą przełączanie się pomiędzy kilkoma czujnikami, moglibyśmy kliknąć aktualizację zawartości testowej, ale nie mielibyśmy już wpływu na zamianę wyświetlanych wartości poszczególnych czujników.

Powstaje tutaj kolejny problem. Python potrafi uzyskać dostęp do elementów utworzonych przez siebie. Jeżeli jednak zamierzamy użyć elementu tekstowego, który włączałby jakąś funkcję, musimy utworzyć ją w samym Pytonie przy pomocy funkcji *createText()*. Manipulowanie elementami tekstowymi zdefiniowanymi w pliku *.theme* przy pomocy interfejsu API Pythona jest niemożliwe. ■

INFO

[1] Wstęp do SuperKaramby: Hagen Höpfner, „Karamba na tapecie”, Linux Magazine, 3/2004

[2] SuperKaramba: *http://netdragon.sourceforge.net*

[3] Pierwsze kroki z Pythonem: *http://www.python.org/doc/Intros.html*

[4] Przykładowa kompozycja: *http://wwwiti.cs.uni-magdeburg.de/~hoepfner/download.html*

[5] Dokumentacja interfejsu API dla SuperKaramba Python: *http://netdragon.sourceforge.net/api.html*

[6] Szablon SuperKaramba Python: *http://netdragon.sourceforge.net/template.py*

[7] Funkcje manipulowania ciągami tekstowymi: *http://www.python.org/doc/2.3.3/lib/module-string.html*

[8] Pętle i warunki w Pythonie: *http://www.python.org/doc/2.3.3/ref/compound.html*

[9] Biblioteka Pythona: *http://www.python.org/doc/2.3.3/lib/lib.html*

Ramka 1: Czym jest Python?

Jeżeli znasz już jakieś inny język programowania, nie będziesz zbytnio przerażony perspektywą konieczności nauczenia się nowego języka tylko na potrzeby SuperKaramby. Jednak dla nowicjuszy w świecie programowania Python ma swoje zalety, ponieważ jest językiem uniwersalnym. Jedną z głównych przyczyn takiego stanu rzeczy jest fakt, że Python łączy w sobie trzy główne podstawy programowania. Python może być używany jako język typowo obiektowy, podobnie jak język C++ czy Java, jako język proceduralny, taki jak Pascal lub C, jednocześnie posiada cechy języka skryptowego, podobnego do PHP czy Perl, którego w dodatku nie trzeba kompilować. Ponadto język Python obsługuje dziedziczenie wielokrotne, może łączyć się z bazami danych, umożliwia dostęp do protokołów sieciowych i obsługuje programowanie **wielowątkowe**.

Podobnie jak Java, kod źródłowy Pythona jest najpierw zamieniany na pseudokod przy pomocy kompilatora. Kod pojawia się w katalogu kompozycji SuperKaramby i jest dla niego automatycznie generowany plik z rozszerzeniem .pyc. Dopiero ten kod jest wykonywany przez maszynę wirtualną. Python jest jednak dużo szybszy niż Java.

Jeżeli przesiadasz się z innego języka programowania na język Python, jest jedyna rzecz, do której należy się przyzwyczaić – definicja bloków ciągłych. Bloki wykorzystywane są do definiowania instrukcji, które zostaną wykonane w przypadku spełnienia określonego warunku. Język C używa do tego celu nawiasów klamrowych, w programowaniu powłoki przy pomocy interpretera Bash blok instrukcji warunkowych if-then można przerwać przy pomocy fi. Pascal korzysta ze słów kluczowych begin i end. Python nie używa żadnej z tych metod - wystarczy zaznaczyć wcięcie bloku instrukcji ciągłych.

Zatem dwa występujące po sobie wiersze o tym samym poziomie wcięcia tworzą blok. Jeżeli drugi wiersz będzie rozpoczynać się bliżej lewej strony niż wiersz pierwszy, nie będzie on należał do tego samego bloku. Dwukropek rozpoczyna nowy poziom, jak widać na naszych listingach. Wiersze o tym samym poziomie wcięcia za def należą do definicji tej funkcji.

Niestety Python wymaga przynajmniej jednej funkcji na każdy poziom. Jest to powód, dla którego plik szablonu zawiera funkcję pass zamiast pustego bloku. Funkcja ta nie wykonuje żadnej operacji podczas wykonywania programu i jest swego rodzaju wypełniaczem, który umożliwia wypełnienie poziomu wymaganą funkcją.

Sposób z wcięciem zapewnia elegancką strukturę i łatwość czytania kodu Python w porównaniu z innymi językami programowania. Może jednak doprowadzać początkujących programistów do szału, ponieważ funkcja z niewłaściwym poziomem wcięcia zostanie pominięta przy przetwarzaniu bloku, w którym powinna zostać wykonana. Zatem pamiętajmy o tej pułapce przy eksperymentach z przykładami i tworzeniu własnych rozszerzeń. No i nie zapominajmy o właściwym poziomie wcięć dla poszczególnych bloków!