Совершенствование компонентов

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Компонент **TEdit** разрешает вводить любые символы и представляет результат ввода как текстовое свойство **text**. Поэтому для того,чтобы получить нужное нам поведение (ввод целых чисел), мы

- добавили обработку ошибок в процедуру **onChange**;
- для перевода текстовой строки в число каждый раз использовали функцию **int**

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создается новый компонент, который обладает всеми необходимыми свойствами.

Конечно, можно создавать компонент «с нуля», но так почти никто не делает. Обычно задача сводится к тому, чтобы как-то улучшить существующий стандартный компонент, который уже есть в библиотеке.

Мы будем совершенствовать компонент **TEdit** (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его **TIntEdit**) будет наследником класса **TEdit**, а класс **TEdit** будет соответственно базовым классом для нового класса **TIntEdit**:

```
class TIntEdit ( TEdit ):
  ...
```

Изменения стандартного класса **TEdit** сводятся к двум пунктам:

- все некорректные символы, которые приводят к тому, что текст нельзя преобразовать в целое число, должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение целого типа; для этого мы добавим к нему свойство **value** (англ. значение).

Сначала добавим в новый класс конструктор:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
```

При вызове этого конструктора нужно указать, по крайней мере, один параметр – ссылку на «родительский» объект **parent**. Затем могут следовать именованные параметры (подробности можно найти в Интернете в описании компонента **Entry** библиотеки **tkinter**). Запись с двумя звёздочками ****kw** говорит о том, что они «собираются» в словарь.

В приведённом варианте класс **TIntEdit** ничем не отличается от **TEdit**. Добавим к нему свойство **value**. Для этого введём закрытое поле **__value**, в котором будем хранить последнее правильное числовое значение:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
    self.__value = 0
  def __setValue ( self, value ):
    self.text = str ( value )
  value = property ( lambda x: x.__value, __setValue )
```

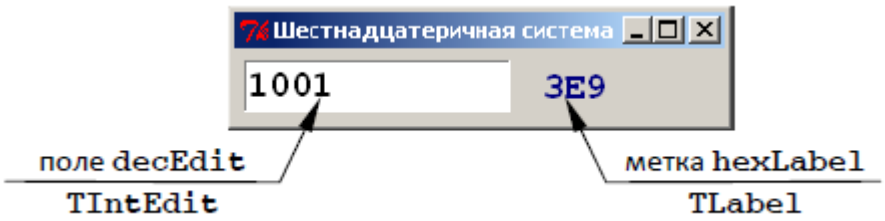
Для чтения введенного числового значения используется «лямбда-функция», которая возвращает значение поля **__value**. Метод записи **__setValue** записывает в свойство **text** переданное число в виде символьной строки.

Для того, чтобы заблокировать ввод нецифровых символов, будем использовать обработчик события **onValidate** (от англ. *validate* – проверять на правильность) компонента **TEdit**. Установим этот обработчик на скрытый метод нового класса **TIntEdit**, который назовём**__onValidate**. Обработчик должен возвращать логическое значение: **True**, если символ допустимый и **False**, если нет (тогда ввод этого символа блокируется). В этом коде оставлены только те строки, который относятся к установке этого обработчика:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    ...
    self.onValidate = self.__validate
  def __validate ( self ):
    try:
      newValue = int ( self.text )
      self.__value = newValue
      return True
    except:
      return False
```

При получении сигнала о событии, обработчик пытается преобразовать новое значение в число. Если это удаётся, полученное число записывается в переменную **__value** и возвращается результат **True**. Если произошла ошибка, функция вернёт **False**. Готовый компонент лучше всего поместить в отдельный модуль, назовем его **int_edit.py**.

Построим новый проект: программа будет переводить целые числа из десятичной системы в шестнадцатеричную:



Импортируем компонент **TIntEdit** из модуля **int_edit**:

```
from int_edit import TIntEdit
```

Создадим объект-приложение:

```
app = TApplication ( "Шестнадцатеричная система" )
app.size = (250, 36)
app.position = (200, 200)
```

Поместим на форму метку **hexLabel** для вывода шестнадцатеричного значения и компонент класса **TIntEdit** для ввода:

```
f = ( "Courier New", 14, "bold" )
hexLabel = TLabel ( app, text = "?", font = f, fg = "navy" )
hexLabel.position = (155, 5)
decEdit = TIntEdit ( app, width = 12, font = f )
decEdit.position = (5, 5)
decEdit.text = "1001"
```

Добавим обработчик события **onChange** для поля ввода:

```
def onNumChange ( sender ):
  hexLabel.text = "{:X}".format ( sender.value )
decEdit.onChange = onNumChange
```

Этот обработчик читает значение свойства **value** объекта-источника события и выводит его наметку **hexLabel** в шестнадцатеричной системе. Формат «**X**» (а не «**x**») означает, что будут использовать заглавные буквы A-F.

Теперь программа готова и можно проверить её работу.

Модель и представление

Одна из важнейших идей технологии быстрого проектирования программ (RAD) – повторное использование написанного ранее готового кода. Чтобы облегчить решение этой задачи, было предложено использовать еще одну декомпозицию: разделить *модель*, то есть данные и методы их обработки, и *представление* – способ взаимодействия модели с пользователем (интерфейс).

Пусть, например, данные об изменении курса доллара хранятся в виде массива, в котором требуется искать максимальное и минимальное значения, а также строить приближенные зависимости, позволяющие прогнозировать изменение курса в ближайшем будущем. Это описание задачи на уровне модели.

Для пользователя эти данные могут быть представлены в различных формах: в виде таблицы, графика, диаграммы и т.п. Полученные зависимости, приближенно описывающие изменение курса, могут быть показаны в виде формулы или в виде кривой. Это уровень представления или интерфейса с пользователем.

Чем хорошо такое разделение? Его главное преимущество состоит в том, что модель не зависит от представления, поэтому одну и ту же модель можно использовать без изменений в программах, имеющих совершенно различный интерфейс.

Вычисление арифметических выражений: модель

Построим программу, которая вычисляет арифметическое выражение, записанное в символьной строке. Для простоты будем считать, что в выражении используются только

- целые числа
- знаки арифметических действий **`+-*/`**

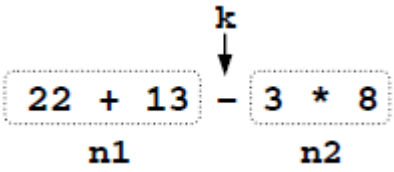
Предположим, что выражение не содержит ошибок и посторонних символов.

Какова модель для этой задачи? По условию данные хранятся в виде символьной строки. Обработка данных состоит в том, что нужно вычислить значение записанного в строке выражения.

Вспомните, что аналогичную задачу мы решали в лекции, где использовалась структура типа «дерево». Теперь мы применим другой способ.

Как вы знаете, при вычислении арифметического выражения последней выполняется крайняя справа операция с наименьшим приоритетом. Таким образом, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке **`s`**:

1. Найти в строке **`s`** последнюю операцию с наименьшим приоритетом (пусть номер этого символа записан в переменной **`k`**)
2. Используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с номером **`k`** и записать результаты вычисления в переменные **`n1`** и **`n2`**.



3. Выполнить операцию, символ которой записан в s[k], с переменными n1 и n2.

Обратите внимание, что в п. 2 этого алгоритма нужно решить ту же самую задачу для левой и правой частей исходного выражения. Как вы знаете, такой прием называется рекурсией.

Основную функцию назовём **`Calc`** (от англ. *calculate* – вычислить). Она принимает символьную строку и возвращает целое число – результат вычисления выражения, записанного в этой строке. Алгоритм её работы на псевдокоде:

```
k = номер символа, соответствующего последней операции
if k < 0: # нет знака операции
  перевести всю строку в число
else:
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
  применить найденную операцию к n1 и n2
```

Для того, чтобы найти последнюю выполняемую операцию, будем использовать функцию **`lastOp`**. Если эта функция вернула «–1», то операция не найдена, то есть вся переданная ей строка – это число (предполагается, что данные корректны).

```
def priority ( op ):
  if op in "+-": return 1
  if op in "*/": return 2
  return 100
def lastOp ( s ):
  minPrt = 50   # любое между 2 и 100
  k = -1
  for i in range(len(s)):
    if priority(s[i]) <= minPrt:
      minPrt = priority(s[i])
      k = i
  return k
```

Теперь можно написать функцию **`Calc`**:

```
def Calc ( s ):
  k = lastOp ( s )
  if k < 0:          # вся строка - число
    return int(s)
  else:
    n1 = Calc ( s[:k] )    # левая часть
    n2 = Calc ( s[k+1:] ) # правая часть
     # выполнить операцию
    if   s[k] == "+": return n1+n2
    elif s[k] == "-": return n1-n2
    elif s[k] == "*": return n1*n2
    else: return n1 // n2
```

Обратите внимание, что функция **`Calc`** – рекурсивная, она дважды вызывает сама себя.

Функции **Calc** и **lastOp** (а также функцию **priority**, которая вызывается из **lastOp**)удобно объединить в отдельный модуль **model.py** (модуль модели). Таким образом, наша модель – это функции, с помощью которых вычисляется арифметическое выражение, записанное в строке.

Вычисление арифметических выражений: представление

Теперь построим интерфейс программы. В верхней части окна будет размещен выпадающий список (компонент **TComboBox**), в котором пользователь вводит выражение. При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится в первой строке обычного списка (компонента **TListBox**). Выпадающий список, в котором хранятся все вводимые выражения, полезен для того, чтобы можно было вернуться к уже введённому ранее варианту и исправить его.

Итак, импортируем из модуля **model** функцию **Calc** и создаём приложение:

```
from model import Calc
app = TApplication ( "Калькулятор" )
app.size = (200, 150)
```

На форму нужно добавить компонент **TComboBox**. Чтобы прижать его к верху, установим свойство **align**, равное **"top"** Назовем этот компонент **Input**

```
Input = TComboBox ( app, values = [], height = 1 )
Input.align = "top"
Input.text = "2+2"
```

При вызове конструктора, кроме родительского объекта, мы указали два именованных параметра: **values** (англ. «значения») – пустой список (сначала в списке нет ни одного элемента) и **height** – высота компонента (одна строка). Добавляем второй компонент – **TListBox**, устанавливаем для него выравнивание **"client"** (заполнить всю свободную область) и имя **Answers**.

```
Answers = TListBox ( app )
Answers.align = "client"
```

Логика работы программы может быть записана в виде псевдокода:

```
if нажата клавиша Enter:
  вычислить выражение
  добавить результат вычислений в начало списка
  if выражения нет в выпадающем списке:
    добавить его в выпадающий список
```

Для перехвата нажатия клавиши **Enter** будем использовать обработчик события «**<Key-Return>**» (англ. «клавиша перевод каретки»), который подключается стандартным способом библиотеки **tkinter** – через метод **bind** (англ. «связать»):

```
Input.bind ( "<Key-Return>", doCalc )
```

Здесь **doCalc** – это название функции, принимающей один параметр – объект-структуру, которая содержит полную информацию о произошедшем событии:

```
def doCalc ( event ):
  ...
```

Вместо многоточия нужно написать операторы **Python**, которые нужно выполнить. Во-первых, читаем текст, введённый в выпадающем списке, и вычисляем выражение с помощью функции **Calc**:

```
expr = Input.text
x = Calc ( expr )
```

Затем добавляем в начало списка вычисленное выражение и его результат:

```
Answers.insert ( 0, expr + "=" + str(x) )
```

Здесь используется метод **insert** (англ. «вставить»), первый аргумент – это номер вставляемого элемента (0 – в начало списка).

Теперь проверим, есть ли такое выражение в выпадающем списке, и если нет – добавим его:

```
if not Input.findItem(expr):
  Input.addItem ( expr )
```

Метод **findItem** (англ. *find item* – найти элемент) возвращает логическое значение: True, если переданная ему строка есть в списке и **False**, если нет. Метод **addItem** (англ. *add item* – добавить элемент) добавляет элемент в конец списка.

Таким образом, полная функция **doCalc** приобретает следующий вид:

```
def doCalc ( event ):
  expr = Input.text
  x = Calc ( expr )
  Answers.insert ( 0, expr + "=" + str(x) )
  if not Input.findItem ( expr ):
    Input.addItem ( expr )
```

Теперь программу можно запускать и испытывать.

Итак, в этой программе мы разделили модель (данные и средства их обработки) и представление (взаимодействие модели с пользователем), которые разнесены по разным модулям. Это позволяет использовать модуль модели в любых программах, где нужно вычислять арифметические выражения.

Часто к паре «модель-представление» добавляют еще управляющий блок (контроллер), который, например, обрабатывает ошибки ввода данных. Но во многих случаях, например, при программировании в RAD-средах, контроллер и представление объединяются вместе – управление данными происходит в обработчиках событий.