



As três Fates

PENSANDO EM TKINTER

Steven Ferg (steve@ferg.org)



Traduzido e adaptado por J. Labaki
labaki@feis.unesp.br
Grupo Python
Departamento de Engenharia Mecânica
UNESP – Ilha Solteira

Índice

Sobre <i>Pensando em Tkinter</i>	3
Os programas	3
TT000 – Introdução	4
As quatro questões básicas da programação de GUIs	4
Alguns jargões da programação de GUIs	4
Sobre o Event Loop	5
TT010 – O programa em Tkinter mais simples possível: três linhas.	6
TT020 – Criando um objeto GUI e fazendo pack; containeres versus widgets.	7
Frames são elásticos	9
TT030 – Criando um Widget e colocando-o no frame.	9
TT035 – Usando a estrutura de classes no programa.	11
Por que estruturar sua aplicação como uma classe?	11
Quando introduzir a estrutura de classes	12
TT040 – Algumas formas diferentes de definir um widget.	13
TT050 – Empacotando.	14
Porque os botões apareceram verticalmente no último programa	14
Alguns termos técnicos – Orientação	15
TT060 – Fazendo Binding.	16
TT070 – Mexendo com foco e ligando eventos de teclado a widgets.	19
TT074 – Command Binding.	22
TT075 – Usando event binding e command binding juntos.	24
Para quais eventos serve command binding?	24
Usando event binding e command binding juntos	25
TT076 – Compartilhando informações entre alimentadores de eventos.	26
Compartilhando informações entre funções alimentadoras de eventos	26
Primeira solução – usar variáveis globais	27
Segunda solução – usar variáveis instanciadas	27
TT077 – Transmitindo argumentos para alimentadores de eventos I: O problema.	29
Características mais avançadas de command binding	29
TT078 – Transmitindo argumentos para alimentadores de eventos II: Usando Lambda.	31
TT079 – Transmitindo argumentos para alimentadores de eventos III: Usando Currying.	35
Sobre Curry	35
Curry – como usá-lo	36
Lambda versus Curry & event_lambda: qual devo usar?	38
TT080 – Opções de widget e configurações de pack	40
Três técnicas de controlar o layout de uma GUI	40
TT090 – Posicionando frames	43
TT095 – Métodos gerenciadores de janelas & controlando o tamanho de janelas com a opção <i>geometry</i>.	46
TT100 – Opções de pack: side, expand, fill e anchor.	49
Um jeito prático de encontrar erros	50

Sobre *Pensando em Tkinter*

Tenho tentado aprender sozinho Tkinter por vários livros, e tenho encontrado mais dificuldade do que poderia pensar.

O problema é que os autores desses livros já começam dizendo tudo sobre os *widgets* disponíveis em Tkinter, mas nunca dão uma parada para explicar os conceitos básicos. Eles não explicam como “pensar em Tkinter”. Então achei que deveria tentar escrever um livro do tipo que eu mesmo estou procurando. Ou ao menos rascunhar esse tipo de livro.

Pensando em Tkinter consiste de pequenos programas que começam a explicar como pensar em Tkinter. Nestes, não me preocupo em catalogar todos os tipos de widgets, atributos e métodos disponíveis em Tkinter, somente em dar um empurrãozinho na direção dos conceitos básicos do assunto.

Estes programas não têm intenção de introduzir todos os aspectos da programação de Tkinter. Para este fim, sugiro que leia “*An Introduction to Tkinter*”, de Frederik Lundh e “*Tkinter, GUI Programming with Python*”, da New México Tech Computer Center.

Acima de tudo, quero enfatizar que “*Practical Programming in Tcl and Tk*” de Brend Welch é absolutamente essencial para trabalhar com Tk e Tkinter. Arranje esse livro!

Note que você não deve rodar estes programas usando o IDLE. O IDLE é por si só uma aplicação de Tkinter; possui seu próprio “mainloop” que entrará em conflito com o mainloop incluído nestes programas. Se você insiste em ver e rodar estes programas usando IDLE, então – *para cada programa* – deverá apagar a linha correspondente ao mainloop antes de rodá-lo.

Este material tem sido substancialmente melhorado pelo feedback dos entusiastas e comunidade pythoniana. Meu grande “Obrigado!” a Alan Colburn, Jeff Epler, Greg Ewing, Tom Good, Steve Holden, Joseph Knapka, Gerrit Müller, Russel Owen, e Chad Netzer. Obrigado a Terry Carroll por recolher e organizar esse material.

Finalmente, se você é novo na programação orientada a eventos (que Tkinter requer), deve dar uma olhada neste assunto antes.

Os Programas

Você deve baixar o arquivo zip que contém todos os programas. Pode ser através do seguinte endereço:

http://geocities.yahoo.com.br/grupopython/pensando_em_tkinter.zip

O pacote contém os programas de “Pensando em Tkinter”. Para instalar estes arquivos, simplesmente descompacte o pacote de arquivos em um diretório de sua escolha.

TT000 – Introdução.

Eis alguns programas que começarão a explicar como pensar em Tkinter. Neles, como já disse, não atendo para todos os tipos de widgets, atributos e métodos disponíveis em Tkinter, tampouco tento dar uma introdução detalhada sobre Tkinter, somente tento iniciar você no caminho para entender os seus conceitos básicos.

No decorrer do programa de ensino, a discussão é dedicada exclusivamente ao gerenciador de geometria *pack*. Não falaremos sobre os gerenciadores *grid* ou *place*.

As quatro questões básicas na programação de GUIs.

Quando você desenvolve uma interface com o usuário (IU) há um conjunto padrão de questões que você deve satisfazer.

- 1) Você deve especificar como você quer que a IU se *pareça*. Isto é, você precisa escrever um código que determina o que o usuário verá na tela do computador;
- 2) Você deve decidir o que quer que a IU *faça*. Isto é, você deve escrever procedimentos que satisfaçam as necessidades do programa;
- 3) Você precisa associar o *parecer* com o *fazer*. Isto é, você deve escrever um código que associa as coisas que o usuário vê na tela com os procedimentos que você escreveu para desempenhar os papéis requeridos pelo programa;
- 4) Finalmente, você deve escrever um código que senta e espera pela entrada do usuário.

Alguns jargões da programação de GUIs.

A programação GUI (Graphic User Interface – Interface Gráfica com o Usuário) tem alguns jargões especiais associados às suas questões básicas.

- 1) Nós especificamos como queremos que um GUI se pareça descrevendo os “*widgets*” que queremos exibir, e suas relações especiais (por exemplo, o quanto um widget está abaixo ou acima, ou à direita ou à esquerda de outros widgets). A palavra “*widget*” é um termo sem tradução que designa “componentes de interface gráfica com o usuário” de um modo geral. *Widgets* inclui elementos como janelas, botões, menus e itens de menus, ícones, listas rolantes, barras de rolagem, etc.
- 2) Os procedimentos que executam as tarefas dos GUIs são chamados “*event handler*”. “Events” são formas de entrada de dados como cliques de mouse ou digitação no teclado. Esses procedimentos são chamados “handlers” (alimentadores) porque eles “alimentam” (isto é, respondem a) estes eventos.
- 3) A associação de um event handler a um widget é chamada “binding”. De modo geral o processo de binding envolve a associação entre três coisas diferentes:

- a. Um tipo de evento (por exemplo, um clique no botão esquerdo do mouse, ou pressionar a tecla ENTER),
- b. Um widget (por exemplo, um botão) e
- c. Um procedimento event handler.

Por exemplo, nós precisamos fazer binding (a) num clique com o botão esquerdo do mouse no (b) botão “FECHAR” na tela para (c) executar o procedimento “fechePrograma”, que fechará a janela e desligará o programa.

- 4) O código que senta e espera pela entrada de dados é chamada de “event loop”.

Sobre o Event Loop

Se você tem visto filmes, sabe que toda cidadezinha tem uma vovó que perde todo seu tempo debruçada na janela, só *olhando*. Ela vê tudo que acontece na vizinhança. Uma parte disso não tem graça, é claro – só pessoas subindo e descendo a rua. Mas a outra é interessante – como uma bela briga entre os pombinhos recém-casados do outro lado da rua. Quando algo interessante acontece, essa vovozinha cão-de-guarda imediatamente corre para o telefone contar tudo à polícia da vizinhança.

O *Event Loop* é algo como essa vovozinha: gasta todo seu tempo esperando que eventos aconteçam. Muitos dos eventos não têm importância (como clicar num ponto neutro da janela), e quando os vê, não faz nada. Mas quando vê alguma coisa interessante – um evento que ele sabe, por meio de um *binding* de *event handler*, que é interessante (como um clique num dos botões da janela) – então imediatamente contata os event handler e faz com que saibam que o evento aconteceu.

Comportamento do Programa

Este programa facilita a você o entendimento da programação de interfaces com o usuário mostrando como estes conceitos básicos são implementados em um programa muito simples. Este programa não usa Tkinter ou qualquer forma de programação GUI, somente coloca um menu e um console, e recebe caracteres digitados no teclado como entrada. Assim, como você pode ver, ele satisfaz as quatro questões básicas da programação de interfaces com o usuário.

```
#Questão 2: Define os procedimentos de event handler
def handle_A():
    print "Wrong! Try again!"

def handle_B():
    print "Absolutely right! Trillium is a kind of flower!"

def handle_C():
    print "Wrong! Try again!"

#Questão 1: Define a aparência da tela
print "\n"*100 # clear the screen
print "          VERY CHALLENGING GUESSING GAME"
print "=====
```

```
print "Press the letter of your answer, then the ENTER key."
print
print "    A.  Animal"
print "    B.  Vegetable"
print "    C.  Mineral"
print
print "    X.  Exit from this program"
print
print "===== "
print "What kind of thing is 'Trillium'?"
```

```
#Questão 4: O Event Loop. Loop eterno, esperando que algo aconteça.
while 1:

    # Observamos o próximo evento
    answer = raw_input().upper()
    # -----
    # Questão 3: Associamos os eventos de teclado que nos interessam
    # com seus event handlers. Uma forma simples de binding.
    # -----
    if answer == "A": handle_A()
    if answer == "B": handle_B()
    if answer == "C": handle_C()
    if answer == "X":
        # clear the screen and exit the event loop
        print "\n"*100
        break

#Perceba que quaisquer outros eventos não interessam, por isso são
ignorados.
```

TT010 – O programa em Tkinter mais simples possível: três linhas.

Das quatro questões básicas na programação de GUIs que nós vimos no último programa, este programa cumpre somente uma – ele roda o event loop.

(1)

A primeira linha importa o módulo Tkinter e deixa-o disponível para uso. Perceba que a forma de importar (“from Tkinter import *”) significa que nós não queremos ter que usar a forma “Tkinter.” para especificar nada que quisermos utilizar.

(2)

A segunda linha cria uma janela toplevel (que pode ou não se tornar visível). Tecnicamente, o que esta linha faz é criar uma instância da classe “Tkinter.Tk”.

Esta janela toplevel é o componente GUI de mais alto nível¹ de qualquer aplicação de Tkinter. Por convenção, esta janela é normalmente chamada de “raiz”.

(3)

A terceira linha executa o método *mainloop* (isso é, o event loop) deste objeto *raiz*. Assim que roda, o mainloop espera que eventos aconteçam no objeto raiz. Se um evento ocorre, então ele é alimentado (o event handler é executado) e o loop continua rodando, esperando

¹ Entenda por “de alto nível” a aplicação que tem mais relação com o usuário que com o código (N do T).

pelo próximo evento, ou até que aconteça um evento que “destrua” a raiz. Um evento destruidor pode ser o fechamento da janela pelo botão X de fechamento. Quando a raiz é destruída, a janela é fechada e o event loop cancelado.

Comportamento do programa

Ao rodar este programa, você verá (graças ao Tk) a janela toplevel automaticamente com os widgets para minimizar, maximizar e fechar a janela. Tente usá-los – você verá que eles realmente funcionam.

Clicando no widget “fechar” (o X em uma caixa, do lado direito da barra de título) será gerado um evento destruidor terminando o event loop principal, que no caso deste programa é *mainloop*. E desde que não haja mais nada depois da linha “`root.mainloop()`”, como neste caso, o programa não faz mais nada e se encerra.

```
from Tkinter import * ### (1)
root = Tk()          ### (2)
root.mainloop()     ### (3)
```

TT020 – Criando um objeto GUI e fazendo pack; containeres versus widgets.

Agora daremos uma pincelada em outra das quatro questões básicas da programação GUI – especificar como a GUI deverá parecer.

Neste programa, introduzimos três dos maiores conceitos da programação em Tkinter:

- criar um objeto GUI e associá-lo com seus mestres;
- pack e
- container versus widget.

De agora em diante, distinguiremos os componentes entre containeres e um widgets. Como usarei sempre estes termos, um widget é um componente GUI que (geralmente) é visível e faz coisas. Já um container é simplesmente um container – uma cesta, como queira – dentro do qual dispõem-se os widgets.

Tkinter oferece vários tipos de containeres. “Canvas” é um container para aplicações de desenho, e o container mais freqüentemente utilizado é o “Frame”.

Frames são oferecidos pelo Tkinter como uma classe chamada “Frame”. Uma expressão como:

```
Frame (meuMestre)
```

cria uma instância da classe `Frame` (isto é, cria um `frame`), e associa a instância ao seu mestre, `meuMestre`. Outra maneira de ver isso é como uma expressão que adiciona um `frame` “escravo” ao componente `meuMestre`.

Então, neste programa (linha 1),

```
myContainer1 = Frame(myParent)
```

cria um `frame` cujo mestre é `myParent` (isto é, a raiz), e dá a ele o nome de “`myContainer1`”. Resumindo, ele cria um `container` dentro do qual podemos agora colocar `widgets`. (Nós não colocaremos nenhum `widget` neste programa, somente posteriormente).

Perceba que a relação mestre/escravo aqui é somente lógica, não tem nada de visual. Esta relação existe para otimizar eventos do tipo destrutivo – isso porque quando um componente mestre (como `root`) é destruído, o mestre sabe quem são seus escravos, e pode destruí-los antes de se destruir.

(2)

A próxima linha define o gerenciador de geometria “`pack`” para administrar `myContainer1`.

```
myContainer1.pack()
```

Simplesmente designado, “`pack`” é um método que transforma em visuais as relações entre os componentes GUI e seus mestres. Se você não definir o componente `pack` ou outro gerenciador de geometria, nunca verá a GUI.

Um gerenciador de geometria é essencialmente um API – um meio de dizer ao Tkinter como você quer que `containers` e `widgets` se apresentem visualmente. Tkinter oferece três gerenciadores para esta finalidade: `pack`, `grid` e `place`. `Pack` e `grid` são os mais usados por serem mais simples. Todos os exemplos em “Pensando em Tkinter” usam `pack` como gerenciador de geometria.

O esquema padrão básico da programação em Tkinter, que veremos ainda diversas vezes, funciona mais ou menos assim:

- uma instância (de um `widget` ou um `container`) é criada, e associada ao seu mestre;
- a instância é administrada por um gerenciador de geometria.

Comportamento do programa

Quando você roda este programa, ele se parecerá muito com seu antecessor, exceto pelo tamanho. Isso é porque...

Frames são elásticos

Um frame é basicamente um container. O interior de um container – o “espaço” existente dentro dele – é chamado “cavidade”, um termo técnico que Tkinter herdou de Tk. Essa cavidade é extensível (ou elástica) como uma borracha. A menos que você especifique um tamanho máximo ou mínimo para o frame, a cavidade será esticada ou comprimida até acomodar qualquer coisa que o frame contiver.

No programa anterior, por não termos colocado nada dentro dela, a raiz mostrou a si mesma na tela com seu tamanho padrão, mas neste programa, nós preenchemos sua cavidade com o `myContainer1`. Agora, a raiz se estica para acomodar o tamanho de `myContainer1`, mas como não colocamos nenhum widget neste frame, nem especificamos um tamanho mínimo para ele, a cavidade da root se encolhe até o limite. Por isso não há nada para ser visto abaixo da barra de título desse programa.

Nos próximos programas, colocaremos widgets e outros containeres dentro do `Container1`, e você verá como ele se arranja para acomodá-los.

```
from Tkinter import *

root = Tk()

myContainer1 = Frame(root)   ### (1)
myContainer1.pack()        ### (2)

root.mainloop()
```

TT030 – Criando um Widget e colocando-o no frame.

Neste programa, nós criamos nosso primeiro widget e o colocamos dentro de `myContainer1`.

(1)

O Widget será um botão – isto é, ele será uma instância da classe “Button” de Tkinter. A linha:

```
Button1=Button(myContainer1)
```

cria o botão, dando-o o nome de “button1”, e associa-o ao seu mestre, o objeto container chamado `myContainer1`.

(2)(3)

Os widgets tem muitos atributos, todos disponíveis no dicionário do *namespace* local. O widget “Button” tem atributos para controlar seu tamanho, sua cor de fundo e de fonte, seu texto, como suas bordas se parecerão, etc. Neste exemplo, nós iremos ajustar somente dois

atributos de `button`: a cor de fundo e o texto. Faremos isso mudando os valores do seu dicionário referentes às chaves “text” e “background”.

```
button1["text"] = "Hello, World!"  
button1["background"] = "green"
```

(4)

E, é claro, nós fazemos `pack` no botão `button1`.

```
button1.pack()
```

Comportamento do programa

Quando você rodar este programa, deverá ver que o `Container1` agora contém um botão verde com o texto “Hello, World!”. Quando você clica nele não acontece nada, porque nós ainda não especificamos o que queremos que aconteça quando o botão for clicado, se bem que o faremos mais tarde.

Por ora, você deverá clicar no botão X da barra de título para fechá-lo, como antes.

```
from Tkinter import *  
  
root = Tk()  
  
myContainer1 = Frame(root)  
myContainer1.pack()  
  
button1 = Button(myContainer1)          ### (1)  
button1["text"] = "Hello, World!"      ### (2)  
button1["background"] = "green"       ### (3)  
button1.pack()                         ### (4)  
  
root.mainloop()
```

TT035 – Usando a estrutura de classes no programa.

Neste programa, introduziremos o conceito de aplicações de Tkinter estruturadas como classes.

Nele, criamos uma classe chamada `MyApp` e transcrevemos alguns códigos dos programas anteriores para dentro de seu método construtor (`__init__`). Nesta versão reestruturada do programa, fazemos 3 coisas diferentes:

(1)

Em nosso código, designamos uma classe (`MyApp`) que define como queremos que a GUI se pareça e que tipo de coisa queremos fazer com isso. Todo este código é inserido no método construtor da classe.

(2)

Quando o programa é executado, a primeira coisa que ele faz é criar uma instância da classe. A linha que cria a instância é

```
myapp=MyApp(root)
```

Perceba que o nome da classe é “MyApp” (observe o jogo de maiúsculas) e o nome da instância é “myapp” (agora tudo minúsculo).

Perceba também que essa linha faz de “root”, a raiz, um argumento dentro do método construtor de MyApp. O método construtor reconhece root sob o nome “myParent”.

(3)

Finalmente, deixamos a raiz em mainloop.

Por que estruturar sua aplicação como uma classe?

Uma das razões para usar uma estrutura de classes em seu programa é simplesmente controlá-lo melhor. Um programa estruturado em classes é provavelmente – especialmente se seu programa for muito grande – muito mais fácil de ser entendido.

Uma consideração muito importante é que estruturar sua aplicação como uma classe ajudará você a evitar o uso de variáveis globais. Eventualmente, conforme seu programa for crescendo, você provavelmente irá querer que alguns de seus event handler consigam compartilhar informações entre si. Uma maneira é usar variáveis globais, mas é uma técnica muito maçante. Um caminho muito melhor é usar instâncias (isto é, usar “self.” nas variáveis), e para isso você precisa estruturar sua aplicação como classes. Exploraremos esse tópico em outros programas desta série.

Quando introduzir a estrutura de classes

Temos introduzido a noção de estrutura de classes para programas de Tkinter logo cedo para podermos explicá-la e partir para outros assuntos. Porém, no andar da carruagem, você poderá escolher proceder de outra forma.

Em muitos casos, um programa em Tkinter começa com um script simples. Todo o código acontece numa única linha, como nossos programas anteriores. Então, conforme a aplicação toma corpo, o programa cresce, de forma que em pouco tempo você se verá envolvido por uma porrada de código. Você pode começar a usar variáveis globais... talvez uma porrada de variáveis globais. O programa começa a ficar difícil de entender e editar. Quando isso acontece, é hora de voltar à prancheta e refazer tudo, desta vez usando classes.

Por outro lado, se você se sentir bem com classes, e tem uma boa idéia da forma final do seu programa, comece estruturando-o em classes desde o começo.

Por outro lado (voltamos assim ao lado anterior?), logo no começo do processo de desenvolvimento (como observou Gerrit Muller) freqüentemente você ainda não sabe a

melhor estrutura de classes a usar – logo no começo, você simplesmente não tem uma idéia clara o suficiente do problema e solução. Começar a usar classes cedo demais no processo pode gerar muitas estruturas desnecessárias que só servem para bagunçar o código, além de geralmente exigir mais consertos.

Tudo depende do gosto pessoal e das circunstâncias. Faça o que parecer bem a você. E – não importa o que você escolha – não se assuste com as reestruturações quando se tornarem necessárias.

Comportamento do programa

Quando você rodar este programa, ele terá uma aparência exatamente igual à anterior. Nada funcional foi mudado – somente como o código foi estruturado. Agora ele está em forma de classes.

```
from Tkinter import *

class MyApp:
    def __init__(self, myParent):
        self.myContainer1 = Frame(myParent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"] = "Hello, World!"
        self.button1["background"] = "green"
        self.button1.pack()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT040 – Algumas formas diferentes de definir um widget.

(1)

No programa anterior, nós criamos um objeto Button, button1, e então mudamos seu texto e cor de fundo mais ou menos assim:

```
self.button1["text"] = "Hello, World!"
self.button1["background"] = "green"
```

Neste programa, adicionamos mais três botões ao Container1, usando métodos ligeiramente diferentes.

(2)

Para o botão button2, o processo é essencialmente o mesmo que no botão button1, mas em vez de acessar o dicionário de Button, usamos o método “configure” atribuído aos objetos Button.

(3)

Para o botão `button3`, vemos que o método `configure` pode ter muitos argumentos, então designamos várias opções em uma única linha.

(4)

Nos exemplos anteriores, fazer um botão era uma tarefa em dois passos: primeiro criar o botão e então configurar suas propriedades. Também é possível configurar estas propriedades ao mesmo tempo em que criamos o botão. O widget “Button”, como qualquer widget, espera como primeiro argumento o nome do seu mestre. Depois desse argumento você pode, se desejar, adicionar um ou mais argumentos sobre as propriedades do widget.

Comportamento do programa

Quando você rodar este programa, deverá ver que o `Container1` agora contém, junto com o antigo botão verde, mais três botões. Veja como `myContainer1` (o frame) se estica para acomodar todos estes botões.

Perceba também que esses botões são empilhados um sobre o outro. No próximo programa, veremos por que eles se arranjam dessa forma, e veremos como arranjá-los de forma diferente.

```
from Tkinter import *

class MyApp:

    def __init__(self, parent):
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"] = "Hello, World!"      ### (1)
        self.button1["background"] = "green"      ### (1)
        self.button1.pack()

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Off to join the circus!") ### (2)
        self.button2.configure(background="tan")    ### (2)
        self.button2.pack()

        self.button3 = Button(self.myContainer1)
        self.button3.configure(text="Join me?", background="cyan") # (3)
        self.button3.pack()

        self.button4 = Button(self.myContainer1, text="Goodbye!",
                               background="red"
                               ) ### (4)
        self.button4.pack()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT050 – Packing.

No último programa, nós vimos quatro botões, empilhados um sobre o outro. Entretanto, provavelmente gostaríamos de vê-los lado a lado em alguma ocasião. Neste programa fazemos isso, e começaremos a ver as possibilidades que `pack()` oferece.

(1) (2) (3) (4)

Fazer `pack` é um das maneiras de controlar a relação visual entre os componentes da GUI. O que faremos agora é usar a opção “side” como argumento de `pack` para colocar os botões lado a lado, desse jeito:

```
self.button1.pack(side=LEFT)
```

Veja que `LEFT` (assim como `RIGHT`, `TOP` e `BOTTOM`) são constantes de nomes bem amigáveis definidas em Tkinter. Isto é, “LEFT” deveria aparecer no código como “Tkinter.LEFT”, mas por causa da nossa maneira de importar o módulo Tkinter (página 6), não precisamos mais usar o prefixo “Tkinter.”.

Porque os botões apareceram verticalmente no último programa

Como você se lembra, no último programa nós simplesmente empacotamos os botões sem especificar a opção “side”, e os botões ficaram daquela forma, empilhados. Isso aconteceu porque a opção pré-designada da opção “side” é `TOP`.

Então, quando fizemos `pack` no botão `button1`, ele foi colocado no topo da cavidade do frame `myContainer1`. Ao fazer `pack` sem argumentos sobre o botão `button2`, ele também é colocado no topo da cavidade deste frame, que neste caso fica exatamente abaixo do botão `button1`, e assim por diante.

Se nós tivéssemos feito `pack` nos botões em ordem diferente – por exemplo, se tivéssemos feito `pack` em `button2` primeiro, e depois em `button1` – suas posições teriam sido invertidas: o botão `button2` estaria em cima.

Então, como você pode ver, uma das maneiras de controlar como sua GUI vai se parecer é controlando a ordem de fazer `pack` em cada um dos widgets dentro do container.

Alguns termos técnicos – “Orientação”

Orientação *vertical* inclui os lados `TOP` (de cima) e `BOTTOM` (de baixo).

Orientação *horizontal* inclui os lados `LEFT` (esquerdo) e `RIGHT` (direito).

Quando você está empacotando widgets e containeres, é possível mesclar dois tipos de orientação. Por exemplo, podemos precisar posicionar um botão com orientação vertical (como `TOP`) e outro com orientação horizontal (como `LEFT`), mas fazer isso misturando orientações dentro do container não é uma boa idéia. Se você misturar orientações poderá prejudicar a maneira como os objetos aparecerão na tela, além da bela surpresa que terá se precisar redimensionar a janela depois.

Por estes motivos é uma boa prática de projeto nunca misturar orientações, mas sem você precisar mesmo fazer isso, é melhor usar um container dentro do outro. Exploraremos este tópico no próximo programa.

Comportamento do programa

Quando você rodar este programa, verá quatro botões lado a lado.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1["text"] = "Hello, World!"
        self.button1["background"] = "green"
        self.button1.pack(side=LEFT)   ### (1)

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Off to join the circus!")
        self.button2.configure(background="tan")
        self.button2.pack(side=LEFT)   ### (2)

        self.button3 = Button(self.myContainer1)
        self.button3.configure(text="Join me?", background="cyan")
        self.button3.pack(side=LEFT)   ### (3)

        self.button4 = Button(self.myContainer1, text="Goodbye!",
                               background="red"
                               )
        self.button4.pack(side=LEFT)   ### (4)

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT060 – Binding.

Agora é hora de colocar nossos botões para trabalhar. Chamamos sua atenção para as duas últimas (das quatro) questões básicas da programação GUI – escrever procedimentos de alimentação de eventos para fazer os trabalhos necessários no seu programa, e ligar estes procedimentos a widgets e eventos.

Perceba que neste programa rejeitamos todos os botões que tínhamos criado no último programa, e retornamos a uma GUI muito simples contendo dois botões: “OK” e “CANCEL”.

Você deve se lembrar da discussão de nosso primeiro programa, de que uma das questões básicas da programação GUI é o “binding”. Binding é o processo de definir a conexão ou relação (geralmente) entre:

- um widget;
- um tipo de evento e
- um alimentador de eventos.

Um alimentador de eventos é um método ou sub-rotina que alimenta eventos quando eles ocorrem. Talvez ajude dizer que em Java, alimentadores de eventos são chamados “listeners” (vigilantes), um nome bem sugestivo considerando sua função – estar atento aos eventos e responder a eles.

Em Tkinter, uma maneira de criar bindings é pelo método `bind()`, da seguinte forma:

```
widget.bind(nome_do_tipo_de_evento,nome_do_alimentador_de_eventos)
```

Esse tipo de ligação é chamado “event binding”.

Há outro jeito de ligar um alimentador de eventos a um widget, chamado “command binding” e nós a veremos em alguns programas daqui pra frente. Por ora, vejamos melhor o event binding, e tendo entendido este, será moleza explicar o command binding.

Antes de começarmos, vamos esclarecer uma coisa: a palavra “botão” pode ser usada para designar duas coisas inteiramente diferentes:

- um widget, `Button` – uma GUI que é mostrada no monitor do computador e
- um botão no seu mouse – aquele que você pressiona com o dedo.

Para evitar confusão, procuraremos distingui-los como “o botão” e “o botão do mouse”, em vez de simplesmente “botão”.

(1)

Nós ligamos o evento `<Button-1>` (um clique com o botão esquerdo do mouse) sobre o botão `button1` ao método `self.button1Click`. Quando `button1` é clicado com o botão esquerdo do mouse, o método `self.button1Click()` é chamado para alimentar o evento.

(3)

Veja que, embora não tenhamos especificado na operação de binding, `self.button1Click()` receberá dois argumentos. O primeiro, é claro, será “self”, primeiro argumento para qualquer método de classes em Python. O segundo será um objeto evento. Esta técnica de binding e evento – isto é, usando o método `bind()` – sempre implica na utilização de um objeto evento como argumento.

Em Python/Tkinter, quando um evento ocorre, toma forma de um objeto evento. Um objeto evento é extremamente útil porque carrega consigo uma coleção de informações úteis e

métodos. Você pode examinar o objeto evento para encontrar que tipo de evento ocorreu, o widget onde ocorreu, e outras informações úteis.

(4)

Então, o que acontece quando `button1` é clicado? Bem, neste caso nós o designamos para fazer algo bem simples: mudar sua cor de verde para amarelo e vice-versa.

(2)

Vamos fazer o `button2` (o botão do “Tchau!”) fazer algo mais útil: fechar a janela. Para isso ligamos o evento de clicar com botão esquerdo do mouse sobre o `button2` ao método `button2Click()` e

(6)

Fazemos o método `button2Click()` destruir a janela raiz, a mestra de `myapp`. Isso vai ter um efeito bem devastador, porque todos os escravos também serão destruídos. Na verdade, toda a GUI será destruída.

É claro que para isso, `myapp` tem que saber quem é sua mestra (aqui, a janela raiz). Então adicionamos um código (7) ao construtor da classe para lembrá-la disso.

Comportamento do programa

Quando você roda este programa, vê dois botões. Clicar no botão “OK” o faz mudar de cor. Clicar no botão “Cancel” fecha a aplicação.

Quando a GUI é aberta, se você apertar TAB, verá que o foco do teclado vai ficar pulando entre os dois botões, embora apertar ENTER no teclado não faça nada. Isso é porque temos ligado somente cliques de mouse aos nossos botões, não eventos de teclado. Os próximos tópicos falarão deste assunto.

Finalmente, observamos que os botões têm tamanhos diferentes porque os textos que contêm têm tamanhos diferentes. Só por causa disso. Não fica muito elegante assim, por isso consertaremos esse “problema” no próximo programa.

```
from Tkinter import *

class MyApp:

    def __init__(self, parent):
        self.myParent = parent    ### (7) lembra seu mestre, a raiz
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1.configure(text="OK", background="green")
        self.button1.pack(side=LEFT)
        self.button1.bind("<Button-1>", self.button1Click)    ### (1)

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Cancel", background="red")
```

```
self.button2.pack(side=RIGHT)
self.button2.bind("<Button-1>", self.button2Click) ### (2)

def button1Click(self, event):    ### (3)
    if self.button1["background"] == "green": ### (4)
        self.button1["background"] = "yellow"
    else:
        self.button1["background"] = "green"

def button2Click(self, event):    ### (5)
    self.myParent.destroy()      ### (6)

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT070 – Mexendo com foco e ligando eventos de teclado a widgets.

No programa anterior, você pôde fazer botões fazerem alguma coisa clicando neles com o mouse, mas não pôde fazê-los trabalhar pressionando teclas no teclado. Neste programa, veremos como fazê-los reagir a eventos de teclado como reagiram aos de mouse.

Primeiramente, precisamos definir o que vem a ser “foco”.

Se você tem alguma intimidade com a mitologia grega (ou se você viu “Hércules”, aquele filme da Disney), se lembrará das Fates. As Fates eram três velhas que controlavam o destino dos homens. Cada humano vivo era uma linha em suas mãos, e quando uma delas cortava a linha, a vida do humano terminava.



Um fato notável sobre as Fates era que havia somente um olho para as três. A que estava com o olho tinha que dizer para as outras duas tudo o que estava vendo. O olho poderia

passar de uma Fate a outra, assim elas podiam se revezar entre ver ou não. É claro, se você conseguisse roubar esse olho, teria algo valiosíssimo para barganhar o que quisesse com elas.

“Foco” é o que permite aos widgets da sua GUI ver os eventos de teclado. O foco está para os widgets da sua GUI como o olho estava para as Fates. Somente um widget por vez pode ter o foco, e o widget que o tem é o que vê, e responde a, os eventos de teclado.

Neste programa, por exemplo, nossa GUI tem dois botões, “OK” e “Cancel”. Supondo que eu pressione o botão ENTER no teclado, será que ele será visto pelo botão “OK” indicando que o usuário está confirmando sua escolha? Ou será que o ato de pressionar esta tecla será visto pelo botão “Cancel”, indicando que o usuário está cancelando (e destruindo) a aplicação? Tudo isso depende de onde está o foco. Isto é, depende de que botão “tem foco”.

Como as Fates, que passavam seu olho de uma a outra, o foco pode ser passado de um widget na GUI para outro. Há várias formas de fazer isso. Uma delas é clicando no widget com o botão esquerdo do mouse (esse modo funciona em Windows e Macintosh, em Tk e Tkinter. Há alguns sistemas que usam a convenção “foco segue o mouse”, em que o widget que está sob o mouse tem foco automaticamente – não é necessário clicar)

Outra maneira de mudar o foco é usando o fato de que os widgets são adicionados a uma lista conforme são criados. Pressionando a tecla TAB, o foco se move do widget atual para o próximo da lista. O widget seguinte ao último é o primeiro. Pressionando SHIFT+TAB, o foco se move para o item anterior da lista.

Quando um botão GUI tem foco, isso é indicado por uma caixinha de pontos em torno do texto do botão. Para ver isso, rode o programa anterior. Quando o programa começa, nenhum dos botões tem foco, por isso nenhum deles tem a caixa de pontos. Pressione TAB e você verá essa caixa em torno do texto do botão esquerdo, mostrando que o foco está sobre ele. Agora pressione várias vezes TAB e observe o foco pular de um botão a outro repetidamente.

(0)

Neste programa, queremos que o botão OK tenha foco desde o começo. Para isso usamos o método `focus_force()`, que força o foco a começar no botão OK. Quando você roda o programa, verá a caixa de pontos sobre o botão OK desde que a aplicação se abrir.

No último programa, nossos botões respondiam ao evento de teclado “pressionar a tecla TAB”, que movia o foco entre os botões, mas ao pressionar ENTER, nada acontecia. Isso porque só foram ligados cliques de mouse, não eventos de teclado, aos botões.

Neste programa nós iremos ligar também alguns eventos de teclado aos botões.

(1)(2)

As linhas para ligar eventos de teclado aos botões são bem simples – elas têm o mesmo formato que aquelas que ligam eventos de mouse. A única diferença é que o nome do evento agora é o nome de um evento de teclado (neste caso, <Return²>).

Quando queremos que a tecla ENTER ou um clique no botão esquerdo do mouse tenham o mesmo efeito no widget, ligamos o mesmo alimentador de eventos a ambos eventos.

Este programa mostra que você pode ligar vários tipos de eventos a um único widget (como um botão), ou mesmo ligar várias combinações de widgets a um mesmo alimentador de eventos.

(3)(4)

Agora que nossos botões respondem a vários tipos de eventos, podemos demonstrar como receber informações de um objeto evento. O que vamos fazer é passar o objeto evento para (5) a rotina `report_event` que irá (6) imprimir uma informação sobre o evento, obtida a partir dos atributos do evento.

Perceba que para vermos estas informações impressas na tela, é necessário rodar o programa usando python (nunca pythonw – como IDLE).

Comportamento do programa

Quando você roda o programa, vê dois botões. Clicando no botão da esquerda, ou pressionando ENTER quando o foco estiver neste botão, a sua cor será mudada. Clicando no botão direito, ou pressionando ENTER quando o foco estiver neste botão, a aplicação será fechada. Para qualquer um desses eventos de mouse ou teclado, você deverá ver uma mensagem impressa na tela informando a hora e descrevendo o evento.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()          ### (0)
        self.button1.bind("<Return>", self.button1Click)
        self.button1.bind("<Return>", self.button1Click)  ### (1)

        self.button2 = Button(self.myContainer1)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)
        self.button2.bind("<Return>", self.button2Click)
```

² Os americanos chamam a tecla ENTER de RETURN, daí o evento de teclado “pressionar ENTER” receber a sintaxe <Return>. (Nota do Tradutor)

```
self.button2.bind("<Return>", self.button2Click)   ### (2)

def button1Click(self, event):
    report_event(event)           ### (3)
    if self.button1["background"] == "green":
        self.button1["background"] = "yellow"
    else:
        self.button1["background"] = "green"

def button2Click(self, event):
    report_event(event)           ### (4)
    self.myParent.destroy()

def report_event(event):          ### (5)
    """Imprime a descrição de um evento, baseado em seus atributos.
    """
    event_name = {"2": "KeyPress", "4": "ButtonPress"}
    print "Time:", str(event.time)   ### (6)
    print "EventType=" + str(event.type), \
          event_name[str(event.type)], \
          "EventWidgetId=" + str(event.widget), \
          "EventKeySymbol=" + str(event.keysym)

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT074 – Command Binding.

Há alguns programas atrás, introduzimos o conceito de “event binding”. Há outro jeito de ligar um alimentador de eventos a um widget chamado “command binding”. Falaremos sobre isso neste programa.

Command Binding

Você deve se lembrar que nos nossos programas anteriores, nós ligamos o evento de mouse <Button-1> ao widget botão. “Button” é outro nome para um evento de mouse “ButtonPress”, que é diferente do evento de mouse “ButtonRelease”³. O evento “ButtonPress” é o ato de pressionar o botão do mouse, mas sem soltá-lo, e “ButtonRelease” é o ato de soltar o botão, deixando-o retornar à sua posição inicial.

Temos que distinguir ButtonPress de ButtonRelease para podermos explicar manipulações como “arrastar e soltar”, nas quais fazemos um ButtonPress (pressionamos o botão do mouse sem soltar) sobre um widget, o arrastamos para algum lugar, e então o soltamos (ButtonRelease).

Os widgets Button, no entanto, não podem ser “arrastados e soltos”. Se um usuário quiser arrastar e soltar um botão, ele precisará executar um ButtonPress sobre o botão e arrastá-lo com o ponteiro do mouse para um lugar qualquer na tela, e então soltar o botão. Isso não é

³ Release = Liberar, soltar (N do T).

um tipo de atividade que nós chamamos de “pressionar” (ou – em termos técnicos – “invocar”) um widget `Button`. Só é considerada uma “invocação” do botão quando o usuário faz um `ButtonPress` no widget, sem arrastá-lo, e executa imediatamente `ButtonRelease`.

Esta é uma noção mais complicada de invocação de botões do que nós temos usado até agora, onde simplesmente ligamos o evento “`Button-1`” ao widget botão usando `event binding`.

Por sorte, há outra forma de fazer `binding` que suporta essa noção mais complicada de invocação de widgets. É o “`command binding`”.

Neste programa, veja as linhas com comentários (1) e (2) para ver como o `command binding` é feito. Nestas linhas, usamos a opção “`command`” para ligar o `button1` ao alimentador de eventos `self.button1Click`, e para ligar o `button2` ao alimentador de eventos `self.button2Click`.

(3)(4)

Dê uma olhada em como os alimentadores de eventos se definem. Veja que – diferente dos alimentadores dos programas anteriores – eles não esperam por um objeto evento como argumento. Isso porque o `command binding`, diferente do `event binding`, não faz automaticamente a passagem do objeto evento como um argumento. Isso faz sentido porque o `command binding` não liga um único evento a um alimentador, e sim múltiplos eventos. Para um botão, por exemplo, ele liga o `ButtonPress` seguido de um `ButtonRelease` a um alimentador.

Daremos uma olhada um pouco melhor nas diferenças entre `event binding` e `command binding` no próximo programa, mas por ora, rodemos o programa.

Comportamento do programa

Quando você roda o programa, o botão que aparece é exatamente igual ao dos programas anteriores... só na aparência.

Para comparar seu comportamento com os botões anteriores, leve o ponteiro do mouse até o botão esquerdo e clique, *mas não solte o botão do mouse*.

Se fosse feito isso no programa anterior, imediatamente o botão mudaria de cor e a mensagem seria impressa. Neste, nada acontece... até que o botão do mouse seja solto. Aí sim, o botão muda de cor e a mensagem é impressa.

Há também outra diferença. Compare seu comportamento quando você pressiona a barra de espaço e `ENTER`. Por exemplo, use `TAB` para focar o botão `OK` e pressione espaço ou `ENTER`.

No programa anterior (onde ligamos o botão `OK` ao evento de pressionar `ENTER`), pressionar a barra de espaço não tinha efeito, mas pressionar `ENTER` fazia o botão mudar

de cor. Neste programa, por outro lado, o comportamento é exatamente o oposto – pressionando a barra de espaços faz o botão mudar de cor, enquanto ENTER não tem efeito algum.

Vamos verificar melhor estes comportamentos no nosso próximo programa.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()
        self.button1 = Button(self.myContainer1,
                               command=self.button1Click
                               ) ### (1)
        self.button1.configure(text="OK", background= "green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()
        self.button2 = Button(self.myContainer1,
                               command=self.button2Click
                               ) ### (2)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)
    def button1Click(self): ### (3)
        print "button1Click event handler"
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"
    def button2Click(self): ### (4)
        print "button2Click event handler"
        self.myParent.destroy()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT075 – Usando event binding e command binding juntos.

No programa anterior, introduzimos o conceito de command binding e delimitamos algumas das diferenças com event binding. Neste, exploraremos suas diferenças um pouco mais detalhadamente.

Para quais eventos serve command bind?

No programa anterior, se você usar TAB para focar o botão OK e pressionar a barra de espaço, o botão mudará de cor, mas pressionar ENTER não tem efeito algum.

A razão para isto é que a opção “command” dá a um widget Button distinção sobre eventos de teclado, assim como eventos de mouse. Neste caso, o evento de teclado aguardado pelo botão é o acionamento da barra de espaço, não da tecla ENTER. Isso quer dizer que, com command binding, pressionar a barra de espaços faz o botão OK mudar de cor, enquanto ENTER não tem efeito.

Este comportamento parece (ao menos para mim, usuário Windows) incomum. Parte da moral da história aqui é que se você está fazendo uso de command binding, é uma boa idéia entender exatamente o que você quer que seu binding faça. Isto é, é uma boa idéia entender exatamente que eventos de mouse e/ou teclado deverão causar os comandos desejados.

Infelizmente, a única fonte confiável dessa informação é o código Tk em si. Para informações mais acessíveis, dê uma olhada nos livros sobre Tk (“Practical Programming in Tcl and Tk”, de Brent Welch, é especialmente bom) ou sobre Tkinter. A documentação sobre Tk é difusa, mas disponível on-line.

Você deve também saber que nem todos os widgets oferecem a opção “command”. Muitos dos widgets Button o fazem (RadioButton, CheckButton, etc.) e outros oferecem opções similares (por exemplo, scrollcommand), mas você realmente deve investigar cada um dos diferentes tipos de widgets para saber se suportam ou não command binding. Isso dará um controle maior sobre o comportamento da sua GUI e tornará mais fácil sua vida de programador.

Usando Event Binding e Command Binding juntos.

Percebemos no último programa que command binding, diferente de event binding, não faz automaticamente a passagem evento objeto – argumento. Isso pode tornar sua vida um pouco complicada se você deseja ligar um alimentador de eventos a um widget usando simultaneamente event binding e command binding.

Por exemplo, neste programa realmente desejaremos que nossos botões respondam tão bem à ativação da tecla ENTER quando da barra de espaços. Para isso, teremos que usar event binding para o evento de teclado <Return>, como foi feito no último programa (1).

O problema é que o command binding não transformará um objeto evento num argumento, mas o event binding sim. Então como deveremos escrever nosso alimentador de eventos?

Há algumas soluções para este problema, mas a mais simples é escrever *dois* alimentadores de eventos. O alimentador “real” (2) será o único usado pelo command binding, que não esperará por um objeto evento.

O outro alimentador (3) vai ser só um wrapper do real. Este wrapper esperará o argumento do objeto evento, mas o ignorará e chamará o alimentador real. Daremos ao wrapper o mesmo nome do alimentador real, porém adicionando o sufixo “_a”.

Comportamento do programa

Se você rodar este programa, o comportamento será o mesmo do programa anterior, exceto pelo fato de que agora os botões irão responder tanto ao ENTER quando à barra de espaços.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.button1 = Button(self.myContainer1,
                               command=self.button1Click
                               )
        self.button1.bind("<Return>", self.button1Click_a) ### (1)
        self.button1.configure(text="OK", background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force()
        self.button2 = Button(self.myContainer1,
                               command=self.button2Click
                               )
        self.button2.bind("<Return>", self.button2Click_a) ### (1)
        self.button2.configure(text="Cancel", background="red")
        self.button2.pack(side=RIGHT)

    def button1Click(self): ### (2)
        print "button1Click event handler"
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self): ### (2)
        print "button2Click event handler"
        self.myParent.destroy()

    def button1Click_a(self, event): ### (3)
        print "button1Click_a event handler (a wrapper)"
        self.button1Click()

    def button2Click_a(self, event): ### (3)
        print "button2Click_a event handler (a wrapper)"
        self.button2Click()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT076 – Compartilhando informações entre alimentadores de eventos.

Nos últimos programas, exploramos maneiras de fazer nossos programas realmente trabalhar com alimentadores de eventos. Agora daremos uma olhadinha em como compartilhar informações entre estes alimentadores.

Compartilhando informações entre funções alimentadoras de eventos.

Há uma variedade de situações nas quais você pode querer que um alimentador de eventos realize alguma tarefa e divida os resultados desta tarefa com outros alimentadores em seu programa.

Um exemplo comum é quando você tem uma aplicação com dois grupos de widgets. Um deles seleciona alguma parte da informação, e então o outro faz alguma coisa com esta informação.

Por exemplo, talvez você queira ter um widget que permita a um usuário escolher um nome de arquivo de uma lista, e outro conjunto de widgets que ofereça várias operações sobre o arquivo escolhido – abrir, deletar, copiar, renomear, etc.

Ou você pode ter uma série de widgets que modifique várias configurações da sua aplicação, e outra série de widgets (oferecendo opções de SALVAR e CANCELAR, talvez) que salve em disco estas modificações de configuração ou as cancele sem salvar.

Quem sabe ainda um conjunto de widgets que mudem alguns parâmetros de um programa que você deseje rodar, enquanto outro widget (provavelmente um botão com um nome RODAR ou EXECUTAR) que rode o programa considerando os parâmetros que você escolheu.

Você pode precisar também requerer de uma função alimentadora de eventos o reconhecimento de alguma informação de uma execução anterior da mesma função. Considere um alimentador que simplesmente alterna uma variável entre dois diferentes valores. Para que ela assimile um novo valor da variável, terá que saber qual valor atribuiu à variável da última vez que rodou.

O problema

O problema aqui é que cada um dos alimentadores é uma função separada. Cada um deles tem suas próprias variáveis locais que não fazem parte das outras funções alimentadoras, nem mesmo de invocações anteriores delas mesmas. Eis o problema: como pode uma função alimentadora de eventos partilhar dados com outros alimentadores, se ela não pode partilhar suas variáveis locais com eles?

A solução, é claro, é que as variáveis que precisamos compartilhar não sejam locais às funções alimentadoras de eventos. Elas precisam ser armazenadas “fora” destas funções.

Primeira solução – usar variáveis globais

Uma técnica para conseguir isto é fazer delas (as variáveis que queremos compartilhar) variáveis globais. Por exemplo, em cada alimentador que precise modificar ou ver `minhaVariavel1` e `minhaVariavel2`, você pode escrever o seguinte:

```
global minhaVariavel1, minhaVariavel2
```

Cuidado: o uso de variáveis globais é potencialmente perigoso por causa de conflitos, e geralmente reservado para programas pequenos.

Segunda solução – usar variáveis instanciadas

Uma boa técnica é usar variáveis instanciadas (isto é, “self.nome_da_variavel”) para trocar informações que você quer compartilhar entre os alimentadores de eventos. Para fazer isso, é claro, sua aplicação deverá ser implementada em uma classe, e não num simples código estruturado.

Este é uma das razões pelas quais nós desenvolvemos as aplicações deste tutorial em forma de classes. É por termos começado neste formato logo de início que neste ponto nossa aplicação já temos uma infra-estrutura que nos permitirá usar variáveis instanciadas.

Neste programa, iremos compartilhar uma informação bem simples: o nome do último botão pressionado. Armazenaremos esta informação numa variável instanciada chamada “self.myLastButtonInvoked” (veja o comentário ### 1).

Para mostrar que realmente estamos lembrando desta informação, toda vez que o alimentador do botão for solicitado, ela será impressa (veja o comentário ### 2).

Comportamento do programa

Este programa mostra três botões. Quando você colocá-lo para rodar, se você clicar em qualquer um dos botões, será mostrado seu próprio nome, e o nome do botão que foi clicado anteriormente.

Perceba que nenhum dos botões fechará a aplicação, então se você desejar fechá-la, deverá clicar no widget FECHAR (o ícone com um “X” em uma caixa, do lado direito da barra de título).

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        ### 1 Ainda não solicitamos o alimentador do botão

        self.myLastButtonInvoked = None

        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        self.yellowButton=Button(self.myContainer1,
                                command=self.yellowButtonClick
                                )
        self.yellowButton.configure(text="YELLOW",
                                    background="yellow"
                                    )
        self.yellowButton.pack(side=LEFT)
```

```
self.redButton=Button(self.myContainer1,
                       command=self.redButtonClick
                       )
self.redButton.configure(text="RED", background="red")
self.redButton.pack(side=LEFT)

self.whiteButton=Button(self.myContainer1,
                        command=self.whiteButtonClick
                        )
self.whiteButton.configure(text="WHITE",background="white")

self.whiteButton.pack(side=LEFT)

def redButtonClick(self):
    print "RED button clicked. Previous button invoked was",
    self.myLastButtonInvoked ### 2
    self.myLastButtonInvoked = "RED" ### 1

def yellowButtonClick(self):
    print "YELLOW button clicked. Previous button invoked was",
    self.myLastButtonInvoked ### 2
    self.myLastButtonInvoked = "YELLOW" ### 1

def whiteButtonClick(self):
    print "WHITE button clicked. Previous button invoked was",
    self.myLastButtonInvoked ### 2
    self.myLastButtonInvoked = "WHITE" ### 1

print "\n"*100 # um jeito simples de limpar a tela
print "Starting..."
root = Tk()
myapp = MyApp(root)
root.mainloop()
print "... Done!"
```

TT077 – Transmitindo argumentos para alimentadores de eventos I O problema

Neste programa exploraremos um pouco...

Características mais avançadas de command binding

No programa tt075.py, usamos a opção “command” para ligar um alimentador de eventos a um widget. Por exemplo, neste programa a linha

```
self.button1 = Button(self.myContainer1, command=self.button1Click)
```

ligou a função button1Click ao widget button1.

Também usamos event binding para ligar nossos botões ao evento de teclado <Return>.

```
self.button1.bind("<Return>", self.button1Click_a)
```

Em um dos últimos programas, os alimentadores de eventos para os dois botões realizaram duas funções diferentes.

Mas suponha que a situação seja diferente. Suponha que temos diversos botões, todos eles desenvolvendo essencialmente o mesmo tipo de ação. A melhor maneira de alimentar este tipo de situação é ligar os eventos de todos os botões a um único alimentador de eventos. Cada um dos botões deverá invocar a mesma rotina alimentadora, mas fornecendo a ela diferentes argumentos contando o que fazer.

É isso que fazemos neste programa.

Command binding

Neste programa, como você pode ver, temos dois botões, e usamos a opção “command” para ligá-los todos ao mesmo alimentador de eventos – a rotina “buttonHandler”. Fornecemos a esta rotina três argumentos: o nome do botão (na variável `button_name`), um número e uma string.

```
self.button1=Button(self.myContainer1,
                    command=self.buttonHandler(button_name, 1,
                                                "Good stuff!"
                                                )
                    )
```

Em aplicações sérias, a rotina `buttonHandler` iria, é claro, trabalhar seriamente, mas neste programa ela meramente imprime os argumentos que recebeu.

Event binding

Chega de command binding. O que diremos sobre event binding?

Você perceberá que comentamos as duas linhas que fazem event binding no evento `<Return>`.

```
self.button1.bind("<Return>", self.buttonHandler_a(event,
                                                    button_name, 1,
                                                    "Good stuff!"
                                                    )
                 )
```

Este é o primeiro sinal de um problema. O event binding automaticamente transmite o argumento de evento – mas simplesmente não há como incluir que o argumento de evento em nossa lista de argumentos.

Teremos que voltar a este problema mais tarde. Por ora, vamos simplesmente rodar o programa e ver o que acontece.

Comportamento do programa

Vendo o código, o programa até que não parece tão mal, mas quando você o roda, percebe que ele não trabalha direito. A rotina `buttonHandler` é solicitada antes mesmo que a GUI seja exibida. Na verdade, ela é solicitada *duas* vezes!

Se você clicar com o botão esquerdo do mouse em qualquer botão, descobrirá que nada acontece – a rotina `eventHandler` não está sendo solicitada.

Veja que a única maneira de fechar o programa é clicando no ícone FECHAR (o “X” na caixa) no lado direito da barra de títulos.

Então rode o programa agora, e veja o que acontece. No próximo programa veremos porque isso acontece.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        button_name = "OK"
        self.button1=Button(self.myContainer1,
                            command=self.buttonHandler(button_name,
                                                         1,"Good stuff!"
                                                         )
                            )

        # self.button1.bind("<Return>", self.buttonHandler_a(event,
        # button_name, 1, "Good stuff!"))
        self.button1.configure(text=button_name, background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force() # Foca o teclado em button1

        button_name = "Cancel"
        self.button2=Button(self.myContainer1,
                            command=self.buttonHandler(button_name,
                                                         2,"Bad stuff!"
                                                         )
                            )

        # self.button2.bind("<Return>",self.buttonHandler_a(event,
        # button_name, 2, "Bad stuff!"))
        self.button2.configure(text=button_name, background="red")
        self.button2.pack(side=LEFT)

    def buttonHandler(self, arg1, arg2, arg3):
        print "buttonHandler routine received arguments:", arg1.ljust
(8), arg2, arg3

    def buttonHandler_a(self, event, arg1, arg2, arg3):
        print "buttonHandler_a received event", event
        self.buttonHandler(arg1, arg2, arg3)
```

```
print "\n"*100 # limpa a tela
print "Starting program tt077."
root = Tk()
myapp = MyApp(root)
print "Ready to start executing the event loop."
root.mainloop()
print "Finished          executing the event loop."
```

TT078 – Transmitindo argumentos para alimentadores de eventos II Usando Lambda

Vendo a execução do último programa, nos perguntamos: “O que está acontecendo aqui??? A rotina `buttonHandler` está sendo executada por cada um dos botões, mesmo antes do event loop entrar em ação!”

A razão é que uma linha como

```
self.button1 = Button(self.myContainer1,
                      command = self.buttonHandler(button_name,
                                                    1, "Good stuff!"
                                                    )
                      )
```

foi chamada de *função* `buttonHandler`, embora fosse melhor tê-la chamado de *rechamada*. Embora não seja o que pretendíamos, é o que realmente acontece.

Veja que:

- `buttonHandler` é um objeto função, e pode ser usado como uma ligação de chamada.
- `buttonHandler()` (viu os parênteses?), por outro lado, é uma chamada real da função `buttonHandler`.

Quando a linha

```
self.button1 = Button(self.myContainer1,
                      command = self.buttonHandler(button_name,
                                                    1, "Good stuff!"
                                                    )
                      )
```

é executada, ela está na verdade fazendo a chamada da rotina `buttonHandler`. A rotina é então executada, imprimindo uma mensagem, e retornando o resultado da chamada (neste caso, o objeto `None`). Aí a opção “`command`” do botão é ligada ao resultado da chamada. Em resumo, “`command`” é ligada ao objeto “`None`”. É por causa disso que, quando você clica em qualquer um dos botões, nada acontece.

Há uma solução?

Então... qual a solução? Há algum meio de parametrizar, e reutilizar, uma função alimentadora de eventos?

Sim. Há uma série de técnicas reconhecidas de fazer isso. Uma delas usa a função Lambda, uma função built-in de Python, e outra usa uma técnica chamada de “currying”.

Neste programa discutiremos como trabalhar com lambda, e no próximo programa daremos uma olhada em currying.

Não vou tentar explicar como lambda e currying trabalham – isso é muito complicado e está bem longe do nosso objetivo principal, que é ter programas Tkinter rodando. Vamos então simplesmente tratá-las como caixas-pretas. Não vou dizer como elas trabalham – só como trabalhar com elas.

Vejamos lambda.

Command binding

Há pouco tempo, achávamos que a seguinte linha devesse funcionar:

```
self.button1 = Button(self.myContainer1,
                      command = self.buttonHandler(button_name,
                                                    1, "Good stuff!"
                                                    )
                      )
```

... mas já descobrimos que ela não faz o que pensamos que faria.

O jeito de fazer o que queremos é reescrever esta linha da seguinte forma:

```
self.button1 = Button(self.myContainer1,
                      command = lambda,
                      arg1=button_name, arg2=1,
                      arg3="Good Stuff!":self.buttonHandler(arg1,
                                                              arg2,
                                                              arg3)
                      )
```

Event binding

Felizmente, lambda também nos dá um jeito de parametrizar event binding. Em vez de:

```
self.button1.bind("<Return>",
                  self.buttonHandler_a(event, button_name, 1, "Good stuff!")
                  )
```

(que não funcionaria porque não é possível incluir argumentos de evento dentro da lista de argumentos), podemos usar lambda, dessa forma:


```
# event binding - transmitindo o evento como um argumento
self.button1.bind("<Return>",
    lambda
        event, arg1=button_name, arg2=1, arg3="Good stuff!" :
            self.buttonHandler_a(event, arg1, arg2, arg3)
    )
```

(Veja que “event” aqui é o nome de uma variável – não uma keyword de Python ou algo que o valha. Este exemplo usa o nome “event” para o argumento de evento; alguns autores usam o nome “e” para isso, mas tanto faz. Poderíamos ter escrito “event_arg”, se quiséssemos.)

Uma das características mais elegantes quando se usa lambda é que podemos (se quisermos), simplesmente não transmitir o argumento de evento. Se fizermos isso, poderemos então chamar a função `self.buttonHandler` diretamente, em vez de indiretamente pela função `self.buttonHandler_a`.

Para ilustrar esta técnica, iremos programar o event binding do `button2` diferente do `button1`.

Eis o que fizemos com o segundo botão:

```
# event binding - sem transmitir o evento como um argumento
self.button2.bind("<Return>",
    lambda
        event, arg1=button_name, arg2=2, arg3="Bad stuff!" :
            self.buttonHandler(arg1, arg2, arg3)
    )
```

Comportamento do programa

Rodando o programa, ele se comportará exatamente como queremos.

Veja que você pode mudar o foco de teclado entre os botões OK e CANCEL pressionando a tecla TAB.

Particularmente, tente ativar OK pressionando ENTER. Fazendo isso você estará colocando-o para trabalhar pela função `buttonHandler_a`, e também receberá uma mensagem impressa, informando sobre qual evento foi recebido por ele.

Em qualquer caso, tanto clicando em um dos botões com o mouse, ou solicitando um widget via ENTER, o programa imprimirá direitinho os argumentos que foram transmitidos pela função `buttonHandler`.

```
from Tkinter import *
class MyApp:
    def __init__(self, parent):
```

```
self.myParent = parent
self.myContainer1 = Frame(parent)
self.myContainer1.pack()

#----- BOTÃO N.º 1 -----
button_name = "OK"

# command binding
self.button1 = Button(self.myContainer1,
    command = lambda
        arg1=button_name, arg2=1, arg3="Good stuff!" :
        self.buttonHandler(arg1, arg2, arg3)
    )

# event binding - transmitindo o evento como um argumento
self.button1.bind("<Return>",
    lambda
        event, arg1=button_name, arg2=1, arg3="Good stuff!" :
        self.buttonHandler_a(event, arg1, arg2, arg3)
    )

self.button1.configure(text=button_name, background="green")
self.button1.pack(side=LEFT)
self.button1.focus_force() # põe o foco de teclado em button1

#----- BOTÃO N.º 2 -----

button_name = "Cancel"

# command binding
self.button2 = Button(self.myContainer1,
    command = lambda
        arg1=button_name, arg2=2, arg3="Bad stuff!":
        self.buttonHandler(arg1, arg2, arg3)
    )

# event binding - sem passar o evento como um argumento
self.button2.bind("<Return>",
    lambda
        event, arg1=button_name, arg2=2, arg3="Bad stuff!" :
        self.buttonHandler(arg1, arg2, arg3)
    )

self.button2.configure(text=button_name, background="red")
self.button2.pack(side=LEFT)

def buttonHandler(self, argument1, argument2, argument3):
    print "buttonHandler routine received arguments:" \
        , argument1.ljust(8), argument2, argument3

def buttonHandler_a(self, event, argument1, argument2, argument3):
    print "buttonHandler_a received event", event
    self.buttonHandler(argument1, argument2, argument3)

print "\n"*100 # limpa a tela
print "Starting program tt078."

root = Tk()
myapp = MyApp(root)
```

```
print "Ready to start executing the event loop."  
root.mainloop()  
print "Finished executing the event loop."
```

TT079 – Transmitindo argumentos para alimentadores de eventos III Usando Currying

No programa anterior, vimos uma técnica envolvendo lambda para transmitir argumentos a uma função alimentadora de eventos. Neste programa, daremos uma olhada em como fazer a mesma coisa usando uma técnica chamada “currying”.

Sobre Curry

Em seu sentido mais simples, currying é a técnica de usar função para construir outras funções.

Currying é uma técnica herdada da programação funcional. Se você quiser saber mais sobre ela, há diversas informações no “Python Cookbook”. A classe curry usada neste programa é a receita de Scott David Daniel, “Curry – associando parâmetros com funções”, disponível em

<http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/52549>

Como foi discutido sobre lambda, não é meu objetivo explicar como funciona currying: trataremos isso como uma caixa-preta.

Curry – como usá-lo

A maneira de usar curry (a técnica, não o tempero!) é incluir a classe “curry” em seu programa, ou importá-la de seu próprio arquivo Python. Neste programa, iremos incluir o código curry diretamente no programa.

Inicialmente, pensamos que a seguinte linha liga self.buttonHandler à opção command de self.button1, mas descobrimos que isso não funciona da maneira que pensávamos.

```
self.button1 = Button(self.myContainer1,  
                      command=self.buttonHandler(button_name,1,"Good stuff!")  
                      )
```

Usando curry, o jeito de fazer o que queremos é reescrever esta linha assim:

```
self.button1 = Button(self.myContainer1,  
                      command=curry(self.buttonHandler,button_name,1,"Good stuff!")  
                      )
```

Como você pode ver, o código é bem direto. Em vez de solicitar a função self.buttonHandler, criamos um objeto curry (isto é, uma instância da classe curry), transmitindo a função self.buttonHandler em seu primeiro argumento. Basicamente, o que

acontece é que o objeto `curry` recorda o nome da função dada. Por isso, quando é requisitado, o objeto `curry` chama a função que lhe foi dada quando foi criado.

Event binding

Chad Netzer desenvolveu uma técnica similar ao `currying`, que pode ser usada para parametrizar `event binding`⁴ que envolve uma função “`event_lambda`”.

Para usar `event_lambda`, como com `curry`, você deve incluir o código para a função “`event_lambda`” em seu programa, ou importá-la de seu próprio arquivo Python. Neste programa, incluímos o código da função `event_lambda` diretamente no programa.

```
# ----- código para a função event_lambda -----
def event_lambda(f, *args, **kwds ):
    """Escrevendo lambda numa interface mais amigável"""
    return lambda event, f=f, args=args, kwds=kwds: f(*args, **kwds)
```

Uma vez tendo a função `event_lambda` disponível, podemos usá-la para ligar `self.buttonHandler` ao evento de teclado `ENTER`, fornecendo a ela alguns argumentos. Eis como fazemos isso:

```
self.button1.bind("<Return>",
                  event_lambda(self.buttonHandler, button_name,
                               1, "Good stuff!"))
```

Se você tiver uma curiosidade insaciável sobre como `event_lambda` funciona, dê uma olhadinha no código do botão `button2`.

Para `button2`, usamos um processo em dois passos. Primeiro solicitamos a função `event_lambda`:

```
event_handler=event_lambda(self.buttonHandler,
                            button_name, 2, "Bad stuff!")
```

Quando ela é solicitada, usa `lambda` para criar um objeto função novo e sem nome (“anônimo”).

```
lambda event, f=f, args=args, kwds=kwds : f(*args, **kwds )
```

O objeto função sem nome é um envoltório para a função que realmente queremos solicitar (“`f`”, que neste programa é “`self.buttonHandler`”) e os argumentos nós especificamos quando chamamos a função `event_lambda`. Assim, a função `event_lambda` retorna esta nova e anônima função.

Quando `event_lambda` retorna a função anônima, damos a ela o nome de “`event_handler`”.

⁴ A utilização desta técnica exige versão de Python igual ou superior a Python 2.0

```
event_handler=event_lambda(self.buttonHandler,  
                             button_name,2,"Bad stuff!")
```

Então, no segundo passo, ligamos o evento ENTER (<Return>) à função “event_handler”:

```
self.button2.bind("<Return>", event_handler)
```

Perceba que para a função anônima, “event” é só um argumento suporte que é descartado e não chega a ser usado. Somente os argumentos posicionais (args) e os argumentos de teclado (kwargs) são transmitidos à rotina alimentadora do botão.

Ufa! Já torrei um monte de neurônios!

Este é um assunto complicado. Mas você não precisa torrar seus neurônios tentando entender como tudo isso funciona. Você não precisa saber como curry e event_lambda trabalham se seu objetivo for usá-los. Trate-os como caixas-pretas: use-os sem se preocupar com o que há dentro.

Lambda versus Curry e event_lambda – Qual devo usar?

Bem...

- O código para usar curry e event_lambda é relativamente intuitivo, curto e simples. O lado negativo é que para usá-los você deve incluí-los no código do seu programa, ou importá-los.
- Lambda, em contrapartida, é built-in em Python – você não precisa fazer nada especial para importá-la; simples assim. Há o lado negativo: o código para usá-la é grande e um pouco confuso.

Escolha o que lhe convier. Como diz o ditado, “o cliente sempre tem razão”. Use o que for cômodo para suas tarefas.

A REAL moral dessa história é que...

Python é uma linguagem poderosa, e oferece várias ferramentas que podem ser usadas para criar funções de chamada para manipulação de eventos. “Pensando em Tkinter” é uma introdução aos conceitos básicos, não uma enciclopédia de técnicas, por isso exploramos só algumas delas aqui. Mas cá entre nós, você se torna muito mais hábil com Python, e como sua necessidade por mais flexibilidade é cada vez maior, há características mais avançadas da linguagem que estarão disponíveis para você desenvolver exatamente o tipo de funções de chamada que você precisar.

Comportamento do programa

Se você rodar este programa, ele se comportará como o anterior. Não mudamos nada no comportamento do programa, somente a maneira como foi escrito.

```
from Tkinter import *

# ----- código da classe curry (início) -----
class curry:
    """da receita de Scott David Daniel
    "curry - associando parâmetros com funções"
    no "Python Cookbook"
    """

    def __init__(self, fun, *args, **kwargs):
        self.fun = fun
        self.pending = args[:]
        self.kwargs = kwargs.copy()

    def __call__(self, *args, **kwargs):
        if kwargs and self.kwargs:
            kw = self.kwargs.copy()
            kw.update(kwargs)
        else:
            kw = kwargs or self.kwargs
        return self.fun(*(self.pending + args), **kw)
# ----- código da classe curry (final) -----

# ----- código da função event_lambda (começo) -----
def event_lambda(f, *args, **kwds ):
    """Uma função auxiliary que envolve
    lambda numa interface mais amigável
    Obrigado a Chad Netzer pelo código."""
    return lambda event, f=f, args=args, kwds=kwds : f( *args, **kwds )
# ----- código da função event_lambda (final) -----

class MyApp:

    def __init__(self, parent):
        self.myParent = parent
        self.myContainer1 = Frame(parent)
        self.myContainer1.pack()

        button_name = "OK"

        # command binding - usando curry
        self.button1 = Button(self.myContainer1,
            command = curry(self.buttonHandler,
                button_name, 1, "Good stuff!")
        )

        # event binding - usando a função auxiliary event_lambda
        self.button1.bind("<Return>",
            event_lambda(self.buttonHandler,
                button_name, 1, "Good stuff!")
        )

        self.button1.configure(text=button_name, background="green")
        self.button1.pack(side=LEFT)
        self.button1.focus_force() # Põe o foco de teclado em button1

        button_name = "Cancel"
```

```
# command binding -- usando curry
self.button2 = Button(self.myContainer1,
                      command = curry(self.buttonHandler,
                                      button_name, 2,
                                      "Bad stuff!"))
)

# event binding - usando a função auxiliary event_lambda em
dois passos
event_handler=event_lambda(self.buttonHandler,
                           button_name,2, "Bad stuff!")
self.button2.bind("<Return>", event_handler )
self.button2.configure(text=button_name, background="red")
self.button2.pack(side=LEFT)

def buttonHandler(self, argument1, argument2, argument3):
    print "buttonHandler routine received arguments:", \
          argument1.ljust(8), argument2, argument3

def buttonHandler_a(self, event, argument1, argument2, argument3):
    print "buttonHandler_a received event", event
    self.buttonHandler(argument1, argument2, argument3)

print "\n"*100 # limpa a tela
print "Starting program tt079."
root = Tk()
myapp = MyApp(root)
print "Ready to start executing the event loop."
root.mainloop()
print "Finished executing the event loop."
```

TT080 – Opções de widget e configurações de pack

Nos últimos programas, perdemos um tempão discutindo técnicas para ligar alimentadores de eventos a widgets.

Com este programa, retornamos ao tópico de configurar a aparência da GUI – mudando os widgets e controlando sua aparência e posição.

Três técnicas de controlar o layout de uma GUI

Há três técnicas de controlar o layout geral de uma GUI:

- Atributos dos widgets;
- Opções de pack() e
- Posicionando os containeres (como frames).

Neste programa, veremos como controlar a aparência mudando os atributos dos widgets e as opções de pack().

Trabalhemos um pouco com alguns botões e com o frame que os contém. Nas últimas versões deste programa, chamamos o frame de “myContainer1”. Aqui, iremos renomeá-lo como algo mais descritivo: “buttons_frame”.

Os números das seções seguintes se referem aos comentários numerados no código fonte.

(1)

Primeiro, para nos certificarmos de que todos os botões têm a mesma largura, especificamos um atributo “largura” (width) que é o mesmo para todos eles. Perceba que o atributo width é específico para o widget Button de Tkinter – nem todos os widgets possuem atributos de largura. Perceba também que o atributo width é especificado em unidades de caracteres (e não, por exemplo, em unidades de pixels, polegadas, milímetros). Sabendo que nosso maior rótulo (“Cancel”) tem seis caracteres, vamos definir a largura dos botões como “6”.

(2)

Agora adicionamos padding aos nossos botões. Padding são espaços extra em torno do texto, entre o texto e a borda do botão. Fazemos isso mudando os atributos “padx” e “pady” dos botões. “padx” cria os espaços extras ao longo do eixo X, horizontalmente, à direita e à esquerda do texto, enquanto “pady” faz isso ao longo do eixo Y, verticalmente, acima e abaixo do texto.

Vamos especificar nosso espaço horizontal como 3 milímetros (padx=”3m”) e nosso espaço vertical como 1 milímetro (pady=”1m”). Veja que, diferente do atributo width, que é numérico, estes atributos são escritos entre aspas. Isso porque estamos especificando as unidades do padding por meio do sufixo “m”, então temos que especificar o tamanho dos espaços como uma string em vez de números.

(3)

Finalmente, adicionamos alguns espaços ao container (buttons_frame) que organiza os botões. Para o container, especificamos quatro atributos de padding. “padx” e “pady” especificam o espaço que deve existir em torno (do lado de fora) do frame. “ipadx” e “ipady” (“padx interno” e “pady interno”) especificam os espaços internos. Este é o espaçamento em torno de cada um dos widgets dentro do container.

Obs: não especificamos o espaçamento do frame como um atributo do frame, e sim como uma opção que fornecemos ao gerenciador de geometria (neste caso, pack).

(4)

Como você pode ver, o espaçamento é um pouco confuso. Frames têm espaçamentos internos, mas widgets como button não. Em alguns casos, o espaçamento é um atributo do widget, enquanto em outros casos temos que especificá-lo com opções de pack().

Comportamento do programa

Quando você rodar este programa, verá dois botões, mas agora eles devem ter o mesmo tamanho. O tamanho dos botões é tal que o texto não fica tão espremido com antes: agora os botões têm uma borda considerável.


```
from Tkinter import *

class MyApp:

    def __init__(self, parent):

        # ----- constantes para controle do layout -----
        button_width = 6      ### (1)

        button_padx = "2m"   ### (2)
        button_pady = "1m"   ### (2)

        buttons_frame_padx = "3m"   ### (3)
        buttons_frame_pady = "2m"   ### (3)
        buttons_frame_ipadx = "3m"  ### (3)
        buttons_frame_ipady = "1m"  ### (3)
        # ----- fim das constantes -----

        self.myParent = parent
        self.buttons_frame = Frame(parent)

        self.buttons_frame.pack(      ### (4)
            ipadx=buttons_frame_ipadx, ### (3)
            ipady=buttons_frame_ipady, ### (3)
            padx=buttons_frame_padx,   ### (3)
            pady=buttons_frame_pady,   ### (3)
        )

        self.button1=Button(self.buttons_frame,
            command=self.button1Click)
        self.button1.configure(text="OK", background= "green")
        self.button1.focus_force()
        self.button1.configure(width=button_width,   ### (1)
            padx=button_padx,   ### (2)
            pady=button_pady   ### (2)
        )
        self.button1.pack(side=LEFT)
        self.button1.bind("<Return>", self.button1Click_a)

        self.button2=Button(self.buttons_frame,
            command=self.button2Click)
        self.button2.configure(text="Cancel", background="red")
        self.button2.configure(width=button_width,   ### (1)
            padx=button_padx,   ### (2)
            pady=button_pady   ### (2)
        )

        self.button2.pack(side=RIGHT)
        self.button2.bind("<Return>", self.button2Click_a)

    def button1Click(self):
        if self.button1["background"] == "green":
            self.button1["background"] = "yellow"
        else:
            self.button1["background"] = "green"

    def button2Click(self):
        self.myParent.destroy()

    def button1Click_a(self, event):
        self.button1Click()
```

```
def button2Click_a(self, event):
    self.button2Click()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT090 – Posicionando frames.

Neste programa, daremos uma olhada em posicionamento de containeres (frames). O que iremos fazer é criar uma série de frames, colocados um dentro do outro: `bottom_frame`, `left_frame` e `big_frame`.

Estes frames não conterão nada – nenhum widget. Normalmente, por causa da elasticidade dos frames, eles se encolheriam até não serem visíveis, mas especificando atributos de altura e largura podemos fornecer um tamanho inicial a eles.

Veja que não especificamos altura e largura para todos os frames. Para `myContainer1`, por exemplo, não especificamos nenhum dos dois atributos, mas especificando estes atributos para os widgets que este frame continha, ele se esticou até acomodar as alturas e larguras acumuladas pelas alturas e larguras desses widgets.

Posteriormente exploraremos como colocar widgets nestes frames; por enquanto simplesmente criaremos os frames, e daremos a eles diferentes tamanhos, posições e cores de fundo.

Colocaremos também bordas em torno dos três frames que nos interessarão mais no futuro: `bottom_frame`, `left_frame` e `right_frame`. Os outros frames (`top_frame` e `buttons_frame`) não receberão bordas.

Comportamento do programa

Quando você roda este programa, vê frames diferentes, com diferentes cores de fundo.

```
from Tkinter import *

class MyApp:
    def __init__(self, parent):

        #----- constantes para controlar o layout dos botões -----
        button_width = 6
        button_padx = "2m"
        button_pady = "1m"
        buttons_frame_padx = "3m"
        buttons_frame_pady = "2m"
        buttons_frame_ipadx = "3m"
        buttons_frame_ipady = "1m"
        # ----- fim das constantes -----
```

```
self.myParent = parent

### Nosso frame mais importante chama-se myContainer1
self.myContainer1 = Frame(parent)
self.myContainer1.pack()

### Usamos orientação VERTICAL (top/bottom) dentro
### de myContainer1. Dentro de myContainer1, primeiro
### criamos o frame buttons_frame
### Então criamos top_frame e bottom_frame
### Estes serão nossos frames-exemplo.

# buttons_frame
self.buttons_frame = Frame(self.myContainer1)
self.buttons_frame.pack(
    side=TOP,
    ipadx=buttons_frame_ipadx,
    ipady=buttons_frame_ipady,
    padx=buttons_frame_padx,
    pady=buttons_frame_pady,
)

# top_frame
self.top_frame = Frame(self.myContainer1)
self.top_frame.pack(side=TOP,
    fill=BOTH,
    expand=YES,
)

# bottom_frame
self.bottom_frame = Frame(self.myContainer1,
    borderwidth=5, relief=RIDGE,
    height=50,
    background="white",
)
self.bottom_frame.pack(side=TOP,
    fill=BOTH,
    expand=YES,
)

### Coloquemos agora mais dois frames, left_frame
### e right_frame,
### dentro de top_frame, usando orientação
### HORIZONTAL (left/right)

# left_frame
self.left_frame = Frame(self.top_frame, background="red",
    borderwidth=5, relief=RIDGE,
    height=250,
    width=50,
)
self.left_frame.pack(side=LEFT,
    fill=BOTH,
    expand=YES,
)
```

```
### right_frame
self.right_frame = Frame(self.top_frame, background="tan",
                          borderwidth=5, relief=RIDGE,
                          width=250)
self.right_frame.pack(side=RIGHT, fill=BOTH, expand=YES)

# agora adicionamos os botões a buttons_frame
#----- constantes para controle do layout -----
button_width = 6      ### (1)

button_padx = "2m"    ### (2)
button_pady = "1m"    ### (2)

buttons_frame_padx = "3m"  ### (3)
buttons_frame_pady = "2m"  ### (3)
buttons_frame_ipadx = "3m"  ### (3)
buttons_frame_ipady = "1m"  ### (3)
# ----- fim das constantes -----

self.button1=Button(self.buttons_frame,
                    command=self.button1Click)
self.button1.configure(text="OK", background= "green")
self.button1.focus_force()
self.button1.configure(
    width=button_width,
    padx=button_padx,
    pady=button_pady
)

self.button1.pack(side=LEFT)
self.button1.bind("<Return>", self.button1Click_a)

self.button2=Button(self.buttons_frame,
                    command=self.button2Click)
self.button2.configure(text="Cancel", background="red")
self.button2.configure(width=button_width,
                        padx=button_padx,
                        pady=button_pady
)

self.button2.pack(side=RIGHT)
self.button2.bind("<Return>", self.button2Click_a)

def button1Click(self):
    if self.button1["background"] == "green":
        self.button1["background"] = "yellow"
    else:
        self.button1["background"] = "green"

def button2Click(self):
    self.myParent.destroy()

def button1Click_a(self, event):
    self.button1Click()

def button2Click_a(self, event):
    self.button2Click()

root = Tk()
myapp = MyApp(root)
root.mainloop()
```

TT095 – Métodos gerenciadores de janelas & controlando o tamanho de janelas com a opção *geometry*.

Dimensionar janelas pode ser uma experiência frustrante quando se trabalha com Tkinter. Imagine esta situação: você acredita em desenvolvimento interativo, então primeiro você cuidadosamente cria um frame com a especificação de altura e largura que deseja e depois de uns testes, percebe que funcionou. Então você vai para o próximo passo, que é de adicionar alguns botões ao frame. Testa de novo, mas agora, para sua surpresa, o Tkinter está agindo como se não houvesse especificação alguma quando à altura e largura do frame: o frame se encolheu ao mínimo suficiente para acomodar os botões.

O que está acontecendo???

Bem, o comportamento do gerenciador de geometria (pack, em nossos programas) é consistente. Ou, permita-me dizer: o comportamento de pack depende da situação. O ponto claro é que o pack honrará com a solicitação de dimensões se o container estiver vazio, mas se este contiver qualquer outro widget, então a elasticidade natural do container virá à tona – as configurações de altura e largura do container serão ignoradas em favor das configurações dos widgets e o container se ajustará a estes tão apertado quando possível.

Isso é fato: *é impossível* controlar o tamanho de um container não-vazio.

O que você pode controlar é o tamanho inicial da janela raiz inteira (aquela que é um container do container – possui barra de título, etc.), e você faz isso pela opção gerenciadora de janelas “geometry”.

(1)

Neste programa, usamos a opção geometry para fazer uma janelona em torno de um framezinho.

(2)

Veja que a opção “title”, que nós também usamos neste programa, é também um método gerenciador de janelas. “Title” controla o texto do título na barra de títulos da janela.

Veja também que a opção gerenciadora de janelas pode opcionalmente ser especificada com um prefixo “wm_”. Por exemplo, “wm_geometry” e “wm_title”. Neste programa, só para mostrar que você pode usar qualquer uma destas sintaxes, usamos “geometry” e “wm_title”.

Comportamento do programa

Este programa mostra quatro janelas em sucessão

Observe que você terá que fechar cada uma delas clicando no widget CLOSE – o famigerado “X” em uma caixa, do lado direito da barra de títulos.

Na primeira, vemos que o frame detém as propriedades de altura e largura que especificamos, mas veja: ele não contém widgets!

Na segunda, vemos que exatamente o mesmo frame se comporta diferentemente quando alguns widgets (aqui, três botões) são adicionados a ele. Veja que o frame se encolheu em torno dos três botões.

Na terceira janela, novamente mostramos como o frame vazio se parece, só que dessa vez usamos a opção `geometry` para controlar o tamanho da janela inteira. Podemos ver o fundo azul do frame dentro de um grande campo cinza da janela.

Na última, novamente mostramos o frame com os três botões dentro dele, mas desta vez especificando o tamanho da janela com a opção `geometry`. O tamanho da janela é o mesmo que na janela anterior, porém (como na segunda janela) o frame se encolheu em torno dos botões, por isso não podemos ver seu fundo azul.

```
from Tkinter import *

class App:
    def __init__(self, root, use_geometry, show_buttons):
        fm = Frame(root, width=300, height=200, bg="blue")
        fm.pack(side=TOP, expand=NO, fill=NONE)

        if use_geometry:
            root.geometry("600x400")
            ### (1) observe o método gerenciador de janelas

        if show_buttons:
            Button(fm, text="Button 1", width=10).pack(side=LEFT)
            Button(fm, text="Button 2", width=10).pack(side=LEFT)
            Button(fm, text="Button 3", width=10).pack(side=LEFT)

case = 0
for use_geometry in (0, 1):
    for show_buttons in (0,1):
        case = case + 1
        root = Tk()
        root.wm_title("Case " + str(case))
        ### (2) observe o método gerenciador de janelas wm_title
        display = App(root, use_geometry, show_buttons)
        root.mainloop()
```

TT100 – Opções de pack: side, expand, fill e anchor.

Neste programa, veremos mais opções de `pack()` para controlar layouts dentro de frames:

- Side;
- Fill;
- Expand e
- Anchor.

Este programa é diferente dos outros da série. Isto é, você não terá que ler seu código para entender suas características. Só precisa *rodar* o programa.

O propósito do programa é mostrar a você os resultados das opções de pack. Rodar o programa vai permitir a você escolher diferentes opções e observar os efeitos das diferentes combinações de opções.

Os conceitos subjacentes da opção pack

Para entender como podemos controlar a aparência dos widgets dentro de um container (como um frame), precisamos lembrar que o gerenciador de geometria pack usa o modelo de arranjo baseado no conceito de “cavidade”. Isto é, cada container é uma cavidade, e pack acomoda o conteúdo dentro dela.

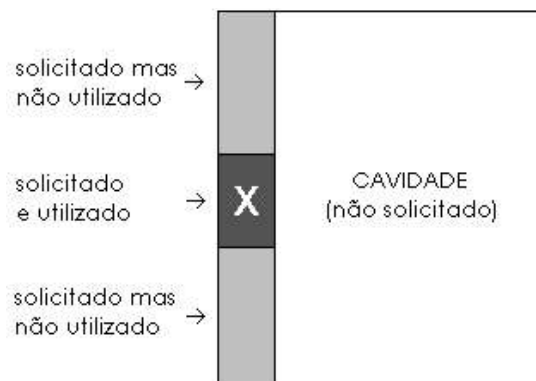
Em termos de posicionamento e exibição dos componentes num container, é útil entender três conceitos:

- Espaço não-solicitado (cavidade);
- Espaço solicitado mas não utilizado e
- Espaço solicitado e utilizado.

Quando você acomoda um widget, como um botão, ele é sempre acomodado ao longo de um dos quatro lados da cavidade. A opção “side” de pack especifica quais lados serão usados. Por exemplo, se especificamos “side=LEFT”, o widget será acomodado (isto é, posicionado) do lado esquerdo da cavidade.

Quando um widget é acomodado ao longo de um lado do container, é solicitado o lado inteiro, mesmo que ele não ocupe todo o espaço solicitado. Suponha que acomodemos um pequeno botão chamado X ao longo do lado esquerdo de uma grande cavidade, como na figura da próxima página.

A cavidade (área não solicitada) é agora ao lado direito do widget. O widget X solicitou o lado esquerdo inteiro, numa tira que é larga somente o suficiente para acomodá-lo. Por seu tamanho reduzido, o widget X usa somente uma pequena parte da área total que foi solicitada, somente o necessário para que ele apareça.



Como você pode ver, o widget X solicitou somente o espaço que necessitava para aparecer. Se especificarmos a opção de pack chamada “expand=YES”, ele irá solicitar toda a área disponível. Nenhum pedaço do lado direito da cavidade permanecerá não-solicitado. Isso não significa que o widget *usa* a área inteira; ele continua usando somente a pequena parte de que necessita.

Em tal situação, o widget X tem um espaço muito grande sem utilidade em torno dele. Em que lugar deste espaço ele deverá se posicionar? É isso que a opção “anchor” diz a ele. Quando um widget solicitou mais espaço do que realmente usa, a opção anchor refina suas coordenadas de posicionamento. “N” significa Norte (isto é, centrado no topo da área solicitada). “NE” significa Nordeste (isto é, acima e à direita da área solicitada). E assim por diante.

Já a opção “fill” serve para dizer ao widget se ele deve ou não ocupar todo o espaço livre, e em que direção ele deve se expandir:

- fill=NOME significa que o widget não deve se expandir;
- fill=X significa que ele deve se expandir na direção do eixo X (horizontalmente);
- fill=Y significa que ele deve se expandir na direção do eixo Y (verticalmente) e
- fill=BOTH significa que ele deve se expandir para todos os lados.

Rodando o programa

Ok, vamos rodar o programa. Você não precisa ler o código, só rodar o programa e experimentar com as várias opções de pack dos três botões-demonstração.

O frame do botão A dá uma cavidade horizontal para o botão se locomover – o frame não é muito mais alto que o botão; o frame do botão B dá a ele uma cavidade vertical para se locomover – o frame não é tão mais largo que o botão, e o frame do botão C tem uma enorme cavidade – muito mais alta e larga que o botão – para ele se divertir lá dentro.

Se a aparência de qualquer dos botões sob certas configurações surpreende você, tente descobrir por que ele se parece assim.

E finalmente...

Um jeito prático de encontrar erros

Veja que empacotar widgets em containeres é um negócio complicado porque o posicionamento de um widget em relação a outros widgets próximos depende em parte de como os widgets próximos foram acomodados. Isto é, se os outros widgets foram acomodados à esquerda, então a cavidade dentro da qual o próximo widget será empacotado será à direita. Mas se eles foram acomodados no topo da cavidade, então a cavidade dentro da qual o próximo widget será empacotado será abaixo deles. Tudo isso é uma bela salada!

Eis uma forma prática de encontrar erros: se você está desenvolvendo seu layout e se deparar com um problema – coisas que não funcionam da maneira que você pensava que deveriam funcionar – então dê a cada um dos seus containeres (isto é, cada um dos seus frames) uma cor de fundo diferente, por exemplo

bg = “red”, ou

bg = “cyan”, ou

bg = “tan”.

... ou amarelo, ou azul, ou vermelho, o que você quiser!


Isso permitirá a você ver como os frames estão se arranjando. Frequentemente, o que você vir lhe dará um indício de qual é o problema.



Meus agradecimentos a Pedro Werneck e Osvaldo Santana pela revisão e a Douglas Soares pelas observações.



TRADUZIDO PELO GRUPO PYTHON
UNESP – ILHA SOLTEIRA

Visite o site do Grupo Python! <http://geocities.yahoo.com.br/grupopython> 
Dúvidas sobre a tradução? Fale com o tradutor: labaki@feis.unesp.br