# coding=utf-8

#-------------------------------------------------------------------------------

# Name: Image processing tool

# Purpose: Process Images

# Author: shenglongshuai

# Created: 07/28/2017

#-------------------------------------------------------------------------------

from \_\_future\_\_ import division

from Tkinter import \*

import tkMessageBox

from tkSimpleDialog import askstring, askinteger

from PIL import Image, ImageTk

import ttk

import os

import glob

import random

from ScrolledText import ScrolledText

import cv2

import numpy as np

import glob as gb

import tkFileDialog

from tkSimpleDialog import askstring, askinteger

import matplotlib

import math

import time

from matplotlib import pyplot as plt

# colors for the bboxes

COLORS = ['red', 'blue', 'olive', 'teal', 'cyan', 'green', 'black']

# image sizes for the examples

SIZE = 180,180

class LabelTool():

def \_\_init\_\_(self, master):

# set up the main frame

self.parent = master

self.parent.title("桥梁裂缝图像处理软件v1.0")

# 设置窗口的大小宽x高+偏移量

#self.parent.geometry('800x500+500+200')

self.frame = Frame(self.parent)

self.frame.pack(fill=BOTH, expand=1)

self.parent.resizable(width = FALSE, height = FALSE)

# initialize global state

self.txtName = ''

self.imageDir = ''

self.imageList= []

self.egDir = ''

self.egList = []

self.outDir = ''

self.cur = 0

self.total = 0

self.category = 0

self.imagename = ''

self.labelfilename = ''

self.tkimg = None

self.currentLabelclass = ''

self.cla\_can\_temp = []

self.classcandidate\_filename = 'Classes/class.txt'

self.index = ''

self.annotationPath = ''

self.imageHeight = ''

self.imageDepth = 3

self.imageWidth = ''

self.sizePath=''

# initialize mouse state

self.STATE = {}

self.STATE['click'] = 0

self.STATE['x'], self.STATE['y'] = 0, 0

# reference to bbox

self.bboxIdList = []

self.bboxId = None

self.bboxList = []

self.hl = None

self.vl = None

# ----------------- GUI stuff ---------------------

# dir entry & load

self.label = Label(self.frame, text = "图像集序号：")

self.label.grid(row = 0, column = 1, sticky = W)

self.entry = Entry(self.frame)

self.entry.grid(row = 0, column = 1)

self.ldBtn = Button(self.frame, text = " 加 载 ", command = self.loadDir)

self.ldBtn.grid(row = 0, column = 1,sticky = E)

# main panel for labeling

self.scrollpane = Frame(self.frame, bd=2, relief=SUNKEN)

self.xscrollbar = Scrollbar(self.scrollpane, orient=HORIZONTAL)

self.xscrollbar.grid(row=0, column=1, sticky=E + W)

self.yscrollbar = Scrollbar(self.scrollpane)

#self.yscrollbar.pack(side=LEFT)

self.yscrollbar.grid(row=1, column=0, sticky=W + N + S)

self.mainPanel = Canvas(self.scrollpane, bd=0, cursor='tcross', scrollregion=(0, 0, 3000, 1800)

, xscrollcommand=self.xscrollbar.set, yscrollcommand=self.yscrollbar.set)

self.mainPanel.grid(row=1, column=1, sticky=N + S + E +W)

self.xscrollbar.config(command=self.mainPanel.xview)

self.yscrollbar.config(command=self.mainPanel.yview)

self.mainPanel.config(scrollregion=self.mainPanel.bbox(ALL))

self.parent.bind("<Escape>", self.cancelBBox) # press <Espace> to cancel current bbox

self.parent.bind("s", self.cancelBBox)

self.parent.bind("a", self.prevImage) # press 'a' to go backforward

self.parent.bind("d", self.nextImage) # press 'd' to go forward

self.scrollpane.grid(row=1, column=1, rowspan=4, sticky=W + N)

# menu for choose class

self.menubar = Menu(root)

filemenu = Menu(self.menubar, tearoff=0)

self.menubar.add\_cascade(label="文件", menu=filemenu)

openmenu = Menu(filemenu, tearoff=0)

filemenu.add\_cascade(label="打开", menu=openmenu)

openmenu.add\_command(label="图像目录", command=self.openImages)

#openmenu.add\_command(label="标签目录", command=self.openLabel)

openmenu.add\_command(label="图片文件夹", command=self.openFile)

filemenu.add\_separator()

filemenu.add\_command(label="退出", command=self.frame.quit)

toolmenu = Menu(self.menubar, tearoff=0)

self.menubar.add\_cascade(label="工具", menu=toolmenu)

self.edgemenu = Menu(toolmenu, tearoff=0)

toolmenu.add\_cascade(label="边缘检测",menu=self.edgemenu)

self.edgemenu.add\_command(label="canny",command=self.canny)

#toolmenu.add\_command(label="单张canny", command=self.cannybar)

#self.cannymenu.add\_command(label="批量canny", command=self.canny)

self.edgemenu.add\_command(label="sobel", command=self.sobel)

self.edgemenu.add\_command(label="laplacian", command=self.laplacian)

toolmenu.add\_separator()

self.binarymenu = Menu(toolmenu, tearoff=0)

toolmenu.add\_cascade(label='二值化处理', menu=self.binarymenu)

self.binarymenu.add\_command(label="Gray Image", command=self.GrayImage)

self.binarymenu.add\_command(label="BINARY", command=self.BINARY)

self.binarymenu.add\_command(label="BINARY\_INV", command=self.BINARY\_INV)

self.binarymenu.add\_command(label="TRUNC", command=self.TRUNC)

self.binarymenu.add\_command(label="TOZERO", command=self.TOZERO)

self.binarymenu.add\_command(label="TOZERO\_INV", command=self.TOZERO\_INV)

toolmenu.add\_separator()

self.angularmenu = Menu(toolmenu, tearoff=0)

toolmenu.add\_cascade(label="角点检测", menu=self.angularmenu)

self.angularmenu.add\_command(label="harris", command=self.harris)

self.angularmenu.add\_command(label="Shi-Tomasi", command=self.Shi\_Tomasi)

self.angularmenu.add\_command(label="brisk", command=self.brisk)

aboutmenu = Menu(self.menubar, tearoff=0)

self.menubar.add\_cascade(label="关于", menu=aboutmenu)

aboutmenu.add\_command(label="捐赠", command=self.Email)

aboutmenu.add\_command(label="联系作者", command=self.Email)

root.config(menu=self.menubar)

self.classname = StringVar()

self.classcandidate = ttk.Combobox(self.frame, state='readonly', textvariable=self.classname)

#self.classcandidate.grid(row=1, column=2)

if os.path.exists(self.classcandidate\_filename):

with open(self.classcandidate\_filename) as cf:

for line in cf.readlines():

# print line

self.cla\_can\_temp.append(line.strip('\n'))

# print self.cla\_can\_temp

self.classcandidate['values'] = self.cla\_can\_temp

self.currentLabelclass = self.classcandidate.get() # init

self.btnclass = Button(self.frame, text='确认', command=self.setClass)

#self.btnclass.grid(row=2, column=2, sticky=W + E)

# showing bbox info & delete bbox

self.lb1 = Label(self.frame, text='标注结果：')

#self.lb1.grid(row=3, column=2, sticky=W + N)

self.listbox = Listbox(self.frame, width=22, height=12)

#self.listbox.grid(row=4, column=2, sticky=N + S)

self.btnDel = Button(self.frame, text='删除', command=self.delBBox)

#self.btnDel.grid(row=5, column=2, sticky=W + E + N)

self.btnClear = Button(self.frame, text='清空', command=self.clearBBox)

#self.btnClear.grid(row=6, column=2, sticky=W + E + N)

# control panel for image navigation

self.ctrPanel = Frame(self.frame)

self.ctrPanel.grid(row = 7, column = 1, columnspan = 2, sticky = W+E)

self.prevBtn = Button(self.ctrPanel, text='<< 上一张', width = 10, command = self.prevImage)

self.prevBtn.pack(side = LEFT, padx = 5, pady = 3)

self.nextBtn = Button(self.ctrPanel, text='下一张 >>', width = 10, command = self.nextImage)

self.nextBtn.pack(side = LEFT, padx = 1, pady = 3)

self.progLabel = Label(self.ctrPanel, text = "页码: / ")

self.progLabel.pack( side = LEFT,padx = 1)

self.tmpLabel = Label(self.ctrPanel, text = "跳转到图片 No.")

self.tmpLabel.pack(side = LEFT,padx = 1)

self.idxEntry = Entry(self.ctrPanel, width = 5)

self.idxEntry.pack(side = LEFT)

self.goBtn = Button(self.ctrPanel, text = 'Go', command = self.gotoImage)

self.goBtn.pack(side = RIGHT)

# example pannel for illustration

self.egPanel = Frame(self.frame, border = 1)

self.egPanel.grid(row = 1, column = 0, rowspan = 5, sticky = N)

self.tmpLabel2 = Label(self.egPanel, text = "样例：")

self.tmpLabel2.pack(side =LEFT, pady = 0)

self.egLabels = []

for i in range(3):

self.egLabels.append(Label(self.egPanel))

self.egLabels[-1].pack(side=TOP)

# display mouse position

self.disp = Label(self.ctrPanel, text='')

self.disp.pack(side = RIGHT)

self.frame.columnconfigure(1, weight = 1)

self.frame.rowconfigure(4, weight = 1)

# display dialog

self.dialogPanel = Frame(self.frame, border=10)

self.dialogPanel.grid(row=1, column=0, rowspan=5, sticky=S)

self.dialogLabel = Label(self.dialogPanel, text=" 日志:")

self.dialogLabel.pack(side=BOTTOM, pady=5)

self.dialogLabel.grid(row=4, column=0, sticky=W + S)

self.dialogText = ScrolledText(self.dialogPanel, width=30, height=20)

self.dialogText.config(state=DISABLED)

self.dialogText.grid(row=5, column=0, sticky=W + S)

# for debugging

# self.setImage()

# self.loadDir()

def canny(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

numMin = askinteger('请输入参数范围', "最小值")

numMax = askinteger('请输入参数范围', "最大值")

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath, 0)

img = cv2.GaussianBlur(img, (3, 3), 0)

canny = cv2.Canny(img, numMax, numMin) # 此处参数是范围，数值越大越不细致，具体数值根据图片情况更改

win = cv2.namedWindow('Canny', flags=0)

cv2.imshow('Canny', canny)

# cv2.imwrite("C:/Users/sls/Desktop/2//" + eachDir, canny) # 图片输出的目录，更改成自己需要的

cv2.imwrite(savePath + '/' + eachDir, canny)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Canny边缘检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def sobel(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/'

path = self.filePath + '/'

pathDir = os.listdir(path)

numMin = askinteger('请输入求导次数', "dx")

numMax = askinteger('请输入求导次数', "dy")

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

x = cv2.Sobel(img, cv2.CV\_16S, numMin, 0)

y = cv2.Sobel(img, cv2.CV\_16S, 0, numMax)

absX = cv2.convertScaleAbs(x) # 转回uint8

absY = cv2.convertScaleAbs(y)

dst = cv2.addWeighted(absX, 0.5, absY, 0.5, 0)

win = cv2.namedWindow('Sobel', flags=0)

cv2.imshow("Sobel", dst)

# cv2.imwrite("C:/Users/sls/Desktop/1/" + eachDir, canny) # 图片输出的目录，更改成自己需要的

cv2.imwrite(savePath + '/' + eachDir, dst)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Sobel边缘检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def laplacian(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/'

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath, 0)

gray\_lap = cv2.Laplacian(img, cv2.CV\_16S, ksize=3)

dst = cv2.convertScaleAbs(gray\_lap)

win = cv2.namedWindow('Laplacian', flags=0)

cv2.imshow('Laplacian', dst)

# cv2.imwrite("C:/Users/sls/Desktop/laplacian/" + eachDir, canny) # 图片输出的目录，更改成自己需要的

cv2.imwrite(savePath + '/' + eachDir, dst)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Laplacian边缘检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def cannybar(self):

def CannyThreshold(lowThreshold):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

detected\_edges = cv2.GaussianBlur(gray, (3, 3), 0)

detected\_edges = cv2.Canny(detected\_edges, lowThreshold, lowThreshold \* ratio, apertureSize=kernel\_size)

dst = cv2.bitwise\_and(img, img, mask=detected\_edges)

cv2.imshow('canny demo', dst)

cv2.imwrite(savePath + '/' + eachDir, dst)

cv2.waitKey(0)

cv2.destroyAllWindows()

lowThreshold = 0

max\_lowThreshold = 100

ratio = 3

kernel\_size = 3

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

print img

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

cv2.namedWindow('canny demo')

cv2.createTrackbar('Min threshold', 'canny demo', lowThreshold, max\_lowThreshold, CannyThreshold)

CannyThreshold(0) # initialization

def harris(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

gray = np.float32(gray)

dst = cv2.cornerHarris(gray, 2, 3, 0.04)

# result is dilated for marking the corners, not important

dst = cv2.dilate(dst, None)

# Threshold for an optimal value, it may vary depending on the image.

img[dst > 0.01 \* dst.max()] = [0, 0, 255]

win = cv2.namedWindow('Harris', flags=0)

cv2.imshow('Harris', img)

cv2.imwrite(savePath + '/' + eachDir, img)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Harris角点检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def Shi\_Tomasi (self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/'

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

corners = cv2.goodFeaturesToTrack(gray, 25, 0.01, 10)

# 返回的结果是[[ 311., 250.]] 两层括号的数组。

corners = np.int0(corners)

for i in corners:

x, y = i.ravel()

cv2.circle(img, (x, y), 3, 255, -1)

win = cv2.namedWindow('Shi\_Tomasi', flags=0)

cv2.imshow('Shi\_Tomasi', img)

cv2.imwrite(savePath + '/' + eachDir, img)

cv2.waitKey(0)

self.dialogText.config(state=NORMAL)

ss = '\n Shi\_Tomasi角点检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def brisk(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

brisk = cv2.BRISK\_create()

(kpt, desc) = brisk.detectAndCompute(img, None)

bk\_img = img.copy()

out\_img = img.copy()

out\_img = cv2.drawKeypoints(bk\_img, kpt, out\_img)

win = cv2.namedWindow('brisk', flags=0)

cv2.imshow('brisk', out\_img)

cv2.imwrite(savePath + '/' + eachDir, out\_img)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Brisk角点检测的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def GrayImage(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

gray = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

win = cv2.namedWindow('Gray Image', flags=0)

cv2.imshow('Gray Image', gray)

cv2.imwrite(savePath + '/' + eachDir, gray)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n Gray二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def BINARY(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

GrayImage = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, thresh1 = cv2.threshold(GrayImage, 127, 255, cv2.THRESH\_BINARY)

win = cv2.namedWindow('BINARY', flags=0)

cv2.imshow('BINARY', thresh1)

cv2.imwrite(savePath + '/' + eachDir, thresh1)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n BINARY二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def BINARY\_INV(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

GrayImage = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, thresh2 = cv2.threshold(GrayImage, 127, 255, cv2.THRESH\_BINARY\_INV)

win = cv2.namedWindow('BINARY\_INV', flags=0)

cv2.imshow('BINARY\_INV', thresh2)

cv2.imwrite(savePath + '/' + eachDir, thresh2)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n BINARY\_INV二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def TRUNC(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

GrayImage = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, thresh3 = cv2.threshold(GrayImage, 127, 255, cv2.THRESH\_TRUNC)

win = cv2.namedWindow('TRUNC', flags=0)

cv2.imshow('TRUNC', thresh3)

cv2.imwrite(savePath + '/' + eachDir, thresh3)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n TRUNC二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def TOZERO(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

GrayImage = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, thresh4 = cv2.threshold(GrayImage, 127, 255, cv2.THRESH\_TOZERO)

win = cv2.namedWindow('TOZERO', flags=0)

cv2.imshow('TOZERO', thresh4)

cv2.imwrite(savePath + '/' + eachDir, thresh4)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n TOZERO二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def TOZERO\_INV(self):

default\_dir = ""

savePath = tkFileDialog.askdirectory(title=u"保存路径", initialdir=(os.path.expanduser(default\_dir)))

# path = 'C:/Users/sls/Desktop/1/' # 图片读取的目录，更改成自己需要的

path = self.filePath + '/'

pathDir = os.listdir(path)

for eachDir in pathDir:

imgPath = path + eachDir

print imgPath

img = cv2.imread(imgPath)

GrayImage = cv2.cvtColor(img, cv2.COLOR\_BGR2GRAY)

ret, thresh5 = cv2.threshold(GrayImage, 127, 255, cv2.THRESH\_TOZERO\_INV)

win = cv2.namedWindow('TOZERO\_INV', flags=0)

cv2.imshow('TOZERO\_INV', thresh5)

cv2.imwrite(savePath + '/' + eachDir, thresh5)

cv2.waitKey(0)

cv2.destroyAllWindows()

self.dialogText.config(state=NORMAL)

ss = '\n TOZERO\_INV二值化处理的图片已保存\n'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S", time.localtime()) + '%s' % (ss))

def loadDir(self, dbg = False):

if not dbg:

s = self.entry.get()

self.parent.focus()

self.category = int(s)

else:

s = r'D:\workspace\python\labelGUI'

# get image list

self.imageDir = os.path.join(r'./Images', '%03d' %(self.category))

# print self.imageDir

self.index = self.imageDir.split('\\')

# print self.index

#print self.category

self.annotationPath = r'annotation/' + self.index[1].replace('\n', '') + '/'

# print self.annotationPath

self.imageList = glob.glob(os.path.join(self.imageDir, '\*.\*'))

#print self.imageList

if len(self.imageList) == 0:

print '图片集输入错误'

tkMessageBox.showerror("error", "请输入正确的图片集序号")

self.dialogText.config(state=NORMAL)

self.dialogText.insert(0.0, '图片集输入错误\n')

self.dialogText.config(state=DISABLED)

return

# default to the 1st image in the collection

self.cur = 1

self.total = len(self.imageList)

# set up output dir

self.outDir = os.path.join(r'./Labels', '%03d' %self.category)

if not os.path.exists(self.outDir):

os.mkdir(self.outDir)

# load example bboxes

# self.egDir = os.path.join(r'./Examples', '%03d' %(self.category))

self.egDir = os.path.join(r'./Examples/demo')

print os.path.exists(self.egDir)

if not os.path.exists(self.egDir):

return

filelist = glob.glob(os.path.join(self.egDir, '\*.\*'))

self.tmp = []

self.egList = []

random.shuffle(filelist)

for (i, f) in enumerate(filelist):

if i == 3:

break

im = Image.open(f)

r = min(SIZE[0] / im.size[0], SIZE[1] / im.size[1])

new\_size = int(r \* im.size[0]), int(r \* im.size[1])

self.tmp.append(im.resize(new\_size, Image.ANTIALIAS))

self.egList.append(ImageTk.PhotoImage(self.tmp[-1]))

self.egLabels[i].config(image = self.egList[-1], width = SIZE[0], height = SIZE[1])

self.loadImage()

print '%d images loaded from %s' %(self.total, s)

def loadImage(self):

# load image

imagepath = self.imageList[self.cur - 1]

self.img = Image.open(imagepath)

self.tkimg = ImageTk.PhotoImage(self.img)

self.imageHeight = str(self.img.size[1])

self.imageWidth = str(self.img.size[0])

self.mainPanel.config(width = max(self.tkimg.width(), 400), height = max(self.tkimg.height(), 400))

#self.mainPanel.config(width=max(self.tkimg.width(), 400), height=max(self.tkimg.height(), 400), yscrollcommand=scr1.set)

self.mainPanel.create\_image(0, 0, image = self.tkimg, anchor=NW)

self.progLabel.config(text = "%04d/%04d" %(self.cur, self.total))

# load labels

self.clearBBox()

self.imagename = os.path.split(imagepath)[-1].split('.')[0]

labelname = self.imagename + '.txt'

self.labelfilename = os.path.join(self.outDir, labelname)

bbox\_cnt = 0

#self.sizePath = r'Size/' + self.index[1] + '/'

if os.path.exists(self.labelfilename):

with open(self.labelfilename) as f:

for (i, line) in enumerate(f):

if i == 0:

bbox\_cnt = int(line.strip())

continue

# tmp = [int(t.strip()) for t in line.split()]

tmp = line.split()

#print tmp

self.bboxList.append(tuple(tmp))

tmpId = self.mainPanel.create\_rectangle(int(tmp[0]), int(tmp[1]),

int(tmp[2]), int(tmp[3]),

width = 2,

outline = COLORS[(len(self.bboxList)-1) % len(COLORS)])

# print tmpId

self.bboxIdList.append(tmpId)

self.listbox.insert(END, '%s : (%d, %d) -> (%d, %d)' %(tmp[4],int(tmp[0]), int(tmp[1]),int(tmp[2]), int(tmp[3])))

self.listbox.itemconfig(len(self.bboxIdList) - 1, fg = COLORS[(len(self.bboxIdList) - 1) % len(COLORS)])

def saveImage(self):

with open(self.labelfilename, 'w') as f:

f.write('%d\n' %len(self.bboxList))

for bbox in self.bboxList:

f.write(' '.join(map(str, bbox)) + '\n')

# print self.sizePath+self.txtName+'.txt'

#self.writeSize(self.sizePath)

# self.writeSize(size)

print 'Image No. %d saved' %(self.cur)

self.dialogText.config(state=NORMAL)

self.dialogText.insert(0.0, 'Image No. %d \n' %(self.cur))

self.dialogText.config(state=DISABLED)

# tkMessageBox.showinfo("", "保存成功")

def cancelBBox(self, event):

if 1 == self.STATE['click']:

if self.bboxId:

self.mainPanel.delete(self.bboxId)

self.bboxId = None

self.STATE['click'] = 0

def delBBox(self):

sel = self.listbox.curselection()

if len(sel) != 1 :

return

idx = int(sel[0])

self.mainPanel.delete(self.bboxIdList[idx])

self.bboxIdList.pop(idx)

self.bboxList.pop(idx)

self.listbox.delete(idx)

def clearBBox(self):

for idx in range(len(self.bboxIdList)):

self.mainPanel.delete(self.bboxIdList[idx])

self.listbox.delete(0, len(self.bboxList))

self.bboxIdList = []

self.bboxList = []

def prevImage(self, event = None):

if self.cur > 1:

self.cur -= 1

self.loadImage()

def nextImage(self, event = None):

if self.cur < self.total:

self.cur += 1

self.loadImage()

else:

tkMessageBox.showinfo("喵", "已到最后一张图片")

def gotoImage(self):

idx = int(self.idxEntry.get())

if 1 <= idx and idx <= self.total:

self.cur = idx

self.loadImage()

else:

tkMessageBox.showwarning("warning", "请输入1到%d之间的数" % self.total, )

self.dialogText.config(state=NORMAL)

self.dialogText.insert(0.0, '请输入1到%d之间的数\n' % self.total)

self.dialogText.config(state=DISABLED)

def setClass(self):

self.currentLabelclass = self.classcandidate.get()

print 'set label class to :', self.currentLabelclass

def setImage(self, imagepath = r'test2.png'):

self.img = Image.open(imagepath)

self.tkimg = ImageTk.PhotoImage(self.img)

self.mainPanel.config(width = self.tkimg.width())

self.mainPanel.config(height = self.tkimg.height())

self.mainPanel.create\_image(0, 0, image = self.tkimg, anchor=NW)

# def getAnnotation(self):

# self.get

def openImages(self):

os.startfile(self.imageDir)

def openLabel(self):

os.startfile(self.outDir)

def Email(self):

tkMessageBox.showinfo("作者邮箱", "作者邮箱：\nlicekong@gmail.com\nshenglongshuai@outlook.com")

def openFile(self):

default\_dir = ""

fpath = tkFileDialog.askdirectory(title=u"选择文件", initialdir=(os.path.expanduser(default\_dir)))

self.filePath = fpath

for filename in os.listdir(fpath): # listdir的参数是文件夹的路径

print (filename )

self.dialogText.config(state=NORMAL)

ss='\n 已从文件夹读取图片，请从工具里选择对图片进行的操作'

self.dialogText.insert(0.0, time.strftime("%Y-%m-%d %H:%M:%S",time.localtime())+'%s' %(ss))

if \_\_name\_\_ == '\_\_main\_\_':

root = Tk()

tool = LabelTool(root)

root.resizable(width = True, height = True)

root.mainloop()

#mysetup.py

from distutils.core import setup

import py2exe

setup(

options = {

"py2exe": {

"dll\_excludes": ["MSVCP90.dll"],

"includes":["sip"]

}

},

windows=["main.py"],

data\_files=[("icon",["icon/del.jpg"])],

#data\_files=[("icon",["icon/del.gif"]),("",["dataPool.xml"])],

)

class CallWrapper:

"""Internal class. Stores function to call when some user

defined Tcl function is called e.g. after an event occurred."""

def \_\_init\_\_(self, func, subst, widget):

"""Store FUNC, SUBST and WIDGET as members."""

self.func = func

self.subst = subst

self.widget = widget

def \_\_call\_\_(self, \*args):

"""Apply first function SUBST to arguments, than FUNC."""

try:

if self.subst:

args = self.subst(\*args)

return self.func(\*args)

except SystemExit, msg:

raise SystemExit, msg

except:

self.widget.\_report\_exception()

class XView:

"""Mix-in class for querying and changing the horizontal position

of a widget's window."""

def xview(self, \*args):

"""Query and change the horizontal position of the view."""

res = self.tk.call(self.\_w, 'xview', \*args)

if not args:

return self.\_getdoubles(res)

def xview\_moveto(self, fraction):

"""Adjusts the view in the window so that FRACTION of the

total width of the canvas is off-screen to the left."""

self.tk.call(self.\_w, 'xview', 'moveto', fraction)

def xview\_scroll(self, number, what):

"""Shift the x-view according to NUMBER which is measured in "units"

or "pages" (WHAT)."""

self.tk.call(self.\_w, 'xview', 'scroll', number, what)

class YView:

"""Mix-in class for querying and changing the vertical position

of a widget's window."""

def yview(self, \*args):

"""Query and change the vertical position of the view."""

res = self.tk.call(self.\_w, 'yview', \*args)

if not args:

return self.\_getdoubles(res)

def yview\_moveto(self, fraction):

"""Adjusts the view in the window so that FRACTION of the

total height of the canvas is off-screen to the top."""

self.tk.call(self.\_w, 'yview', 'moveto', fraction)

def yview\_scroll(self, number, what):

"""Shift the y-view according to NUMBER which is measured in

"units" or "pages" (WHAT)."""

self.tk.call(self.\_w, 'yview', 'scroll', number, what)

class Wm:

"""Provides functions for the communication with the window manager."""

def wm\_aspect(self,

minNumer=None, minDenom=None,

maxNumer=None, maxDenom=None):

"""Instruct the window manager to set the aspect ratio (width/height)

of this widget to be between MINNUMER/MINDENOM and MAXNUMER/MAXDENOM. Return a tuple

of the actual values if no argument is given."""

return self.\_getints(

self.tk.call('wm', 'aspect', self.\_w,

minNumer, minDenom,

maxNumer, maxDenom))

aspect = wm\_aspect

def wm\_attributes(self, \*args):

"""This subcommand returns or sets platform specific attributes

The first form returns a list of the platform specific flags and

their values. The second form returns the value for the specific

option. The third form sets one or more of the values. The values

are as follows:

On Windows, -disabled gets or sets whether the window is in a

disabled state. -toolwindow gets or sets the style of the window

to toolwindow (as defined in the MSDN). -topmost gets or sets

whether this is a topmost window (displays above all other

windows).

On Macintosh, XXXXX

On Unix, there are currently no special attribute values.

"""

args = ('wm', 'attributes', self.\_w) + args

return self.tk.call(args)

attributes=wm\_attributes

def wm\_client(self, name=None):

"""Store NAME in WM\_CLIENT\_MACHINE property of this widget. Return

current value."""

return self.tk.call('wm', 'client', self.\_w, name)

client = wm\_client

def wm\_colormapwindows(self, \*wlist):

"""Store list of window names (WLIST) into WM\_COLORMAPWINDOWS property

of this widget. This list contains windows whose colormaps differ from their

parents. Return current list of widgets if WLIST is empty."""

if len(wlist) > 1:

wlist = (wlist,) # Tk needs a list of windows here

args = ('wm', 'colormapwindows', self.\_w) + wlist

if wlist:

self.tk.call(args)

else:

return map(self.\_nametowidget, self.tk.splitlist(self.tk.call(args)))

colormapwindows = wm\_colormapwindows

def wm\_command(self, value=None):

"""Store VALUE in WM\_COMMAND property. It is the command

which shall be used to invoke the application. Return current

command if VALUE is None."""

return self.tk.call('wm', 'command', self.\_w, value)

command = wm\_command

def wm\_deiconify(self):

"""Deiconify this widget. If it was never mapped it will not be mapped.

On Windows it will raise this widget and give it the focus."""

return self.tk.call('wm', 'deiconify', self.\_w)

deiconify = wm\_deiconify

def wm\_focusmodel(self, model=None):

"""Set focus model to MODEL. "active" means that this widget will claim

the focus itself, "passive" means that the window manager shall give

the focus. Return current focus model if MODEL is None."""

return self.tk.call('wm', 'focusmodel', self.\_w, model)

focusmodel = wm\_focusmodel

def wm\_frame(self):

"""Return identifier for decorative frame of this widget if present."""

return self.tk.call('wm', 'frame', self.\_w)

frame = wm\_frame

def wm\_geometry(self, newGeometry=None):

"""Set geometry to NEWGEOMETRY of the form =widthxheight+x+y. Return

current value if None is given."""

return self.tk.call('wm', 'geometry', self.\_w, newGeometry)

geometry = wm\_geometry

def wm\_grid(self,

baseWidth=None, baseHeight=None,

widthInc=None, heightInc=None):

"""Instruct the window manager that this widget shall only be

resized on grid boundaries. WIDTHINC and HEIGHTINC are the width and

height of a grid unit in pixels. BASEWIDTH and BASEHEIGHT are the

number of grid units requested in Tk\_GeometryRequest."""

return self.\_getints(self.tk.call(

'wm', 'grid', self.\_w,

baseWidth, baseHeight, widthInc, heightInc))

grid = wm\_grid

def wm\_group(self, pathName=None):

"""Set the group leader widgets for related widgets to PATHNAME. Return

the group leader of this widget if None is given."""

return self.tk.call('wm', 'group', self.\_w, pathName)

group = wm\_group

def wm\_iconbitmap(self, bitmap=None, default=None):

"""Set bitmap for the iconified widget to BITMAP. Return

the bitmap if None is given.

Under Windows, the DEFAULT parameter can be used to set the icon

for the widget and any descendents that don't have an icon set

explicitly. DEFAULT can be the relative path to a .ico file

(example: root.iconbitmap(default='myicon.ico') ). See Tk

documentation for more information."""

if default:

return self.tk.call('wm', 'iconbitmap', self.\_w, '-default', default)

else:

return self.tk.call('wm', 'iconbitmap', self.\_w, bitmap)

iconbitmap = wm\_iconbitmap

def wm\_iconify(self):

"""Display widget as icon."""

return self.tk.call('wm', 'iconify', self.\_w)

iconify = wm\_iconify

def wm\_iconmask(self, bitmap=None):

"""Set mask for the icon bitmap of this widget. Return the

mask if None is given."""

return self.tk.call('wm', 'iconmask', self.\_w, bitmap)

iconmask = wm\_iconmask

def wm\_iconname(self, newName=None):

"""Set the name of the icon for this widget. Return the name if

None is given."""

return self.tk.call('wm', 'iconname', self.\_w, newName)

iconname = wm\_iconname

def wm\_iconposition(self, x=None, y=None):

"""Set the position of the icon of this widget to X and Y. Return

a tuple of the current values of X and X if None is given."""

return self.\_getints(self.tk.call(

'wm', 'iconposition', self.\_w, x, y))

iconposition = wm\_iconposition

def wm\_iconwindow(self, pathName=None):

"""Set widget PATHNAME to be displayed instead of icon. Return the current

value if None is given."""

return self.tk.call('wm', 'iconwindow', self.\_w, pathName)

iconwindow = wm\_iconwindow

def wm\_maxsize(self, width=None, height=None):

"""Set max WIDTH and HEIGHT for this widget. If the window is gridded

the values are given in grid units. Return the current values if None

is given."""

return self.\_getints(self.tk.call(

'wm', 'maxsize', self.\_w, width, height))

maxsize = wm\_maxsize

def wm\_minsize(self, width=None, height=None):

"""Set min WIDTH and HEIGHT for this widget. If the window is gridded

the values are given in grid units. Return the current values if None

is given."""

return self.\_getints(self.tk.call(

'wm', 'minsize', self.\_w, width, height))

minsize = wm\_minsize

def wm\_overrideredirect(self, boolean=None):

"""Instruct the window manager to ignore this widget

if BOOLEAN is given with 1. Return the current value if None

is given."""

return self.\_getboolean(self.tk.call(

'wm', 'overrideredirect', self.\_w, boolean))

overrideredirect = wm\_overrideredirect

def wm\_positionfrom(self, who=None):

"""Instruct the window manager that the position of this widget shall

be defined by the user if WHO is "user", and by its own policy if WHO is

"program"."""

return self.tk.call('wm', 'positionfrom', self.\_w, who)

positionfrom = wm\_positionfrom

def wm\_protocol(self, name=None, func=None):

"""Bind function FUNC to command NAME for this widget.

Return the function bound to NAME if None is given. NAME could be

e.g. "WM\_SAVE\_YOURSELF" or "WM\_DELETE\_WINDOW"."""

if hasattr(func, '\_\_call\_\_'):

command = self.\_register(func)

else:

command = func

return self.tk.call(

'wm', 'protocol', self.\_w, name, command)

protocol = wm\_protocol

def wm\_resizable(self, width=None, height=None):

"""Instruct the window manager whether this width can be resized

in WIDTH or HEIGHT. Both values are boolean values."""

return self.tk.call('wm', 'resizable', self.\_w, width, height)

resizable = wm\_resizable

def wm\_sizefrom(self, who=None):

"""Instruct the window manager that the size of this widget shall

be defined by the user if WHO is "user", and by its own policy if WHO is

"program"."""

return self.tk.call('wm', 'sizefrom', self.\_w, who)

sizefrom = wm\_sizefrom

def wm\_state(self, newstate=None):

"""Query or set the state of this widget as one of normal, icon,

iconic (see wm\_iconwindow), withdrawn, or zoomed (Windows only)."""

return self.tk.call('wm', 'state', self.\_w, newstate)

state = wm\_state

def wm\_title(self, string=None):

"""Set the title of this widget."""

return self.tk.call('wm', 'title', self.\_w, string)

title = wm\_title

def wm\_transient(self, master=None):

"""Instruct the window manager that this widget is transient

with regard to widget MASTER."""

return self.tk.call('wm', 'transient', self.\_w, master)

transient = wm\_transient

def wm\_withdraw(self):

"""Withdraw this widget from the screen such that it is unmapped

and forgotten by the window manager. Re-draw it with wm\_deiconify."""

return self.tk.call('wm', 'withdraw', self.\_w)

withdraw = wm\_withdraw

class Tk(Misc, Wm):

"""Toplevel widget of Tk which represents mostly the main window

of an application. It has an associated Tcl interpreter."""

\_w = '.'

def \_\_init\_\_(self, screenName=None, baseName=None, className='Tk',

useTk=1, sync=0, use=None):

"""Return a new Toplevel widget on screen SCREENNAME. A new Tcl interpreter will

be created. BASENAME will be used for the identification of the profile file (see

readprofile).

It is constructed from sys.argv[0] without extensions if None is given. CLASSNAME

is the name of the widget class."""

self.master = None

self.children = {}

self.\_tkloaded = 0

# to avoid recursions in the getattr code in case of failure, we

# ensure that self.tk is always \_something\_.

self.tk = None

if baseName is None:

import os

baseName = os.path.basename(sys.argv[0])

baseName, ext = os.path.splitext(baseName)

if ext not in ('.py', '.pyc', '.pyo'):

baseName = baseName + ext

interactive = 0

self.tk = \_tkinter.create(screenName, baseName, className, interactive, wantobjects, useTk, sync, use)

if useTk:

self.\_loadtk()

if not sys.flags.ignore\_environment:

# Issue #16248: Honor the -E flag to avoid code injection.

self.readprofile(baseName, className)

def loadtk(self):

if not self.\_tkloaded:

self.tk.loadtk()

self.\_loadtk()

def \_loadtk(self):

self.\_tkloaded = 1

global \_default\_root

# Version sanity checks

tk\_version = self.tk.getvar('tk\_version')

if tk\_version != \_tkinter.TK\_VERSION:

raise RuntimeError, \

"tk.h version (%s) doesn't match libtk.a version (%s)" \

% (\_tkinter.TK\_VERSION, tk\_version)

# Under unknown circumstances, tcl\_version gets coerced to float

tcl\_version = str(self.tk.getvar('tcl\_version'))

if tcl\_version != \_tkinter.TCL\_VERSION:

raise RuntimeError, \

"tcl.h version (%s) doesn't match libtcl.a version (%s)" \

% (\_tkinter.TCL\_VERSION, tcl\_version)

if TkVersion < 4.0:

raise RuntimeError, \

"Tk 4.0 or higher is required; found Tk %s" \

% str(TkVersion)

# Create and register the tkerror and exit commands

# We need to inline parts of \_register here, \_ register

# would register differently-named commands.

if self.\_tclCommands is None:

self.\_tclCommands = []

self.tk.createcommand('tkerror', \_tkerror)

self.tk.createcommand('exit', \_exit)

self.\_tclCommands.append('tkerror')

self.\_tclCommands.append('exit')

if \_support\_default\_root and not \_default\_root:

\_default\_root = self

self.protocol("WM\_DELETE\_WINDOW", self.destroy)

def destroy(self):

"""Destroy this and all descendants widgets. This will

end the application of this Tcl interpreter."""

for c in self.children.values(): c.destroy()

self.tk.call('destroy', self.\_w)

Misc.destroy(self)

global \_default\_root

if \_support\_default\_root and \_default\_root is self:

\_default\_root = None

def readprofile(self, baseName, className):

"""Internal function. It reads BASENAME.tcl and CLASSNAME.tcl into

the Tcl Interpreter and calls execfile on BASENAME.py and CLASSNAME.py if

such a file exists in the home directory."""

import os

if 'HOME' in os.environ: home = os.environ['HOME']

else: home = os.curdir

class\_tcl = os.path.join(home, '.%s.tcl' % className)

class\_py = os.path.join(home, '.%s.py' % className)

base\_tcl = os.path.join(home, '.%s.tcl' % baseName)

base\_py = os.path.join(home, '.%s.py' % baseName)

dir = {'self': self}

exec 'from Tkinter import \*' in dir

if os.path.isfile(class\_tcl):

self.tk.call('source', class\_tcl)

if os.path.isfile(class\_py):

execfile(class\_py, dir)

if os.path.isfile(base\_tcl):

self.tk.call('source', base\_tcl)

if os.path.isfile(base\_py):

execfile(base\_py, dir)

def report\_callback\_exception(self, exc, val, tb):

"""Report callback exception on sys.stderr.

Applications may want to override this internal function, and

should when sys.stderr is None."""

import traceback, sys

print >>sys.stderr, "Exception in Tkinter callback"

sys.last\_type = exc

sys.last\_value = val

sys.last\_traceback = tb

traceback.print\_exception(exc, val, tb)

def \_\_getattr\_\_(self, attr):

"Delegate attribute access to the interpreter object"

return getattr(self.tk, attr)

# Ideally, the classes Pack, Place and Grid disappear, the

# pack/place/grid methods are defined on the Widget class, and

# everybody uses w.pack\_whatever(...) instead of Pack.whatever(w,

# ...), with pack(), place() and grid() being short for

# pack\_configure(), place\_configure() and grid\_columnconfigure(), and

# forget() being short for pack\_forget(). As a practical matter, I'm

# afraid that there is too much code out there that may be using the

# Pack, Place or Grid class, so I leave them intact -- but only as

# backwards compatibility features. Also note that those methods that

# take a master as argument (e.g. pack\_propagate) have been moved to

# the Misc class (which now incorporates all methods common between

# toplevel and interior widgets). Again, for compatibility, these are

# copied into the Pack, Place or Grid class.

def Tcl(screenName=None, baseName=None, className='Tk', useTk=0):

return Tk(screenName, baseName, className, useTk)

class Pack:

"""Geometry manager Pack.

Base class to use the methods pack\_\* in every widget."""

def pack\_configure(self, cnf={}, \*\*kw):

"""Pack a widget in the parent widget. Use as options:

after=widget - pack it after you have packed widget

anchor=NSEW (or subset) - position widget according to

given direction

before=widget - pack it before you will pack widget

expand=bool - expand widget if parent size grows

fill=NONE or X or Y or BOTH - fill widget if widget grows

in=master - use master to contain this widget

in\_=master - see 'in' option description

ipadx=amount - add internal padding in x direction

ipady=amount - add internal padding in y direction

padx=amount - add padding in x direction

pady=amount - add padding in y direction

side=TOP or BOTTOM or LEFT or RIGHT - where to add this widget.

"""

self.tk.call(

('pack', 'configure', self.\_w)

+ self.\_options(cnf, kw))

pack = configure = config = pack\_configure

def pack\_forget(self):

"""Unmap this widget and do not use it for the packing order."""

self.tk.call('pack', 'forget', self.\_w)

forget = pack\_forget

def pack\_info(self):

"""Return information about the packing options

for this widget."""

d = \_splitdict(self.tk, self.tk.call('pack', 'info', self.\_w))

if 'in' in d:

d['in'] = self.nametowidget(d['in'])

return d

info = pack\_info

propagate = pack\_propagate = Misc.pack\_propagate

slaves = pack\_slaves = Misc.pack\_slaves

class Place:

"""Geometry manager Place.

Base class to use the methods place\_\* in every widget."""

def place\_configure(self, cnf={}, \*\*kw):

"""Place a widget in the parent widget. Use as options:

in=master - master relative to which the widget is placed

in\_=master - see 'in' option description

x=amount - locate anchor of this widget at position x of master

y=amount - locate anchor of this widget at position y of master

relx=amount - locate anchor of this widget between 0.0 and 1.0

relative to width of master (1.0 is right edge)

rely=amount - locate anchor of this widget between 0.0 and 1.0

relative to height of master (1.0 is bottom edge)

anchor=NSEW (or subset) - position anchor according to given direction

width=amount - width of this widget in pixel

height=amount - height of this widget in pixel

relwidth=amount - width of this widget between 0.0 and 1.0

relative to width of master (1.0 is the same width

as the master)

relheight=amount - height of this widget between 0.0 and 1.0

relative to height of master (1.0 is the same

height as the master)

bordermode="inside" or "outside" - whether to take border width of

master widget into account

"""

self.tk.call(

('place', 'configure', self.\_w)

+ self.\_options(cnf, kw))

place = configure = config = place\_configure

def place\_forget(self):

"""Unmap this widget."""

self.tk.call('place', 'forget', self.\_w)

forget = place\_forget

def place\_info(self):

"""Return information about the placing options

for this widget."""

d = \_splitdict(self.tk, self.tk.call('place', 'info', self.\_w))

if 'in' in d:

d['in'] = self.nametowidget(d['in'])

return d

info = place\_info

slaves = place\_slaves = Misc.place\_slaves

class Grid:

"""Geometry manager Grid.

Base class to use the methods grid\_\* in every widget."""

# Thanks to Masazumi Yoshikawa (yosikawa@isi.edu)

def grid\_configure(self, cnf={}, \*\*kw):

"""Position a widget in the parent widget in a grid. Use as options:

column=number - use cell identified with given column (starting with 0)

columnspan=number - this widget will span several columns

in=master - use master to contain this widget

in\_=master - see 'in' option description

ipadx=amount - add internal padding in x direction

ipady=amount - add internal padding in y direction

padx=amount - add padding in x direction

pady=amount - add padding in y direction

row=number - use cell identified with given row (starting with 0)

rowspan=number - this widget will span several rows

sticky=NSEW - if cell is larger on which sides will this

widget stick to the cell boundary

"""

self.tk.call(

('grid', 'configure', self.\_w)

+ self.\_options(cnf, kw))

grid = configure = config = grid\_configure

bbox = grid\_bbox = Misc.grid\_bbox

columnconfigure = grid\_columnconfigure = Misc.grid\_columnconfigure

def grid\_forget(self):

"""Unmap this widget."""

self.tk.call('grid', 'forget', self.\_w)

forget = grid\_forget

def grid\_remove(self):

"""Unmap this widget but remember the grid options."""

self.tk.call('grid', 'remove', self.\_w)

def grid\_info(self):

"""Return information about the options

for positioning this widget in a grid."""

d = \_splitdict(self.tk, self.tk.call('grid', 'info', self.\_w))

if 'in' in d:

d['in'] = self.nametowidget(d['in'])

return d

info = grid\_info

location = grid\_location = Misc.grid\_location

propagate = grid\_propagate = Misc.grid\_propagate

rowconfigure = grid\_rowconfigure = Misc.grid\_rowconfigure

size = grid\_size = Misc.grid\_size

slaves = grid\_slaves = Misc.grid\_slaves

class BaseWidget(Misc):

"""Internal class."""

def \_setup(self, master, cnf):

"""Internal function. Sets up information about children."""

if \_support\_default\_root:

global \_default\_root

if not master:

if not \_default\_root:

\_default\_root = Tk()

master = \_default\_root

self.master = master

self.tk = master.tk

name = None

if 'name' in cnf:

name = cnf['name']

del cnf['name']

if not name:

name = repr(id(self))

self.\_name = name

if master.\_w=='.':

self.\_w = '.' + name

else:

self.\_w = master.\_w + '.' + name

self.children = {}

if self.\_name in self.master.children:

self.master.children[self.\_name].destroy()

self.master.children[self.\_name] = self

def \_\_init\_\_(self, master, widgetName, cnf={}, kw={}, extra=()):

"""Construct a widget with the parent widget MASTER, a name WIDGETNAME

and appropriate options."""

if kw:

cnf = \_cnfmerge((cnf, kw))

self.widgetName = widgetName

BaseWidget.\_setup(self, master, cnf)

if self.\_tclCommands is None:

self.\_tclCommands = []

classes = []

for k in cnf.keys():

if type(k) is ClassType:

classes.append((k, cnf[k]))

del cnf[k]

self.tk.call(

(widgetName, self.\_w) + extra + self.\_options(cnf))

for k, v in classes:

k.configure(self, v)

def destroy(self):

"""Destroy this and all descendants widgets."""

for c in self.children.values(): c.destroy()

self.tk.call('destroy', self.\_w)

if self.\_name in self.master.children:

del self.master.children[self.\_name]

Misc.destroy(self)

def \_do(self, name, args=()):

# XXX Obsolete -- better use self.tk.call directly!

return self.tk.call((self.\_w, name) + args)

class Widget(BaseWidget, Pack, Place, Grid):

"""Internal class.

Base class for a widget which can be positioned with the geometry managers

Pack, Place or Grid."""

pass

class Toplevel(BaseWidget, Wm):

"""Toplevel widget, e.g. for dialogs."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a toplevel widget with the parent MASTER.

Valid resource names: background, bd, bg, borderwidth, class,

colormap, container, cursor, height, highlightbackground,

highlightcolor, highlightthickness, menu, relief, screen, takefocus,

use, visual, width."""

if kw:

cnf = \_cnfmerge((cnf, kw))

extra = ()

for wmkey in ['screen', 'class\_', 'class', 'visual',

'colormap']:

if wmkey in cnf:

val = cnf[wmkey]

# TBD: a hack needed because some keys

# are not valid as keyword arguments

if wmkey[-1] == '\_': opt = '-'+wmkey[:-1]

else: opt = '-'+wmkey

extra = extra + (opt, val)

del cnf[wmkey]

BaseWidget.\_\_init\_\_(self, master, 'toplevel', cnf, {}, extra)

root = self.\_root()

self.iconname(root.iconname())

self.title(root.title())

self.protocol("WM\_DELETE\_WINDOW", self.destroy)

class Button(Widget):

"""Button widget."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a button widget with the parent MASTER.

STANDARD OPTIONS

activebackground, activeforeground, anchor,

background, bitmap, borderwidth, cursor,

disabledforeground, font, foreground

highlightbackground, highlightcolor,

highlightthickness, image, justify,

padx, pady, relief, repeatdelay,

repeatinterval, takefocus, text,

textvariable, underline, wraplength

WIDGET-SPECIFIC OPTIONS

command, compound, default, height,

overrelief, state, width

"""

Widget.\_\_init\_\_(self, master, 'button', cnf, kw)

def tkButtonEnter(self, \*dummy):

self.tk.call('tkButtonEnter', self.\_w)

def tkButtonLeave(self, \*dummy):

self.tk.call('tkButtonLeave', self.\_w)

def tkButtonDown(self, \*dummy):

self.tk.call('tkButtonDown', self.\_w)

def tkButtonUp(self, \*dummy):

self.tk.call('tkButtonUp', self.\_w)

def tkButtonInvoke(self, \*dummy):

self.tk.call('tkButtonInvoke', self.\_w)

def flash(self):

"""Flash the button.

This is accomplished by redisplaying

the button several times, alternating between active and

normal colors. At the end of the flash the button is left

in the same normal/active state as when the command was

invoked. This command is ignored if the button's state is

disabled.

"""

self.tk.call(self.\_w, 'flash')

def invoke(self):

"""Invoke the command associated with the button.

The return value is the return value from the command,

or an empty string if there is no command associated with

the button. This command is ignored if the button's state

is disabled.

"""

return self.tk.call(self.\_w, 'invoke')

# Indices:

# XXX I don't like these -- take them away

def AtEnd():

return 'end'

def AtInsert(\*args):

s = 'insert'

for a in args:

if a: s = s + (' ' + a)

return s

def AtSelFirst():

return 'sel.first'

def AtSelLast():

return 'sel.last'

def At(x, y=None):

if y is None:

return '@%r' % (x,)

else:

return '@%r,%r' % (x, y)

class Canvas(Widget, XView, YView):

"""Canvas widget to display graphical elements like lines or text."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a canvas widget with the parent MASTER.

Valid resource names: background, bd, bg, borderwidth, closeenough,

confine, cursor, height, highlightbackground, highlightcolor,

highlightthickness, insertbackground, insertborderwidth,

insertofftime, insertontime, insertwidth, offset, relief,

scrollregion, selectbackground, selectborderwidth, selectforeground,

state, takefocus, width, xscrollcommand, xscrollincrement,

yscrollcommand, yscrollincrement."""

Widget.\_\_init\_\_(self, master, 'canvas', cnf, kw)

def addtag(self, \*args):

"""Internal function."""

self.tk.call((self.\_w, 'addtag') + args)

def addtag\_above(self, newtag, tagOrId):

"""Add tag NEWTAG to all items above TAGORID."""

self.addtag(newtag, 'above', tagOrId)

def addtag\_all(self, newtag):

"""Add tag NEWTAG to all items."""

self.addtag(newtag, 'all')

def addtag\_below(self, newtag, tagOrId):

"""Add tag NEWTAG to all items below TAGORID."""

self.addtag(newtag, 'below', tagOrId)

def addtag\_closest(self, newtag, x, y, halo=None, start=None):

"""Add tag NEWTAG to item which is closest to pixel at X, Y.

If several match take the top-most.

All items closer than HALO are considered overlapping (all are

closests). If START is specified the next below this tag is taken."""

self.addtag(newtag, 'closest', x, y, halo, start)

def addtag\_enclosed(self, newtag, x1, y1, x2, y2):

"""Add tag NEWTAG to all items in the rectangle defined

by X1,Y1,X2,Y2."""

self.addtag(newtag, 'enclosed', x1, y1, x2, y2)

def addtag\_overlapping(self, newtag, x1, y1, x2, y2):

"""Add tag NEWTAG to all items which overlap the rectangle

defined by X1,Y1,X2,Y2."""

self.addtag(newtag, 'overlapping', x1, y1, x2, y2)

def addtag\_withtag(self, newtag, tagOrId):

"""Add tag NEWTAG to all items with TAGORID."""

self.addtag(newtag, 'withtag', tagOrId)

def bbox(self, \*args):

"""Return a tuple of X1,Y1,X2,Y2 coordinates for a rectangle

which encloses all items with tags specified as arguments."""

return self.\_getints(

self.tk.call((self.\_w, 'bbox') + args)) or None

def tag\_unbind(self, tagOrId, sequence, funcid=None):

"""Unbind for all items with TAGORID for event SEQUENCE the

function identified with FUNCID."""

self.tk.call(self.\_w, 'bind', tagOrId, sequence, '')

if funcid:

self.deletecommand(funcid)

def tag\_bind(self, tagOrId, sequence=None, func=None, add=None):

"""Bind to all items with TAGORID at event SEQUENCE a call to function FUNC.

An additional boolean parameter ADD specifies whether FUNC will be

called additionally to the other bound function or whether it will

replace the previous function. See bind for the return value."""

return self.\_bind((self.\_w, 'bind', tagOrId),

sequence, func, add)

def canvasx(self, screenx, gridspacing=None):

"""Return the canvas x coordinate of pixel position SCREENX rounded

to nearest multiple of GRIDSPACING units."""

return getdouble(self.tk.call(

self.\_w, 'canvasx', screenx, gridspacing))

def canvasy(self, screeny, gridspacing=None):

"""Return the canvas y coordinate of pixel position SCREENY rounded

to nearest multiple of GRIDSPACING units."""

return getdouble(self.tk.call(

self.\_w, 'canvasy', screeny, gridspacing))

def coords(self, \*args):

"""Return a list of coordinates for the item given in ARGS."""

# XXX Should use \_flatten on args

return map(getdouble,

self.tk.splitlist(

self.tk.call((self.\_w, 'coords') + args)))

def \_create(self, itemType, args, kw): # Args: (val, val, ..., cnf={})

"""Internal function."""

args = \_flatten(args)

cnf = args[-1]

if type(cnf) in (DictionaryType, TupleType):

args = args[:-1]

else:

cnf = {}

return getint(self.tk.call(

self.\_w, 'create', itemType,

\*(args + self.\_options(cnf, kw))))

def create\_arc(self, \*args, \*\*kw):

"""Create arc shaped region with coordinates x1,y1,x2,y2."""

return self.\_create('arc', args, kw)

def create\_bitmap(self, \*args, \*\*kw):

"""Create bitmap with coordinates x1,y1."""

return self.\_create('bitmap', args, kw)

def create\_image(self, \*args, \*\*kw):

"""Create image item with coordinates x1,y1."""

return self.\_create('image', args, kw)

def create\_line(self, \*args, \*\*kw):

"""Create line with coordinates x1,y1,...,xn,yn."""

return self.\_create('line', args, kw)

def create\_oval(self, \*args, \*\*kw):

"""Create oval with coordinates x1,y1,x2,y2."""

return self.\_create('oval', args, kw)

def create\_polygon(self, \*args, \*\*kw):

"""Create polygon with coordinates x1,y1,...,xn,yn."""

return self.\_create('polygon', args, kw)

def create\_rectangle(self, \*args, \*\*kw):

"""Create rectangle with coordinates x1,y1,x2,y2."""

return self.\_create('rectangle', args, kw)

def create\_text(self, \*args, \*\*kw):

"""Create text with coordinates x1,y1."""

return self.\_create('text', args, kw)

def create\_window(self, \*args, \*\*kw):

"""Create window with coordinates x1,y1,x2,y2."""

return self.\_create('window', args, kw)

def dchars(self, \*args):

"""Delete characters of text items identified by tag or id in ARGS (possibly

several times) from FIRST to LAST character (including)."""

self.tk.call((self.\_w, 'dchars') + args)

def delete(self, \*args):

"""Delete items identified by all tag or ids contained in ARGS."""

self.tk.call((self.\_w, 'delete') + args)

def dtag(self, \*args):

"""Delete tag or id given as last arguments in ARGS from items

identified by first argument in ARGS."""

self.tk.call((self.\_w, 'dtag') + args)

def find(self, \*args):

"""Internal function."""

return self.\_getints(

self.tk.call((self.\_w, 'find') + args)) or ()

def find\_above(self, tagOrId):

"""Return items above TAGORID."""

return self.find('above', tagOrId)

def find\_all(self):

"""Return all items."""

return self.find('all')

def find\_below(self, tagOrId):

"""Return all items below TAGORID."""

return self.find('below', tagOrId)

def find\_closest(self, x, y, halo=None, start=None):

"""Return item which is closest to pixel at X, Y.

If several match take the top-most.

All items closer than HALO are considered overlapping (all are

closests). If START is specified the next below this tag is taken."""

return self.find('closest', x, y, halo, start)

def find\_enclosed(self, x1, y1, x2, y2):

"""Return all items in rectangle defined

by X1,Y1,X2,Y2."""

return self.find('enclosed', x1, y1, x2, y2)

def find\_overlapping(self, x1, y1, x2, y2):

"""Return all items which overlap the rectangle

defined by X1,Y1,X2,Y2."""

return self.find('overlapping', x1, y1, x2, y2)

def find\_withtag(self, tagOrId):

"""Return all items with TAGORID."""

return self.find('withtag', tagOrId)

def focus(self, \*args):

"""Set focus to the first item specified in ARGS."""

return self.tk.call((self.\_w, 'focus') + args)

def gettags(self, \*args):

"""Return tags associated with the first item specified in ARGS."""

return self.tk.splitlist(

self.tk.call((self.\_w, 'gettags') + args))

def icursor(self, \*args):

"""Set cursor at position POS in the item identified by TAGORID.

In ARGS TAGORID must be first."""

self.tk.call((self.\_w, 'icursor') + args)

def index(self, \*args):

"""Return position of cursor as integer in item specified in ARGS."""

return getint(self.tk.call((self.\_w, 'index') + args))

def insert(self, \*args):

"""Insert TEXT in item TAGORID at position POS. ARGS must

be TAGORID POS TEXT."""

self.tk.call((self.\_w, 'insert') + args)

def itemcget(self, tagOrId, option):

"""Return the resource value for an OPTION for item TAGORID."""

return self.tk.call(

(self.\_w, 'itemcget') + (tagOrId, '-'+option))

def itemconfigure(self, tagOrId, cnf=None, \*\*kw):

"""Configure resources of an item TAGORID.

The values for resources are specified as keyword

arguments. To get an overview about

the allowed keyword arguments call the method without arguments.

"""

return self.\_configure(('itemconfigure', tagOrId), cnf, kw)

itemconfig = itemconfigure

# lower, tkraise/lift hide Misc.lower, Misc.tkraise/lift,

# so the preferred name for them is tag\_lower, tag\_raise

# (similar to tag\_bind, and similar to the Text widget);

# unfortunately can't delete the old ones yet (maybe in 1.6)

def tag\_lower(self, \*args):

"""Lower an item TAGORID given in ARGS

(optional below another item)."""

self.tk.call((self.\_w, 'lower') + args)

lower = tag\_lower

def move(self, \*args):

"""Move an item TAGORID given in ARGS."""

self.tk.call((self.\_w, 'move') + args)

def postscript(self, cnf={}, \*\*kw):

"""Print the contents of the canvas to a postscript

file. Valid options: colormap, colormode, file, fontmap,

height, pageanchor, pageheight, pagewidth, pagex, pagey,

rotate, witdh, x, y."""

return self.tk.call((self.\_w, 'postscript') +

self.\_options(cnf, kw))

def tag\_raise(self, \*args):

"""Raise an item TAGORID given in ARGS

(optional above another item)."""

self.tk.call((self.\_w, 'raise') + args)

lift = tkraise = tag\_raise

def scale(self, \*args):

"""Scale item TAGORID with XORIGIN, YORIGIN, XSCALE, YSCALE."""

self.tk.call((self.\_w, 'scale') + args)

def scan\_mark(self, x, y):

"""Remember the current X, Y coordinates."""

self.tk.call(self.\_w, 'scan', 'mark', x, y)

def scan\_dragto(self, x, y, gain=10):

"""Adjust the view of the canvas to GAIN times the

difference between X and Y and the coordinates given in

scan\_mark."""

self.tk.call(self.\_w, 'scan', 'dragto', x, y, gain)

def select\_adjust(self, tagOrId, index):

"""Adjust the end of the selection near the cursor of an item TAGORID to index."""

self.tk.call(self.\_w, 'select', 'adjust', tagOrId, index)

def select\_clear(self):

"""Clear the selection if it is in this widget."""

self.tk.call(self.\_w, 'select', 'clear')

def select\_from(self, tagOrId, index):

"""Set the fixed end of a selection in item TAGORID to INDEX."""

self.tk.call(self.\_w, 'select', 'from', tagOrId, index)

def select\_item(self):

"""Return the item which has the selection."""

return self.tk.call(self.\_w, 'select', 'item') or None

def select\_to(self, tagOrId, index):

"""Set the variable end of a selection in item TAGORID to INDEX."""

self.tk.call(self.\_w, 'select', 'to', tagOrId, index)

def type(self, tagOrId):

"""Return the type of the item TAGORID."""

return self.tk.call(self.\_w, 'type', tagOrId) or None

class Checkbutton(Widget):

"""Checkbutton widget which is either in on- or off-state."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a checkbutton widget with the parent MASTER.

Valid resource names: activebackground, activeforeground, anchor,

background, bd, bg, bitmap, borderwidth, command, cursor,

disabledforeground, fg, font, foreground, height,

highlightbackground, highlightcolor, highlightthickness, image,

indicatoron, justify, offvalue, onvalue, padx, pady, relief,

selectcolor, selectimage, state, takefocus, text, textvariable,

underline, variable, width, wraplength."""

Widget.\_\_init\_\_(self, master, 'checkbutton', cnf, kw)

def deselect(self):

"""Put the button in off-state."""

self.tk.call(self.\_w, 'deselect')

def flash(self):

"""Flash the button."""

self.tk.call(self.\_w, 'flash')

def invoke(self):

"""Toggle the button and invoke a command if given as resource."""

return self.tk.call(self.\_w, 'invoke')

def select(self):

"""Put the button in on-state."""

self.tk.call(self.\_w, 'select')

def toggle(self):

"""Toggle the button."""

self.tk.call(self.\_w, 'toggle')

class Entry(Widget, XView):

"""Entry widget which allows to display simple text."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct an entry widget with the parent MASTER.

Valid resource names: background, bd, bg, borderwidth, cursor,

exportselection, fg, font, foreground, highlightbackground,

highlightcolor, highlightthickness, insertbackground,

insertborderwidth, insertofftime, insertontime, insertwidth,

invalidcommand, invcmd, justify, relief, selectbackground,

selectborderwidth, selectforeground, show, state, takefocus,

textvariable, validate, validatecommand, vcmd, width,

xscrollcommand."""

Widget.\_\_init\_\_(self, master, 'entry', cnf, kw)

def delete(self, first, last=None):

"""Delete text from FIRST to LAST (not included)."""

self.tk.call(self.\_w, 'delete', first, last)

def get(self):

"""Return the text."""

return self.tk.call(self.\_w, 'get')

def icursor(self, index):

"""Insert cursor at INDEX."""

self.tk.call(self.\_w, 'icursor', index)

def index(self, index):

"""Return position of cursor."""

return getint(self.tk.call(

self.\_w, 'index', index))

def insert(self, index, string):

"""Insert STRING at INDEX."""

self.tk.call(self.\_w, 'insert', index, string)

def scan\_mark(self, x):

"""Remember the current X, Y coordinates."""

self.tk.call(self.\_w, 'scan', 'mark', x)

def scan\_dragto(self, x):

"""Adjust the view of the canvas to 10 times the

difference between X and Y and the coordinates given in

scan\_mark."""

self.tk.call(self.\_w, 'scan', 'dragto', x)

def selection\_adjust(self, index):

"""Adjust the end of the selection near the cursor to INDEX."""

self.tk.call(self.\_w, 'selection', 'adjust', index)

select\_adjust = selection\_adjust

def selection\_clear(self):

"""Clear the selection if it is in this widget."""

self.tk.call(self.\_w, 'selection', 'clear')

select\_clear = selection\_clear

def selection\_from(self, index):

"""Set the fixed end of a selection to INDEX."""

self.tk.call(self.\_w, 'selection', 'from', index)

select\_from = selection\_from

def selection\_present(self):

"""Return True if there are characters selected in the entry, False

otherwise."""

return self.tk.getboolean(

self.tk.call(self.\_w, 'selection', 'present'))

select\_present = selection\_present

def selection\_range(self, start, end):

"""Set the selection from START to END (not included)."""

self.tk.call(self.\_w, 'selection', 'range', start, end)

select\_range = selection\_range

def selection\_to(self, index):

"""Set the variable end of a selection to INDEX."""

self.tk.call(self.\_w, 'selection', 'to', index)

select\_to = selection\_to

class Frame(Widget):

"""Frame widget which may contain other widgets and can have a 3D border."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a frame widget with the parent MASTER.

Valid resource names: background, bd, bg, borderwidth, class,

colormap, container, cursor, height, highlightbackground,

highlightcolor, highlightthickness, relief, takefocus, visual, width."""

cnf = \_cnfmerge((cnf, kw))

extra = ()

if 'class\_' in cnf:

extra = ('-class', cnf['class\_'])

del cnf['class\_']

elif 'class' in cnf:

extra = ('-class', cnf['class'])

del cnf['class']

Widget.\_\_init\_\_(self, master, 'frame', cnf, {}, extra)

class Label(Widget):

"""Label widget which can display text and bitmaps."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a label widget with the parent MASTER.

STANDARD OPTIONS

activebackground, activeforeground, anchor,

background, bitmap, borderwidth, cursor,

disabledforeground, font, foreground,

highlightbackground, highlightcolor,

highlightthickness, image, justify,

padx, pady, relief, takefocus, text,

textvariable, underline, wraplength

WIDGET-SPECIFIC OPTIONS

height, state, width

"""

Widget.\_\_init\_\_(self, master, 'label', cnf, kw)

class Listbox(Widget, XView, YView):

"""Listbox widget which can display a list of strings."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a listbox widget with the parent MASTER.

Valid resource names: background, bd, bg, borderwidth, cursor,

exportselection, fg, font, foreground, height, highlightbackground,

highlightcolor, highlightthickness, relief, selectbackground,

selectborderwidth, selectforeground, selectmode, setgrid, takefocus,

width, xscrollcommand, yscrollcommand, listvariable."""

Widget.\_\_init\_\_(self, master, 'listbox', cnf, kw)

def activate(self, index):

"""Activate item identified by INDEX."""

self.tk.call(self.\_w, 'activate', index)

def bbox(self, index):

"""Return a tuple of X1,Y1,X2,Y2 coordinates for a rectangle

which encloses the item identified by the given index."""

return self.\_getints(self.tk.call(self.\_w, 'bbox', index)) or None

def curselection(self):

"""Return the indices of currently selected item."""

return self.\_getints(self.tk.call(self.\_w, 'curselection')) or ()

def delete(self, first, last=None):

"""Delete items from FIRST to LAST (included)."""

self.tk.call(self.\_w, 'delete', first, last)

def get(self, first, last=None):

"""Get list of items from FIRST to LAST (included)."""

if last is not None:

return self.tk.splitlist(self.tk.call(

self.\_w, 'get', first, last))

else:

return self.tk.call(self.\_w, 'get', first)

def index(self, index):

"""Return index of item identified with INDEX."""

i = self.tk.call(self.\_w, 'index', index)

if i == 'none': return None

return getint(i)

def insert(self, index, \*elements):

"""Insert ELEMENTS at INDEX."""

self.tk.call((self.\_w, 'insert', index) + elements)

def nearest(self, y):

"""Get index of item which is nearest to y coordinate Y."""

return getint(self.tk.call(

self.\_w, 'nearest', y))

def scan\_mark(self, x, y):

"""Remember the current X, Y coordinates."""

self.tk.call(self.\_w, 'scan', 'mark', x, y)

def scan\_dragto(self, x, y):

"""Adjust the view of the listbox to 10 times the

difference between X and Y and the coordinates given in

scan\_mark."""

self.tk.call(self.\_w, 'scan', 'dragto', x, y)

def see(self, index):

"""Scroll such that INDEX is visible."""

self.tk.call(self.\_w, 'see', index)

def selection\_anchor(self, index):

"""Set the fixed end oft the selection to INDEX."""

self.tk.call(self.\_w, 'selection', 'anchor', index)

select\_anchor = selection\_anchor

def selection\_clear(self, first, last=None):

"""Clear the selection from FIRST to LAST (included)."""

self.tk.call(self.\_w,

'selection', 'clear', first, last)

select\_clear = selection\_clear

def selection\_includes(self, index):

"""Return 1 if INDEX is part of the selection."""

return self.tk.getboolean(self.tk.call(

self.\_w, 'selection', 'includes', index))

select\_includes = selection\_includes

def selection\_set(self, first, last=None):

"""Set the selection from FIRST to LAST (included) without

changing the currently selected elements."""

self.tk.call(self.\_w, 'selection', 'set', first, last)

select\_set = selection\_set

def size(self):

"""Return the number of elements in the listbox."""

return getint(self.tk.call(self.\_w, 'size'))

def itemcget(self, index, option):

"""Return the resource value for an ITEM and an OPTION."""

return self.tk.call(

(self.\_w, 'itemcget') + (index, '-'+option))

def itemconfigure(self, index, cnf=None, \*\*kw):

"""Configure resources of an ITEM.

The values for resources are specified as keyword arguments.

To get an overview about the allowed keyword arguments

call the method without arguments.

Valid resource names: background, bg, foreground, fg,

selectbackground, selectforeground."""

return self.\_configure(('itemconfigure', index), cnf, kw)

itemconfig = itemconfigure

class Menu(Widget):

"""Menu widget which allows to display menu bars, pull-down menus and pop-up menus."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct menu widget with the parent MASTER.

Valid resource names: activebackground, activeborderwidth,

activeforeground, background, bd, bg, borderwidth, cursor,

disabledforeground, fg, font, foreground, postcommand, relief,

selectcolor, takefocus, tearoff, tearoffcommand, title, type."""

Widget.\_\_init\_\_(self, master, 'menu', cnf, kw)

def tk\_bindForTraversal(self):

# obsolete since Tk 4.0

import warnings

warnings.warn('tk\_bindForTraversal() does nothing and '

'will be removed in 3.6',

DeprecationWarning, stacklevel=2)

def tk\_mbPost(self):

self.tk.call('tk\_mbPost', self.\_w)

def tk\_mbUnpost(self):

self.tk.call('tk\_mbUnpost')

def tk\_traverseToMenu(self, char):

self.tk.call('tk\_traverseToMenu', self.\_w, char)

def tk\_traverseWithinMenu(self, char):

self.tk.call('tk\_traverseWithinMenu', self.\_w, char)

def tk\_getMenuButtons(self):

return self.tk.call('tk\_getMenuButtons', self.\_w)

def tk\_nextMenu(self, count):

self.tk.call('tk\_nextMenu', count)

def tk\_nextMenuEntry(self, count):

self.tk.call('tk\_nextMenuEntry', count)

def tk\_invokeMenu(self):

self.tk.call('tk\_invokeMenu', self.\_w)

def tk\_firstMenu(self):

self.tk.call('tk\_firstMenu', self.\_w)

def tk\_mbButtonDown(self):

self.tk.call('tk\_mbButtonDown', self.\_w)

def tk\_popup(self, x, y, entry=""):

"""Post the menu at position X,Y with entry ENTRY."""

self.tk.call('tk\_popup', self.\_w, x, y, entry)

def activate(self, index):

"""Activate entry at INDEX."""

self.tk.call(self.\_w, 'activate', index)

def add(self, itemType, cnf={}, \*\*kw):

"""Internal function."""

self.tk.call((self.\_w, 'add', itemType) +

self.\_options(cnf, kw))

def add\_cascade(self, cnf={}, \*\*kw):

"""Add hierarchical menu item."""

self.add('cascade', cnf or kw)

def add\_checkbutton(self, cnf={}, \*\*kw):

"""Add checkbutton menu item."""

self.add('checkbutton', cnf or kw)

def add\_command(self, cnf={}, \*\*kw):

"""Add command menu item."""

self.add('command', cnf or kw)

def add\_radiobutton(self, cnf={}, \*\*kw):

"""Addd radio menu item."""

self.add('radiobutton', cnf or kw)

def add\_separator(self, cnf={}, \*\*kw):

"""Add separator."""

self.add('separator', cnf or kw)

def insert(self, index, itemType, cnf={}, \*\*kw):

"""Internal function."""

self.tk.call((self.\_w, 'insert', index, itemType) +

self.\_options(cnf, kw))

def insert\_cascade(self, index, cnf={}, \*\*kw):

"""Add hierarchical menu item at INDEX."""

self.insert(index, 'cascade', cnf or kw)

def insert\_checkbutton(self, index, cnf={}, \*\*kw):

"""Add checkbutton menu item at INDEX."""

self.insert(index, 'checkbutton', cnf or kw)

def insert\_command(self, index, cnf={}, \*\*kw):

"""Add command menu item at INDEX."""

self.insert(index, 'command', cnf or kw)

def insert\_radiobutton(self, index, cnf={}, \*\*kw):

"""Addd radio menu item at INDEX."""

self.insert(index, 'radiobutton', cnf or kw)

def insert\_separator(self, index, cnf={}, \*\*kw):

"""Add separator at INDEX."""

self.insert(index, 'separator', cnf or kw)

def delete(self, index1, index2=None):

"""Delete menu items between INDEX1 and INDEX2 (included)."""

if index2 is None:

index2 = index1

num\_index1, num\_index2 = self.index(index1), self.index(index2)

if (num\_index1 is None) or (num\_index2 is None):

num\_index1, num\_index2 = 0, -1

for i in range(num\_index1, num\_index2 + 1):

if 'command' in self.entryconfig(i):

c = str(self.entrycget(i, 'command'))

if c:

self.deletecommand(c)

self.tk.call(self.\_w, 'delete', index1, index2)

def entrycget(self, index, option):

"""Return the resource value of an menu item for OPTION at INDEX."""

return self.tk.call(self.\_w, 'entrycget', index, '-' + option)

def entryconfigure(self, index, cnf=None, \*\*kw):

"""Configure a menu item at INDEX."""

return self.\_configure(('entryconfigure', index), cnf, kw)

entryconfig = entryconfigure

def index(self, index):

"""Return the index of a menu item identified by INDEX."""

i = self.tk.call(self.\_w, 'index', index)

if i == 'none': return None

return getint(i)

def invoke(self, index):

"""Invoke a menu item identified by INDEX and execute

the associated command."""

return self.tk.call(self.\_w, 'invoke', index)

def post(self, x, y):

"""Display a menu at position X,Y."""

self.tk.call(self.\_w, 'post', x, y)

def type(self, index):

"""Return the type of the menu item at INDEX."""

return self.tk.call(self.\_w, 'type', index)

def unpost(self):

"""Unmap a menu."""

self.tk.call(self.\_w, 'unpost')

def yposition(self, index):

"""Return the y-position of the topmost pixel of the menu item at INDEX."""

return getint(self.tk.call(

self.\_w, 'yposition', index))

class Menubutton(Widget):

"""Menubutton widget, obsolete since Tk8.0."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

Widget.\_\_init\_\_(self, master, 'menubutton', cnf, kw)

class Message(Widget):

"""Message widget to display multiline text. Obsolete since Label does it too."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

Widget.\_\_init\_\_(self, master, 'message', cnf, kw)

class Radiobutton(Widget):

"""Radiobutton widget which shows only one of several buttons in on-state."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a radiobutton widget with the parent MASTER.

Valid resource names: activebackground, activeforeground, anchor,

background, bd, bg, bitmap, borderwidth, command, cursor,

disabledforeground, fg, font, foreground, height,

highlightbackground, highlightcolor, highlightthickness, image,

indicatoron, justify, padx, pady, relief, selectcolor, selectimage,

state, takefocus, text, textvariable, underline, value, variable,

width, wraplength."""

Widget.\_\_init\_\_(self, master, 'radiobutton', cnf, kw)

def deselect(self):

"""Put the button in off-state."""

self.tk.call(self.\_w, 'deselect')

def flash(self):

"""Flash the button."""

self.tk.call(self.\_w, 'flash')

def invoke(self):

"""Toggle the button and invoke a command if given as resource."""

return self.tk.call(self.\_w, 'invoke')

def select(self):

"""Put the button in on-state."""

self.tk.call(self.\_w, 'select')

class Scale(Widget):

"""Scale widget which can display a numerical scale."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a scale widget with the parent MASTER.

Valid resource names: activebackground, background, bigincrement, bd,

bg, borderwidth, command, cursor, digits, fg, font, foreground, from,

highlightbackground, highlightcolor, highlightthickness, label,

length, orient, relief, repeatdelay, repeatinterval, resolution,

showvalue, sliderlength, sliderrelief, state, takefocus,

tickinterval, to, troughcolor, variable, width."""

Widget.\_\_init\_\_(self, master, 'scale', cnf, kw)

def get(self):

"""Get the current value as integer or float."""

value = self.tk.call(self.\_w, 'get')

try:

return getint(value)

except ValueError:

return getdouble(value)

def set(self, value):

"""Set the value to VALUE."""

self.tk.call(self.\_w, 'set', value)

def coords(self, value=None):

"""Return a tuple (X,Y) of the point along the centerline of the

trough that corresponds to VALUE or the current value if None is

given."""

return self.\_getints(self.tk.call(self.\_w, 'coords', value))

def identify(self, x, y):

"""Return where the point X,Y lies. Valid return values are "slider",

"though1" and "though2"."""

return self.tk.call(self.\_w, 'identify', x, y)

class Scrollbar(Widget):

"""Scrollbar widget which displays a slider at a certain position."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a scrollbar widget with the parent MASTER.

Valid resource names: activebackground, activerelief,

background, bd, bg, borderwidth, command, cursor,

elementborderwidth, highlightbackground,

highlightcolor, highlightthickness, jump, orient,

relief, repeatdelay, repeatinterval, takefocus,

troughcolor, width."""

Widget.\_\_init\_\_(self, master, 'scrollbar', cnf, kw)

def activate(self, index):

"""Display the element at INDEX with activebackground and activerelief.

INDEX can be "arrow1","slider" or "arrow2"."""

self.tk.call(self.\_w, 'activate', index)

def delta(self, deltax, deltay):

"""Return the fractional change of the scrollbar setting if it

would be moved by DELTAX or DELTAY pixels."""

return getdouble(

self.tk.call(self.\_w, 'delta', deltax, deltay))

def fraction(self, x, y):

"""Return the fractional value which corresponds to a slider

position of X,Y."""

return getdouble(self.tk.call(self.\_w, 'fraction', x, y))

def identify(self, x, y):

"""Return the element under position X,Y as one of

"arrow1","slider","arrow2" or ""."""

return self.tk.call(self.\_w, 'identify', x, y)

def get(self):

"""Return the current fractional values (upper and lower end)

of the slider position."""

return self.\_getdoubles(self.tk.call(self.\_w, 'get'))

def set(self, \*args):

"""Set the fractional values of the slider position (upper and

lower ends as value between 0 and 1)."""

self.tk.call((self.\_w, 'set') + args)

class Text(Widget, XView, YView):

"""Text widget which can display text in various forms."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a text widget with the parent MASTER.

STANDARD OPTIONS

background, borderwidth, cursor,

exportselection, font, foreground,

highlightbackground, highlightcolor,

highlightthickness, insertbackground,

insertborderwidth, insertofftime,

insertontime, insertwidth, padx, pady,

relief, selectbackground,

selectborderwidth, selectforeground,

setgrid, takefocus,

xscrollcommand, yscrollcommand,

WIDGET-SPECIFIC OPTIONS

autoseparators, height, maxundo,

spacing1, spacing2, spacing3,

state, tabs, undo, width, wrap,

"""

Widget.\_\_init\_\_(self, master, 'text', cnf, kw)

def bbox(self, \*args):

"""Return a tuple of (x,y,width,height) which gives the bounding

box of the visible part of the character at the index in ARGS."""

return self.\_getints(

self.tk.call((self.\_w, 'bbox') + args)) or None

def tk\_textSelectTo(self, index):

self.tk.call('tk\_textSelectTo', self.\_w, index)

def tk\_textBackspace(self):

self.tk.call('tk\_textBackspace', self.\_w)

def tk\_textIndexCloser(self, a, b, c):

self.tk.call('tk\_textIndexCloser', self.\_w, a, b, c)

def tk\_textResetAnchor(self, index):

self.tk.call('tk\_textResetAnchor', self.\_w, index)

def compare(self, index1, op, index2):

"""Return whether between index INDEX1 and index INDEX2 the

relation OP is satisfied. OP is one of <, <=, ==, >=, >, or !=."""

return self.tk.getboolean(self.tk.call(

self.\_w, 'compare', index1, op, index2))

def debug(self, boolean=None):

"""Turn on the internal consistency checks of the B-Tree inside the text

widget according to BOOLEAN."""

if boolean is None:

return self.tk.getboolean(self.tk.call(self.\_w, 'debug'))

self.tk.call(self.\_w, 'debug', boolean)

def delete(self, index1, index2=None):

"""Delete the characters between INDEX1 and INDEX2 (not included)."""

self.tk.call(self.\_w, 'delete', index1, index2)

def dlineinfo(self, index):

"""Return tuple (x,y,width,height,baseline) giving the bounding box

and baseline position of the visible part of the line containing

the character at INDEX."""

return self.\_getints(self.tk.call(self.\_w, 'dlineinfo', index))

def dump(self, index1, index2=None, command=None, \*\*kw):

"""Return the contents of the widget between index1 and index2.

The type of contents returned in filtered based on the keyword

parameters; if 'all', 'image', 'mark', 'tag', 'text', or 'window' are

given and true, then the corresponding items are returned. The result

is a list of triples of the form (key, value, index). If none of the

keywords are true then 'all' is used by default.

If the 'command' argument is given, it is called once for each element

of the list of triples, with the values of each triple serving as the

arguments to the function. In this case the list is not returned."""

args = []

func\_name = None

result = None

if not command:

# Never call the dump command without the -command flag, since the

# output could involve Tcl quoting and would be a pain to parse

# right. Instead just set the command to build a list of triples

# as if we had done the parsing.

result = []

def append\_triple(key, value, index, result=result):

result.append((key, value, index))

command = append\_triple

try:

if not isinstance(command, str):

func\_name = command = self.\_register(command)

args += ["-command", command]

for key in kw:

if kw[key]: args.append("-" + key)

args.append(index1)

if index2:

args.append(index2)

self.tk.call(self.\_w, "dump", \*args)

return result

finally:

if func\_name:

self.deletecommand(func\_name)

## new in tk8.4

def edit(self, \*args):

"""Internal method

This method controls the undo mechanism and

the modified flag. The exact behavior of the

command depends on the option argument that

follows the edit argument. The following forms

of the command are currently supported:

edit\_modified, edit\_redo, edit\_reset, edit\_separator

and edit\_undo

"""

return self.tk.call(self.\_w, 'edit', \*args)

def edit\_modified(self, arg=None):

"""Get or Set the modified flag

If arg is not specified, returns the modified

flag of the widget. The insert, delete, edit undo and

edit redo commands or the user can set or clear the

modified flag. If boolean is specified, sets the

modified flag of the widget to arg.

"""

return self.edit("modified", arg)

def edit\_redo(self):

"""Redo the last undone edit

When the undo option is true, reapplies the last

undone edits provided no other edits were done since

then. Generates an error when the redo stack is empty.

Does nothing when the undo option is false.

"""

return self.edit("redo")

def edit\_reset(self):

"""Clears the undo and redo stacks

"""

return self.edit("reset")

def edit\_separator(self):

"""Inserts a separator (boundary) on the undo stack.

Does nothing when the undo option is false

"""

return self.edit("separator")

def edit\_undo(self):

"""Undoes the last edit action

If the undo option is true. An edit action is defined

as all the insert and delete commands that are recorded

on the undo stack in between two separators. Generates

an error when the undo stack is empty. Does nothing

when the undo option is false

"""

return self.edit("undo")

def get(self, index1, index2=None):

"""Return the text from INDEX1 to INDEX2 (not included)."""

return self.tk.call(self.\_w, 'get', index1, index2)

# (Image commands are new in 8.0)

def image\_cget(self, index, option):

"""Return the value of OPTION of an embedded image at INDEX."""

if option[:1] != "-":

option = "-" + option

if option[-1:] == "\_":

option = option[:-1]

return self.tk.call(self.\_w, "image", "cget", index, option)

def image\_configure(self, index, cnf=None, \*\*kw):

"""Configure an embedded image at INDEX."""

return self.\_configure(('image', 'configure', index), cnf, kw)

def image\_create(self, index, cnf={}, \*\*kw):

"""Create an embedded image at INDEX."""

return self.tk.call(

self.\_w, "image", "create", index,

\*self.\_options(cnf, kw))

def image\_names(self):

"""Return all names of embedded images in this widget."""

return self.tk.call(self.\_w, "image", "names")

def index(self, index):

"""Return the index in the form line.char for INDEX."""

return str(self.tk.call(self.\_w, 'index', index))

def insert(self, index, chars, \*args):

"""Insert CHARS before the characters at INDEX. An additional

tag can be given in ARGS. Additional CHARS and tags can follow in ARGS."""

self.tk.call((self.\_w, 'insert', index, chars) + args)

def mark\_gravity(self, markName, direction=None):

"""Change the gravity of a mark MARKNAME to DIRECTION (LEFT or RIGHT).

Return the current value if None is given for DIRECTION."""

return self.tk.call(

(self.\_w, 'mark', 'gravity', markName, direction))

def mark\_names(self):

"""Return all mark names."""

return self.tk.splitlist(self.tk.call(

self.\_w, 'mark', 'names'))

def mark\_set(self, markName, index):

"""Set mark MARKNAME before the character at INDEX."""

self.tk.call(self.\_w, 'mark', 'set', markName, index)

def mark\_unset(self, \*markNames):

"""Delete all marks in MARKNAMES."""

self.tk.call((self.\_w, 'mark', 'unset') + markNames)

def mark\_next(self, index):

"""Return the name of the next mark after INDEX."""

return self.tk.call(self.\_w, 'mark', 'next', index) or None

def mark\_previous(self, index):

"""Return the name of the previous mark before INDEX."""

return self.tk.call(self.\_w, 'mark', 'previous', index) or None

def scan\_mark(self, x, y):

"""Remember the current X, Y coordinates."""

self.tk.call(self.\_w, 'scan', 'mark', x, y)

def scan\_dragto(self, x, y):

"""Adjust the view of the text to 10 times the

difference between X and Y and the coordinates given in

scan\_mark."""

self.tk.call(self.\_w, 'scan', 'dragto', x, y)

def search(self, pattern, index, stopindex=None,

forwards=None, backwards=None, exact=None,

regexp=None, nocase=None, count=None, elide=None):

"""Search PATTERN beginning from INDEX until STOPINDEX.

Return the index of the first character of a match or an

empty string."""

args = [self.\_w, 'search']

if forwards: args.append('-forwards')

if backwards: args.append('-backwards')

if exact: args.append('-exact')

if regexp: args.append('-regexp')

if nocase: args.append('-nocase')

if elide: args.append('-elide')

if count: args.append('-count'); args.append(count)

if pattern and pattern[0] == '-': args.append('--')

args.append(pattern)

args.append(index)

if stopindex: args.append(stopindex)

return str(self.tk.call(tuple(args)))

def see(self, index):

"""Scroll such that the character at INDEX is visible."""

self.tk.call(self.\_w, 'see', index)

def tag\_add(self, tagName, index1, \*args):

"""Add tag TAGNAME to all characters between INDEX1 and index2 in ARGS.

Additional pairs of indices may follow in ARGS."""

self.tk.call(

(self.\_w, 'tag', 'add', tagName, index1) + args)

def tag\_unbind(self, tagName, sequence, funcid=None):

"""Unbind for all characters with TAGNAME for event SEQUENCE the

function identified with FUNCID."""

self.tk.call(self.\_w, 'tag', 'bind', tagName, sequence, '')

if funcid:

self.deletecommand(funcid)

def tag\_bind(self, tagName, sequence, func, add=None):

"""Bind to all characters with TAGNAME at event SEQUENCE a call to function FUNC.

An additional boolean parameter ADD specifies whether FUNC will be

called additionally to the other bound function or whether it will

replace the previous function. See bind for the return value."""

return self.\_bind((self.\_w, 'tag', 'bind', tagName),

sequence, func, add)

def tag\_cget(self, tagName, option):

"""Return the value of OPTION for tag TAGNAME."""

if option[:1] != '-':

option = '-' + option

if option[-1:] == '\_':

option = option[:-1]

return self.tk.call(self.\_w, 'tag', 'cget', tagName, option)

def tag\_configure(self, tagName, cnf=None, \*\*kw):

"""Configure a tag TAGNAME."""

return self.\_configure(('tag', 'configure', tagName), cnf, kw)

tag\_config = tag\_configure

def tag\_delete(self, \*tagNames):

"""Delete all tags in TAGNAMES."""

self.tk.call((self.\_w, 'tag', 'delete') + tagNames)

def tag\_lower(self, tagName, belowThis=None):

"""Change the priority of tag TAGNAME such that it is lower

than the priority of BELOWTHIS."""

self.tk.call(self.\_w, 'tag', 'lower', tagName, belowThis)

def tag\_names(self, index=None):

"""Return a list of all tag names."""

return self.tk.splitlist(

self.tk.call(self.\_w, 'tag', 'names', index))

def tag\_nextrange(self, tagName, index1, index2=None):

"""Return a list of start and end index for the first sequence of

characters between INDEX1 and INDEX2 which all have tag TAGNAME.

The text is searched forward from INDEX1."""

return self.tk.splitlist(self.tk.call(

self.\_w, 'tag', 'nextrange', tagName, index1, index2))

def tag\_prevrange(self, tagName, index1, index2=None):

"""Return a list of start and end index for the first sequence of

characters between INDEX1 and INDEX2 which all have tag TAGNAME.

The text is searched backwards from INDEX1."""

return self.tk.splitlist(self.tk.call(

self.\_w, 'tag', 'prevrange', tagName, index1, index2))

def tag\_raise(self, tagName, aboveThis=None):

"""Change the priority of tag TAGNAME such that it is higher

than the priority of ABOVETHIS."""

self.tk.call(

self.\_w, 'tag', 'raise', tagName, aboveThis)

def tag\_ranges(self, tagName):

"""Return a list of ranges of text which have tag TAGNAME."""

return self.tk.splitlist(self.tk.call(

self.\_w, 'tag', 'ranges', tagName))

def tag\_remove(self, tagName, index1, index2=None):

"""Remove tag TAGNAME from all characters between INDEX1 and INDEX2."""

self.tk.call(

self.\_w, 'tag', 'remove', tagName, index1, index2)

def window\_cget(self, index, option):

"""Return the value of OPTION of an embedded window at INDEX."""

if option[:1] != '-':

option = '-' + option

if option[-1:] == '\_':

option = option[:-1]

return self.tk.call(self.\_w, 'window', 'cget', index, option)

def window\_configure(self, index, cnf=None, \*\*kw):

"""Configure an embedded window at INDEX."""

return self.\_configure(('window', 'configure', index), cnf, kw)

window\_config = window\_configure

def window\_create(self, index, cnf={}, \*\*kw):

"""Create a window at INDEX."""

self.tk.call(

(self.\_w, 'window', 'create', index)

+ self.\_options(cnf, kw))

def window\_names(self):

"""Return all names of embedded windows in this widget."""

return self.tk.splitlist(

self.tk.call(self.\_w, 'window', 'names'))

def yview\_pickplace(self, \*what):

"""Obsolete function, use see."""

self.tk.call((self.\_w, 'yview', '-pickplace') + what)

class \_setit:

"""Internal class. It wraps the command in the widget OptionMenu."""

def \_\_init\_\_(self, var, value, callback=None):

self.\_\_value = value

self.\_\_var = var

self.\_\_callback = callback

def \_\_call\_\_(self, \*args):

self.\_\_var.set(self.\_\_value)

if self.\_\_callback:

self.\_\_callback(self.\_\_value, \*args)

class OptionMenu(Menubutton):

"""OptionMenu which allows the user to select a value from a menu."""

def \_\_init\_\_(self, master, variable, value, \*values, \*\*kwargs):

"""Construct an optionmenu widget with the parent MASTER, with

the resource textvariable set to VARIABLE, the initially selected

value VALUE, the other menu values VALUES and an additional

keyword argument command."""

kw = {"borderwidth": 2, "textvariable": variable,

"indicatoron": 1, "relief": RAISED, "anchor": "c",

"highlightthickness": 2}

Widget.\_\_init\_\_(self, master, "menubutton", kw)

self.widgetName = 'tk\_optionMenu'

menu = self.\_\_menu = Menu(self, name="menu", tearoff=0)

self.menuname = menu.\_w

# 'command' is the only supported keyword

callback = kwargs.get('command')

if 'command' in kwargs:

del kwargs['command']

if kwargs:

raise TclError, 'unknown option -'+kwargs.keys()[0]

menu.add\_command(label=value,

command=\_setit(variable, value, callback))

for v in values:

menu.add\_command(label=v,

command=\_setit(variable, v, callback))

self["menu"] = menu

def \_\_getitem\_\_(self, name):

if name == 'menu':

return self.\_\_menu

return Widget.\_\_getitem\_\_(self, name)

def destroy(self):

"""Destroy this widget and the associated menu."""

Menubutton.destroy(self)

self.\_\_menu = None

class Image:

"""Base class for images."""

\_last\_id = 0

def \_\_init\_\_(self, imgtype, name=None, cnf={}, master=None, \*\*kw):

self.name = None

if not master:

master = \_default\_root

if not master:

raise RuntimeError, 'Too early to create image'

self.tk = getattr(master, 'tk', master)

if not name:

Image.\_last\_id += 1

name = "pyimage%r" % (Image.\_last\_id,) # tk itself would use image<x>

# The following is needed for systems where id(x)

# can return a negative number, such as Linux/m68k:

if name[0] == '-': name = '\_' + name[1:]

if kw and cnf: cnf = \_cnfmerge((cnf, kw))

elif kw: cnf = kw

options = ()

for k, v in cnf.items():

if hasattr(v, '\_\_call\_\_'):

v = self.\_register(v)

elif k in ('data', 'maskdata'):

v = self.tk.\_createbytearray(v)

options = options + ('-'+k, v)

self.tk.call(('image', 'create', imgtype, name,) + options)

self.name = name

def \_\_str\_\_(self): return self.name

def \_\_del\_\_(self):

if self.name:

try:

self.tk.call('image', 'delete', self.name)

except TclError:

# May happen if the root was destroyed

pass

def \_\_setitem\_\_(self, key, value):

self.tk.call(self.name, 'configure', '-'+key, value)

def \_\_getitem\_\_(self, key):

return self.tk.call(self.name, 'configure', '-'+key)

def configure(self, \*\*kw):

"""Configure the image."""

res = ()

for k, v in \_cnfmerge(kw).items():

if v is not None:

if k[-1] == '\_': k = k[:-1]

if hasattr(v, '\_\_call\_\_'):

v = self.\_register(v)

elif k in ('data', 'maskdata'):

v = self.tk.\_createbytearray(v)

res = res + ('-'+k, v)

self.tk.call((self.name, 'config') + res)

config = configure

def height(self):

"""Return the height of the image."""

return getint(

self.tk.call('image', 'height', self.name))

def type(self):

"""Return the type of the imgage, e.g. "photo" or "bitmap"."""

return self.tk.call('image', 'type', self.name)

def width(self):

"""Return the width of the image."""

return getint(

self.tk.call('image', 'width', self.name))

class PhotoImage(Image):

"""Widget which can display colored images in GIF, PPM/PGM format."""

def \_\_init\_\_(self, name=None, cnf={}, master=None, \*\*kw):

"""Create an image with NAME.

Valid resource names: data, format, file, gamma, height, palette,

width."""

Image.\_\_init\_\_(self, 'photo', name, cnf, master, \*\*kw)

def blank(self):

"""Display a transparent image."""

self.tk.call(self.name, 'blank')

def cget(self, option):

"""Return the value of OPTION."""

return self.tk.call(self.name, 'cget', '-' + option)

# XXX config

def \_\_getitem\_\_(self, key):

return self.tk.call(self.name, 'cget', '-' + key)

# XXX copy -from, -to, ...?

def copy(self):

"""Return a new PhotoImage with the same image as this widget."""

destImage = PhotoImage(master=self.tk)

self.tk.call(destImage, 'copy', self.name)

return destImage

def zoom(self,x,y=''):

"""Return a new PhotoImage with the same image as this widget

but zoom it with X and Y."""

destImage = PhotoImage(master=self.tk)

if y=='': y=x

self.tk.call(destImage, 'copy', self.name, '-zoom',x,y)

return destImage

def subsample(self,x,y=''):

"""Return a new PhotoImage based on the same image as this widget

but use only every Xth or Yth pixel."""

destImage = PhotoImage(master=self.tk)

if y=='': y=x

self.tk.call(destImage, 'copy', self.name, '-subsample',x,y)

return destImage

def get(self, x, y):

"""Return the color (red, green, blue) of the pixel at X,Y."""

return self.tk.call(self.name, 'get', x, y)

def put(self, data, to=None):

"""Put row formatted colors to image starting from

position TO, e.g. image.put("{red green} {blue yellow}", to=(4,6))"""

args = (self.name, 'put', data)

if to:

if to[0] == '-to':

to = to[1:]

args = args + ('-to',) + tuple(to)

self.tk.call(args)

# XXX read

def write(self, filename, format=None, from\_coords=None):

"""Write image to file FILENAME in FORMAT starting from

position FROM\_COORDS."""

args = (self.name, 'write', filename)

if format:

args = args + ('-format', format)

if from\_coords:

args = args + ('-from',) + tuple(from\_coords)

self.tk.call(args)

class BitmapImage(Image):

"""Widget which can display a bitmap."""

def \_\_init\_\_(self, name=None, cnf={}, master=None, \*\*kw):

"""Create a bitmap with NAME.

Valid resource names: background, data, file, foreground, maskdata, maskfile."""

Image.\_\_init\_\_(self, 'bitmap', name, cnf, master, \*\*kw)

def image\_names():

return \_default\_root.tk.splitlist(\_default\_root.tk.call('image', 'names'))

def image\_types():

return \_default\_root.tk.splitlist(\_default\_root.tk.call('image', 'types'))

class Spinbox(Widget, XView):

"""spinbox widget."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a spinbox widget with the parent MASTER.

STANDARD OPTIONS

activebackground, background, borderwidth,

cursor, exportselection, font, foreground,

highlightbackground, highlightcolor,

highlightthickness, insertbackground,

insertborderwidth, insertofftime,

insertontime, insertwidth, justify, relief,

repeatdelay, repeatinterval,

selectbackground, selectborderwidth

selectforeground, takefocus, textvariable

xscrollcommand.

WIDGET-SPECIFIC OPTIONS

buttonbackground, buttoncursor,

buttondownrelief, buttonuprelief,

command, disabledbackground,

disabledforeground, format, from,

invalidcommand, increment,

readonlybackground, state, to,

validate, validatecommand values,

width, wrap,

"""

Widget.\_\_init\_\_(self, master, 'spinbox', cnf, kw)

def bbox(self, index):

"""Return a tuple of X1,Y1,X2,Y2 coordinates for a

rectangle which encloses the character given by index.

The first two elements of the list give the x and y

coordinates of the upper-left corner of the screen

area covered by the character (in pixels relative

to the widget) and the last two elements give the

width and height of the character, in pixels. The

bounding box may refer to a region outside the

visible area of the window.

"""

return self.\_getints(self.tk.call(self.\_w, 'bbox', index)) or None

def delete(self, first, last=None):

"""Delete one or more elements of the spinbox.

First is the index of the first character to delete,

and last is the index of the character just after

the last one to delete. If last isn't specified it

defaults to first+1, i.e. a single character is

deleted. This command returns an empty string.

"""

return self.tk.call(self.\_w, 'delete', first, last)

def get(self):

"""Returns the spinbox's string"""

return self.tk.call(self.\_w, 'get')

def icursor(self, index):

"""Alter the position of the insertion cursor.

The insertion cursor will be displayed just before

the character given by index. Returns an empty string

"""

return self.tk.call(self.\_w, 'icursor', index)

def identify(self, x, y):

"""Returns the name of the widget at position x, y

Return value is one of: none, buttondown, buttonup, entry

"""

return self.tk.call(self.\_w, 'identify', x, y)

def index(self, index):

"""Returns the numerical index corresponding to index

"""

return self.tk.call(self.\_w, 'index', index)

def insert(self, index, s):

"""Insert string s at index

Returns an empty string.

"""

return self.tk.call(self.\_w, 'insert', index, s)

def invoke(self, element):

"""Causes the specified element to be invoked

The element could be buttondown or buttonup

triggering the action associated with it.

"""

return self.tk.call(self.\_w, 'invoke', element)

def scan(self, \*args):

"""Internal function."""

return self.\_getints(

self.tk.call((self.\_w, 'scan') + args)) or ()

def scan\_mark(self, x):

"""Records x and the current view in the spinbox window;

used in conjunction with later scan dragto commands.

Typically this command is associated with a mouse button

press in the widget. It returns an empty string.

"""

return self.scan("mark", x)

def scan\_dragto(self, x):

"""Compute the difference between the given x argument

and the x argument to the last scan mark command

It then adjusts the view left or right by 10 times the

difference in x-coordinates. This command is typically

associated with mouse motion events in the widget, to

produce the effect of dragging the spinbox at high speed

through the window. The return value is an empty string.

"""

return self.scan("dragto", x)

def selection(self, \*args):

"""Internal function."""

return self.\_getints(

self.tk.call((self.\_w, 'selection') + args)) or ()

def selection\_adjust(self, index):

"""Locate the end of the selection nearest to the character

given by index,

Then adjust that end of the selection to be at index

(i.e including but not going beyond index). The other

end of the selection is made the anchor point for future

select to commands. If the selection isn't currently in

the spinbox, then a new selection is created to include

the characters between index and the most recent selection

anchor point, inclusive. Returns an empty string.

"""

return self.selection("adjust", index)

def selection\_clear(self):

"""Clear the selection

If the selection isn't in this widget then the

command has no effect. Returns an empty string.

"""

return self.selection("clear")

def selection\_element(self, element=None):

"""Sets or gets the currently selected element.

If a spinbutton element is specified, it will be

displayed depressed

"""

return self.selection("element", element)

###########################################################################

class LabelFrame(Widget):

"""labelframe widget."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a labelframe widget with the parent MASTER.

STANDARD OPTIONS

borderwidth, cursor, font, foreground,

highlightbackground, highlightcolor,

highlightthickness, padx, pady, relief,

takefocus, text

WIDGET-SPECIFIC OPTIONS

background, class, colormap, container,

height, labelanchor, labelwidget,

visual, width

"""

Widget.\_\_init\_\_(self, master, 'labelframe', cnf, kw)

########################################################################

class PanedWindow(Widget):

"""panedwindow widget."""

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

"""Construct a panedwindow widget with the parent MASTER.

STANDARD OPTIONS

background, borderwidth, cursor, height,

orient, relief, width

WIDGET-SPECIFIC OPTIONS

handlepad, handlesize, opaqueresize,

sashcursor, sashpad, sashrelief,

sashwidth, showhandle,

"""

Widget.\_\_init\_\_(self, master, 'panedwindow', cnf, kw)

def add(self, child, \*\*kw):

"""Add a child widget to the panedwindow in a new pane.

The child argument is the name of the child widget

followed by pairs of arguments that specify how to

manage the windows. The possible options and values

are the ones accepted by the paneconfigure method.

"""

self.tk.call((self.\_w, 'add', child) + self.\_options(kw))

def remove(self, child):

"""Remove the pane containing child from the panedwindow

All geometry management options for child will be forgotten.

"""

self.tk.call(self.\_w, 'forget', child)

forget=remove

def identify(self, x, y):

"""Identify the panedwindow component at point x, y

If the point is over a sash or a sash handle, the result

is a two element list containing the index of the sash or

handle, and a word indicating whether it is over a sash

or a handle, such as {0 sash} or {2 handle}. If the point

is over any other part of the panedwindow, the result is

an empty list.

"""

return self.tk.call(self.\_w, 'identify', x, y)

def proxy(self, \*args):

"""Internal function."""

return self.\_getints(

self.tk.call((self.\_w, 'proxy') + args)) or ()

def proxy\_coord(self):

"""Return the x and y pair of the most recent proxy location

"""

return self.proxy("coord")

def proxy\_forget(self):

"""Remove the proxy from the display.

"""

return self.proxy("forget")

def proxy\_place(self, x, y):

"""Place the proxy at the given x and y coordinates.

"""

return self.proxy("place", x, y)

def sash(self, \*args):

"""Internal function."""

return self.\_getints(

self.tk.call((self.\_w, 'sash') + args)) or ()

def sash\_coord(self, index):

"""Return the current x and y pair for the sash given by index.

Index must be an integer between 0 and 1 less than the

number of panes in the panedwindow. The coordinates given are

those of the top left corner of the region containing the sash.

pathName sash dragto index x y This command computes the

difference between the given coordinates and the coordinates

given to the last sash coord command for the given sash. It then

moves that sash the computed difference. The return value is the

empty string.

"""

return self.sash("coord", index)

def sash\_mark(self, index):

"""Records x and y for the sash given by index;

Used in conjunction with later dragto commands to move the sash.

"""

return self.sash("mark", index)

def sash\_place(self, index, x, y):

"""Place the sash given by index at the given coordinates

"""

return self.sash("place", index, x, y)

def panecget(self, child, option):

"""Query a management option for window.

Option may be any value allowed by the paneconfigure subcommand

"""

return self.tk.call(

(self.\_w, 'panecget') + (child, '-'+option))

def paneconfigure(self, tagOrId, cnf=None, \*\*kw):

"""Query or modify the management options for window.

after window

Insert the window after the window specified. window

should be the name of a window already managed by pathName.

before window

Insert the window before the window specified. window

should be the name of a window already managed by pathName.

height size

Specify a height for the window. The height will be the

outer dimension of the window including its border, if

any. If size is an empty string, or if -height is not

specified, then the height requested internally by the

window will be used initially; the height may later be

adjusted by the movement of sashes in the panedwindow.

Size may be any value accepted by Tk\_GetPixels.

minsize n

Specifies that the size of the window cannot be made

less than n. This constraint only affects the size of

the widget in the paned dimension -- the x dimension

for horizontal panedwindows, the y dimension for

vertical panedwindows. May be any value accepted by

Tk\_GetPixels.

padx n

Specifies a non-negative value indicating how much

extra space to leave on each side of the window in

the X-direction. The value may have any of the forms

accepted by Tk\_GetPixels.

pady n

sticky style

If a window's pane is larger than the requested

dimensions of the window, this option may be used

to position (or stretch) the window within its pane.

Style is a string that contains zero or more of the

characters n, s, e or w. The string can optionally

contains spaces or commas, but they are ignored. Each

letter refers to a side (north, south, east, or west)

that the window will "stick" to. If both n and s

(or e and w) are specified, the window will be

stretched to fill the entire height (or width) of

its cavity.

width size

Specify a width for the window. The width will be

the outer dimension of the window including its

border, if any. If size is an empty string, or

if -width is not specified, then the width requested

internally by the window will be used initially; the

width may later be adjusted by the movement of sashes

in the panedwindow. Size may be any value accepted by

Tk\_GetPixels.

"""

if cnf is None and not kw:

return self.\_getconfigure(self.\_w, 'paneconfigure', tagOrId)

if type(cnf) == StringType and not kw:

return self.\_getconfigure1(

self.\_w, 'paneconfigure', tagOrId, '-'+cnf)

self.tk.call((self.\_w, 'paneconfigure', tagOrId) +

self.\_options(cnf, kw))

paneconfig = paneconfigure

def panes(self):

"""Returns an ordered list of the child panes."""

return self.tk.splitlist(self.tk.call(self.\_w, 'panes'))

######################################################################

# Extensions:

class Studbutton(Button):

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

Widget.\_\_init\_\_(self, master, 'studbutton', cnf, kw)

self.bind('<Any-Enter>', self.tkButtonEnter)

self.bind('<Any-Leave>', self.tkButtonLeave)

self.bind('<1>', self.tkButtonDown)

self.bind('<ButtonRelease-1>', self.tkButtonUp)

class Tributton(Button):

def \_\_init\_\_(self, master=None, cnf={}, \*\*kw):

Widget.\_\_init\_\_(self, master, 'tributton', cnf, kw)

self.bind('<Any-Enter>', self.tkButtonEnter)

self.bind('<Any-Leave>', self.tkButtonLeave)

self.bind('<1>', self.tkButtonDown)

self.bind('<ButtonRelease-1>', self.tkButtonUp)

self['fg'] = self['bg']

self['activebackground'] = self['bg']

######################################################################

# Test:

def \_test():

root = Tk()

text = "This is Tcl/Tk version %s" % TclVersion

if TclVersion >= 8.1:

try:

text = text + unicode("\nThis should be a cedilla: \347",

"iso-8859-1")

except NameError:

pass # no unicode support

label = Label(root, text=text)

label.pack()

test = Button(root, text="Click me!",

command=lambda root=root: root.test.configure(

text="[%s]" % root.test['text']))

test.pack()

root.test = test

quit = Button(root, text="QUIT", command=root.destroy)

quit.pack()

# The following three commands are needed so the window pops

# up on top on Windows...

root.iconify()

root.update()

root.deiconify()

root.mainloop()

if \_\_name\_\_ == '\_\_main\_\_':

\_test()