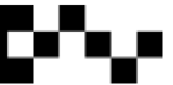


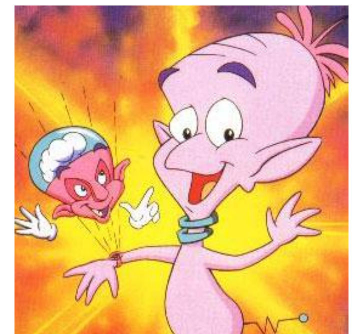
Python Tkinter

Python Tkinter is used to create interfaces.

Then you develop a user interface (UI) there is a standard set of tasks you must accomplish:

1. You must specify how you want the UI to look. That is, you must write code that determines what the user will see on the computer screen.
2. You must decide what the UI will do. That is, you must write routines that accomplish the tasks of the program.
3. You must associate the "looking" with the "doing". That is, you must write code that associates things that the user sees on the screen with the routines that you have written to perform the program's tasks.
4. Finally, you must write code that sits and waits for input from the user. Any UI should jump around and act silly.

GUI (pronounced "gooey") is an acronym for Graphical User Interface and it has some special jargon that is only used when dealing with them.

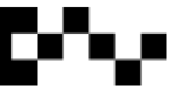


Widget - a cartoon character from the early 90's. Tkinter's widgets don't look like this character!

- We specify how we want a GUI to look by describing the **widgets** we want to display, and their special relationships (i.e. whether one widget is above or below, or to the right or left of other widgets). The word *widget* is a nonsense word that has become the common term for GUI components. Widgets include things such as windows, buttons, menus icons, drop-down lists, scroll bars and so on.
- The routines that actually do the work of the GUI are called **event handlers**. Events are read as "input by the user", such as mouse clicks, or a press of a key on the keyboard. These routines are called *handlers* because they handle (that is, respond to) such events.
- Associating an event handler with a widget is called **binding**. Roughly, the process of binding involved associating three different things:
    1. A type of event (e.g. click of the left mouse button, or by pressing ENTER on the keyboard),
    2. A widget (e.g. button),
    3. An event handler routine

  For example, we might bind a single click of the left mouse button on the CLOSE button on the screen to run the "closeProgram" routine, which closes the window and shuts down the program.

- The code that sits and waits for input is called the **event loop**.



About the Event Loop

If you believe the movies, every small town has a little old lady who spends all of her time at her front window, just WATCHING. She sees everything that goes on in the neighbourhood. A lot of what she sees is uninteresting of course -- just people going to and fro in the street. But some of it is interesting -- like a big fight between the newly-wed couple in the house across the street. When interesting events happen, the watchdog lady immediately is on the phone with the news to the police or to her neighbours.

The event loop is a lot like this watchdog lady. The event loop spends all of its time watching events go by, and it sees all of them. Most of the events are uninteresting, and when it sees them, it does nothing. But when it sees something interesting -- an event that it knows is interesting, because an event handler has been bound to the event -- then it immediately calls up the event handler and lets it know that the event has happened.

The simplest Tkinter program ever!

All you need to type is: (save as **tk_simple.py**)

```
from tkinter import *
root = Tk()
root.mainloop()
```

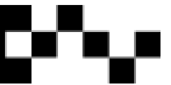
1. The first line imports Tkinter (be careful with capital and lowercase letters in the module name!!)
2. The second line creates a **top level** window. Technically, this statement is creating a copy (an instance) of the class “tkinter.Tk”.

   The top level window is the highest-leve GUI component in any Tkinter application. by conventions, the top level window is always named **root**.

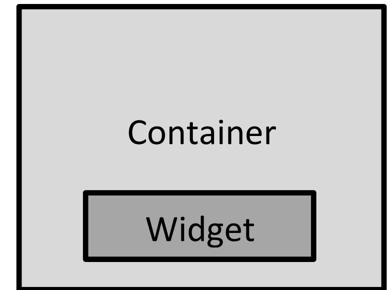
3. The third line executes the mainloop (aka the event loop) method of the root object. As the mainloop runs, it waits for events to happen in root. If an event occurs, then it is handled and the loop continues running, waiting for the next event. The loop executes until a **destroy** event happens to the root window. A destroy event is one that closes the window. When the root is destroyed, the window is closed and the event loop is exited. The main loop is **always** on the bottom of your script.

When you run the program, you will see that (thanks to the people who make Tk) the top level window automatically comes with widgets: minimise, maximise and close. They will work like any other program’s window.

By clicking the close widget, this forces the event loop to search through the program and find the destroy event. When it finds it, it terminates the window.



Specifying how the GUI should look



From now on, we will use two terms**, container** and **widget**. Widgets are components that (usually) are visible and do things. A container is the place where we put widgets.

Tkinter provides a number of containers:

- **Canvas** is a container for drawing applications
- **Frame** is a more frequently used window

Frames are created by calling the Frame module: e.g., Frame()

Within the parenthesis it is required that you name a parameter. The new parameter acts as a placeholder for the newly created frame. The frame parameter is always named the same as the top level window, root.

To create a frame you have to make an instance of the Frame module:

```
container1 = Frame(root)
```

When creating an instance of a class (whether it's a frame or a widget) always store it in a variable.

In short, it creates a container into which we can put widgets.

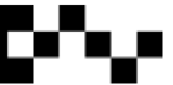
```
container1.pack()
```

Packing is a process of setting up the visual elements in your GUI. You must always pack containers and widgets, or they will not be seen when you run your GUI.

When you run this program, it will look very much like the previous one, except that there will be less to see. That is because…

Frames are elastic

A frame is literally a frame, like around a photo. The space within the frame is called the **cavity**. The cavity is stretchy like a rubber band. Unless you specify a window size for the frame, the cavity will stretch or shrink to accommodate whatever is placed within it.



Button Widget

To create a button:

```
button1 = Button(container1)
```

This creates a new button, and associates it with the container we want it to be placed in.

Button Attributes

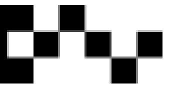
Widgets have lots of attributes that control their size, text and background colours, the text they display, how their borders look, and so on.

```
button1[“text”] = “Hello, world!”

button1[“background”] = “green”

button1.pack()    #always, always, always pack the objects!
```

Now *container1* has stretched to accommodate *button1*.

Your code should look like this:

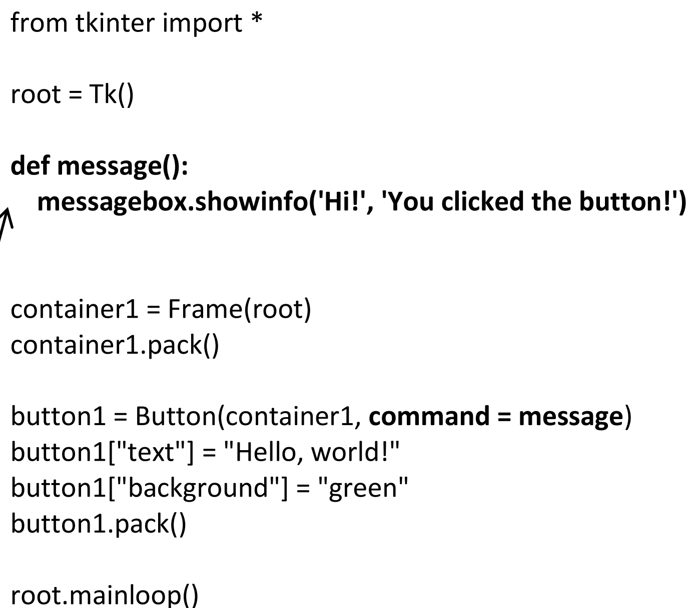
```
from tkinter import *

root = Tk()

container1 = Frame(root)
container1.pack()

button1 = Button(container1)
button1["text"] = "Hello, world!"
button1["background"] = "green"
button1.pack()

root.mainloop()
```



Button reactions

Just say you want a dialog box to pop up when the user presses the button.

```
from tkinter import *

root = Tk()

def message():
   messagebox.showinfo('Hi!', 'You clicked the button!')


container1 = Frame(root)
container1.pack()

button1 = Button(container1, command = message)
button1["text"] = "Hello, world!"
button1["background"] = "green"
button1.pack()

root.mainloop()
```



Create a function that displays the message. The function calls the **messagebox** module in Tk and reads the **showinfo()** function within the module. Inside the parenthesis, the first string is the name of the dialog box, and the second string is the message you want to appear inside the box.

In the button1 variable, you have to tell Python to call the function you have just created. You do this by typing: **command = NameOfFunction**.

Other dialog boxes

There are a range of dialog boxes you can use, you should try them all out!

showinfo,

showwarning

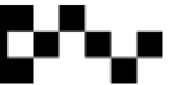
showerror

askquestion

askokcancel

askyesno

askyesnocancel



Collapsing events

Most of the time events can take up many lines of code. It is possible to collapse some of these lines into a single line of code.

For example - instead of writing:

```
button1 = Button(container1, command = message)
button1["text"] = "Hello, world!"
button1["background"] = "green"
button1.pack()
```

It can be written:

```
button1 = Button(container1, text= 'New', bg = 'green', width=6,  command=message).pack()
```

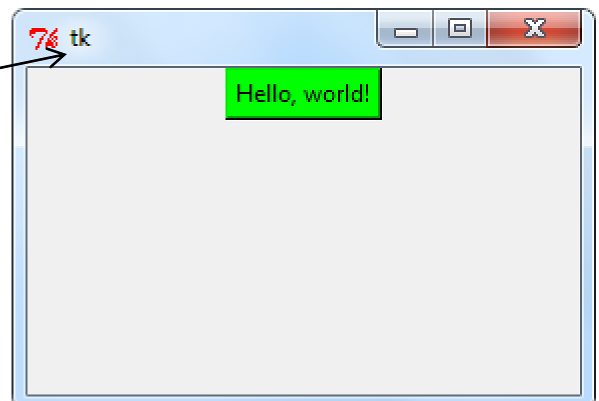
By specifying the width or height of the button you can change the way the buttons look.

In the .pack() function you can include whether you want the button to sit on the left, right, top or bottom, and the padding surrounding the button. It is typed like this:

```
.pack(side=LEFT, padx=2, pady=2)
```

Naming the Window

Usually the window is named **tk**.



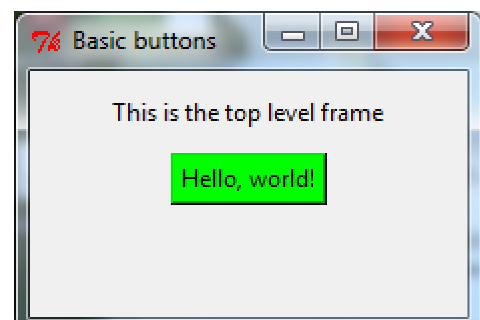
You can change it by typing:

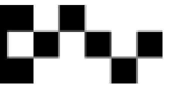
```
root = Tk()
root.title('New Window Title')
```

Adding text into your window

To add text into your window, type:

```
Label(root, text='This is some text.').pack(pady=10)
```





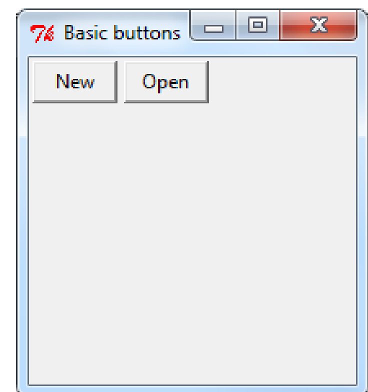
Changing the size of your window

To change the size of the main window, type:

root.geometry("200x200")

You can type any pixel ratio in the parenthesis. The first number is x-axis (width), the second is the y-axis (height).

Making a toolbar

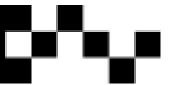


To make a toolbar you must have several buttons, but for this we will have two buttons, New and Open.

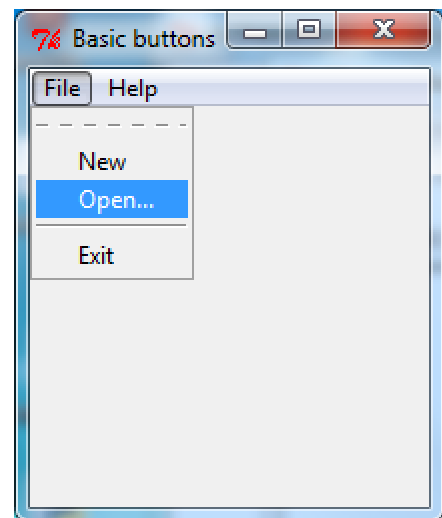
First we have to create a new container for the toolbar to sit in.

```
def buttons():
   print('pressed the button!')

toolbar = Frame(root)

new_btn = Button(toolbar, text='New',  width=6, command=buttons).pack(side=LEFT,
padx=2, pady=2)

open_btn = Button(toolbar, text='Open', width=6, command=buttons).pack(side=LEFT,
padx=2, pady=2)

toolbar.pack(side=TOP, fill=X)
```

The fill=x means that if you expand the window, the button will grow to fill the window only along the x-axis. You can change it to Y if you want the buttons to expand along the y-axis.

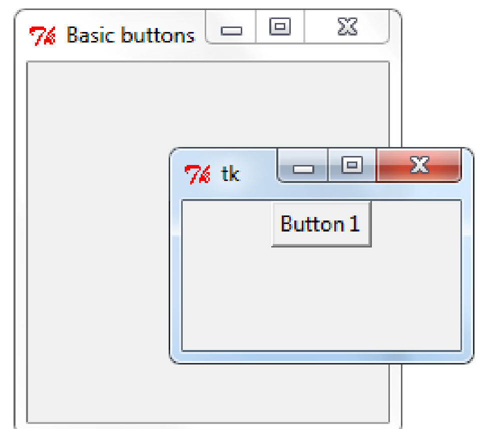


Making a menu




```
menu = Menu(root)
root.config(menu=menu)

filemenu = Menu(menu)
menu.add_cascade(label="File", menu=filemenu)

filemenu.add_command(label='New', command=buttons)
filemenu.add_command(label='Open...', command=buttons)
filemenu.add_separator()
filemenu.add_command(label='Exit', command=buttons)


helpmenu = Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label="About...", command=buttons)
```

An independent window

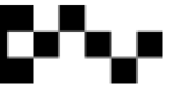


To set up an independent window all you have to do is set up a new variable to store it. Up until now we have been using root as the main window, all we have to do is create a new one.

```
window1 = Tk()

Button(window1, text='Button 1', command=window1.destory).pack()
```

The command called is an built in destroy function.

When dealing with lots of independent windows, you should create modules to store each window, so that the main code doesn't get too long.



Changing the cursor

```
from tkinter import *
widget = Button(text='Button', padx=10, pady=10)
widget.pack(padx=20, pady=20)
widget.config(cursor='gumby')

widget.config(font=('helvetica', 20, 'underline italic'))
mainloop()
```

Python Tkinter supports quite a number of different mouse cursors available. The exact graphic may vary according to your operating system.

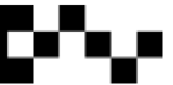
Here is the list of interesting ones:

- "arrow"
- "circle"
- "clock"
- "cross"
- "dotbox"
- "exchange"
- "fleur"
- "heart"
- "heart"
- "man"
- "mouse"
- "pirate"
- "plus"
- "shuttle"
- "sizing"
- "spider"
- "spraycan"
- "star"
- "target"
- "tcross"
- "trek"
- "watch"

Custom Window Icon

Create a custom Icon, make sure it is 24x24px. Save as an .ico and it in the same folder as your scripts.

```
root = Tk()
root.title('New Window Title')
root.iconbitmap('iconName.ico')
```



Using a canvas

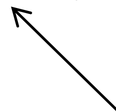
Lately we have only been using a Frame, now we will create a new script file and use a canvas.

Make sure you import tkinter:

```
from tkinter import *
```

Instead of naming this window **root**, we will call it **draw**, so when we call the canvas as a module in the root script we will know which one we are working with.

```
draw = Tk()
draw.title('Drawing area')
draw.geometry("500x500")


# code goes here


draw.mainloop()
```



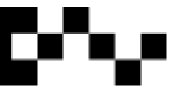
Remember that we have to pack the whole canvas at the end of the script.

A canvas works similar to a frame, but on a canvas you can have custom images on your menu buttons!!! (on a frame, you can't)

Configuring Widget Appearance

```
from tkinter import *
root = Tk()
labelfont = ('times', 20, 'bold')                  #font family, size, style
widget = label(root, text='Hello config world!')
widget.config(bg='black', fg='yellow')             # yellow text on black background
widget.config(font=labelfont)                      # use larger font
widget.config(height=3, width=20)                  # initial size: lines, characters
widget.pack(expand=YES, fill=BOTH)
root.mainloop()
```

.config changes the appearance of widgets.



Information to remember when configuring widgets

Colour – bg = 'color' #background
fg = 'color' #foreground'

Size - Can be specified for any widget. Usually measured in lines high, characters wide.

Font - font – i.e font family – any generic font, stay away from fancy ones
size – in pixels
style – normal, bold, roman, italic, underline, overstrike, or combinations e.g, bold italic

Layout and Expansion – expand – allows widgets (i.e window to expand)
fill – makes widgets fill the window

Border and Relief – bd = n - n is the width of the border in pixels
relief – can be FLAT, SUNKEN, RAISED, GROVE, SOLID, or RIDGE

Cursor - cursor - see page 9.

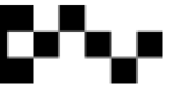
State - state = DISABLED (deactivates widgets), NORMAL, READONLY(unresponsive)

Padding - padx = pixel width. Space on the left and right of the widget.
Pady = pixel width. Space on the top and bottom of the widget.

Custom images

Here are some rules to go with custom button images:

- They shouldn't be too big, icons are usually 24x24px, or 32x32px. For this example my images are 50x50px.
- All images ***<u>have</u>*** to be saved as .gifs – else they won't work. Other image formats save too much information in with the image's code.
- Save the images as btn_*Name*.gif, (the *Name* can be anything you want - keep it sensible). The reason for this is so you can find it in your files, and know what it is for.
- Save the images in the same folder as your GUI scripts. If you don't they won't be found.



Using custom images

These are my buttons. btn_home.gif, btn_one.gif, btn_two.gif, btn_three.gif, and btn_four.gif

The first thing you have to do is create variables to store the images:

```
img0 = PhotoImage(file='btn_home.gif')
img1 = PhotoImage(file='btn_one.gif')
img2 = PhotoImage(file='btn_two.gif')
img3 = PhotoImage(file='btn_three.gif')
img4 = PhotoImage(file='btn_four.gif')
```

Then set up a toolbar to place the buttons:

```
toolbar = Canvas(draw)
```

Create functions for each button:

```
def btnHome():
   messagebox.askyesno('Going home!', 'Are you going home?')
def btnOne():
   messagebox.showwarning('One!', 'You pressed One')
def btnTwo():
   messagebox.askquestion('Two!', 'Does Flow rhyme with No?')
def btnThree():
   messagebox.showerror('Three!', 'That\'s not right!')
def btnFour():
   messagebox.showinfo('Four!', 'Four is the fourth number when you start counting from one!')
```

Create the buttons:

```
home = Button(toolbar, command = btnHome, image = img0).pack(side=LEFT, padx=2, pady=2)
one = Button(toolbar, command = btnOne, image = img1).pack(side=LEFT, padx=2, pady=2)
two = Button(toolbar, command = btnTwo, image = img2).pack(side=LEFT, padx=2, pady=2)
three = Button(toolbar, command = btnThree, image = img3).pack(side=LEFT, padx=2, pady=2)
four = Button(toolbar, command = btnFour, image = img4).pack(side=LEFT, padx=2, pady=2)
```

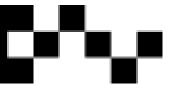
Pack the toolbar:

```
toolbar.pack(side=TOP, fill=X)
```

And you're done!

FYI: if you don't put the **side = LEFT** in each button, the buttons will sit beneath each other (y-axis).

Let's get all technical, and side track a little….



Object Oriented Programming (OOP)

An object is a script that contains data and methods. It looks like a module, but it behaves differently. When using objects, computers can run programs much faster than reading a single script.

An object has built in attributes that help to describe it to other scripts. For example, if *students* was the object, you would have attributes to **describe** it.

students.smart

Objects also have methods, they are exactly like functions, but you call it a *method* when it is inside an *object*. A method uses the data from the object it is coded in and it describes things that the object **can do**, for example:

students.learning

Before you do all of this, you have to build a *class*, which is a blueprint for your object.

In IDLE Type :

```
>>>class students:
```

This tells Python that you will be building a *class* called *students*.

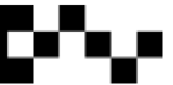
In your class you will have a lot of different types of data; variables and methods etc. Classes are like a cookie cutter, every time you use it, it will create a cookie for you.

Type:

```
>>>uniform = 'green'
>>>hair = 'short'
>>>shirts = 'white'
>>>ties =True
```

These variables describe what the class does and specify how they will be used within the class (depending on their data type (string, Boolean, integer etc)).

Make sure the variables are indented in one place.



Methods

To make a method is exactly the same as making a function.

>>>def learning()

The only thing you have to do is add parameters in the parenthesis, this helps the class behave correctly.

The first parameter you must use is *self*. A description about self is in the **More Methods** section, below.

>>>def learning(self):

Now you can add code into your method. Make sure that the code is indented.

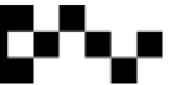
>>>def learning(self):
    >>>return ('This is the learning method')

Press Enter twice. This creates the class.

```
>>> class students:
        uniform = 'green'
        hair = 'short'
        shirts = 'white'
        ties = True
        def learning(self):
                return ("This is the learning method")


>>> 
```

If you type *students* into IDLE you will get a strange message, this means that the class has been created.

```
>>> students
<class '__main__.students'>
>>> 
```



Objects

You need to make objects to get the data from your class. An object is like a variable that stores the class.

studentObject = students()

Have empty parameters.

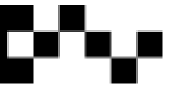
The *studentObject* now refers to the class *students*.

Now you can call the variables and methods inside the class:

studentObject.shirts

call variables with a full stop [.].

```
>>> studentObject.shirts
'white'
>>> studentObject.hair
'short'
>>> studentObject.ties
True
>>> studentObject.learning()
'This is the learning method'
>>>
```



More methods

In IDLE create a new class:

```
>>> class className:
	def createName(self,name):
		self.name=name
	def displayName(self):
		return self.name
	def saying(self):
		print("hello %s" % self.name)

>>> 
```

Self is a temporary placeholder for the object, which means that if you have an object name *apples*, then whenever you call the object it will change *self* to *apples*. You can name the first parameter (*self*) anything you want, but *self* is the **standard** word that all programmers use when creating classes. Every method you create in a class must have the *self* parameter first.

You can create lots of objects from the methods in the class, I created two, first and second.

```
>>> first=className()
>>> second=className()
>>> 
```

```
>>> class className:
	def createName(self,name):
		self.name=name
	def displayName(self):
		return self.name
	def saying(self):
		print("hello %s" % self.name)


>>> className
<class '__main__.className'>
>>> first=className()
>>> second=className()
>>> first.createName('Nigel')
>>> 
```

To call the methods in your class, you have to type the object name, a full stop[.], then the method you want to call.

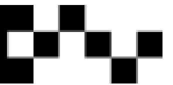
The *createName* method has two parameters, *self* and *name*.

The *self* parameter stores the name of the object, in this case *first*.

The *name* parameter will store the name you type inside the parenthesis (e.g. 'Nigel')

Call the *second* object with another name.

```
>>> first.createName('Nigel')
>>> second.createName('Wade')
>>> 
```



This will use the class to store two completely different names in the same *name* parameter.

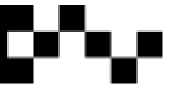
To display the name, type:

```
>>> first.createName('Nigel')
>>> second.createName('Wade')
>>> first.displayName()
'Nigel'
>>> second.displayName()
'Wade'
>>>
```

This reads the function called displayName and changes the *self* parameter to the name stored in the *first* object.

Use the saying method to print the name.

```
>>> first.saying()
hello Nigel
>>> second.saying()
hello Wade
>>>
```



Back to tkinter

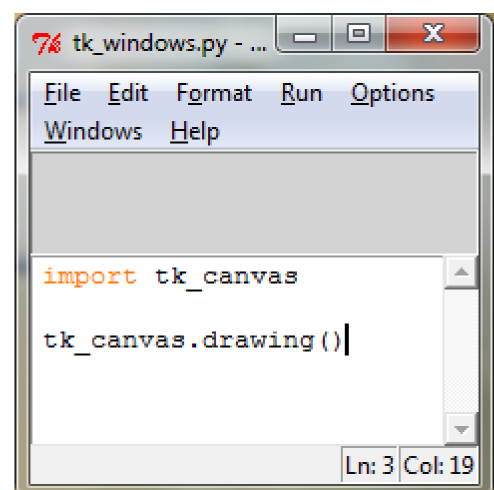
Open up the canvas script you created on page 10, and create a class. Make sure you indent everything. Save your file as **tk_canvas.py**.

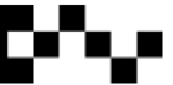
```
from tkinter import *

class drawing:

    draw = Tk()
    draw.title('Drawing area')
    draw.geometry("500x500")

    img0 = PhotoImage(file='btn-home.gif')
    img1 = PhotoImage(file='btn-one.gif')
    img2 = PhotoImage(file='btn-two.gif')
    img3 = PhotoImage(file='btn-three.gif')
    img4 = PhotoImage(file='btn-four.gif')

    toolbar = Canvas(draw)

    def btnHome():
        messagebox.askyesno('Going home!', 'Are you going home?')
    def btnOne():
        messagebox.showwarning('One!', 'You pressed One')
    def btnTwo():
        messagebox.askquestion('Two!', 'Does Flow rhyme with No?')
    def btnThree():
        messagebox.showerror('Three!', 'That\'s not right!')
    def btnFour():
        messagebox.showinfo('Four!', 'Four is the fourth number when you start counting from one!')

    home = Button(toolbar, command = btnHome, image = img0).pack(side=LEFT, padx=2, pady=2)
    one = Button(toolbar, command = btnOne, image = img1).pack(side=LEFT, padx=2, pady=2)
    two = Button(toolbar, command = btnTwo, image = img2).pack(side=LEFT, padx=2, pady=2)
    three = Button(toolbar, command = btnThree, image = img3).pack(side=LEFT, padx=2, pady=2)
    four = Button(toolbar, command = btnFour, image = img4).pack(side=LEFT, padx=2, pady=2)

    toolbar.pack(side=TOP, fill=X)

    draw.mainloop()
```

These functions don't need the *self* in the parenthesis, due to being used for GUI display. If they were used to store variables, *self* would be used.

Now this class can be called from within a new module. Name this new one **tk_windows.py**.

Save and run it!!





The different importing styles

You have probably found that there are two ways to import modules into python:

import moduleName

from moduleName import *

The first way is to import your **custom modules**. By importing modules this way you have to specify the moduleName then the function in your code, example: moduleName.functionname()

The second way is to import python's **built in modules**. Using this way you don't have to specify the moduleName throughout the script, like the above example.

_ _init_ _

__init__ (double under score on both sides) is a special method name used when defining your own classes. init is an abbreviation for initialisation.

__init__ makes coding classes a little more professional looking. But on the technical side, __init__ is a way to specify default values when you declare an object; the values are set in the class method.

For instance if I have a Point class... that has two objects which are the x and y coordinates i can do this:

```
>>> class Point():
        pass

>>> point = Point()
>>> point.x = 3.0
>>> point.y = 4.0
```

Instead you can do this:

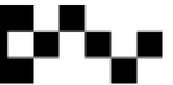
```
>>> class Point():
        def __init__(self, x=0,y=0):
            self.x = x
            self.y = y
```

now you can make a point like this:

```
>>> point=Point(3,4)
>>> 
```

There are many of those special methods or function names, all with prefix __ (double-under).

__main__ is used so the Python interpreter denotes the start of actual code, not including function and class definitions.

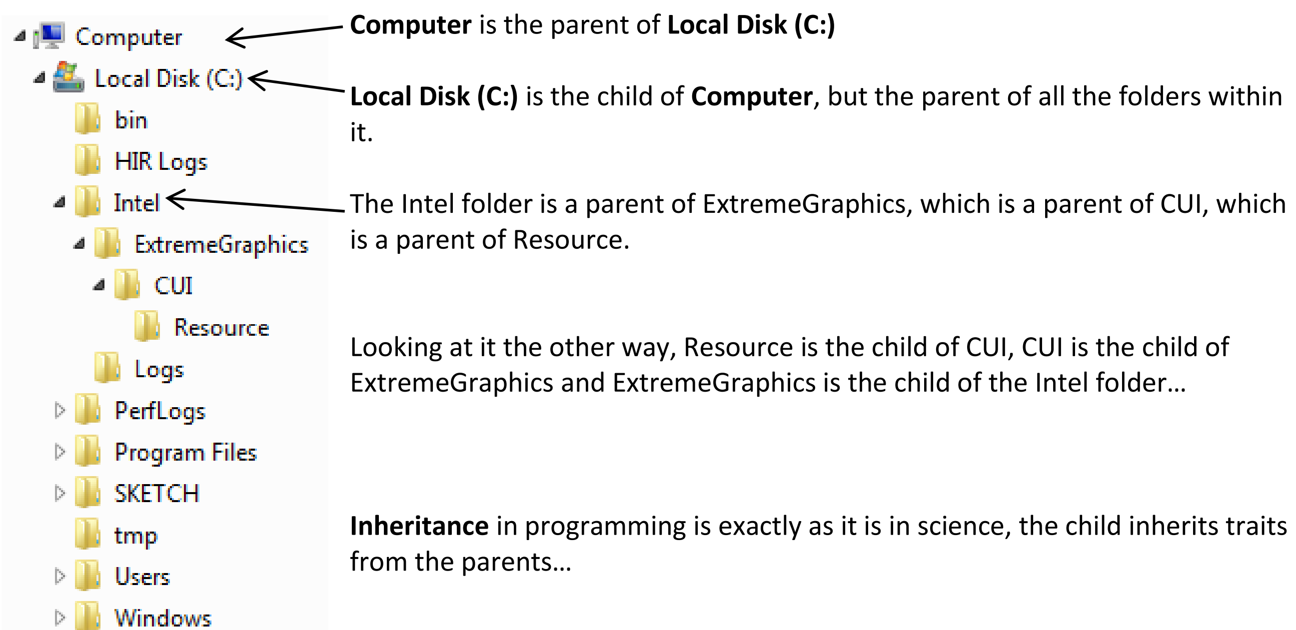


Parent Classes and Child Classes (aka, superclass and subclass)

Superclasses and sub classes are one way to get one class to inherit all traits of another class. This has to do with something called **parenting**.

Parenting is exactly as it sounds: parents have children, those children grow up and become parents, then have children of their own, and on into infinity.

When you are dealing with parenting in programs, it is the same thing. Take a folder structure for example:



In IDLE:

Create a **parentClass**, give it two variables.

```
>>> class parentClass:
        var1 = 'i am var 1'
        var2 = 'i am var 2'
```

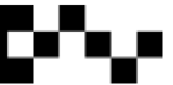
Create a **childClass**, now in the parenthesis you type the class you want it to inherit.
Then type *pass*, this means that the class doesn't do anything.

```
>>> class childClass(parentClass):
        pass
```

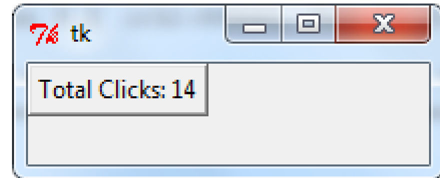
Create an object for the parent class, then get it to read the var1.

Because you told the childClass to inherit the information from the parentClass, you can set up a childObject and get it to print var1 (or var2!).

```
>>> parentObject=parentClass()
>>> parentObject.var1
'i am var 1'
>>> childObject=childClass()
>>> childObject.var1
'i am var 1'
>>> 
```



Counting clicks on a button




```
from tkinter import *

class Application(Frame):
    def __init__(self, master):
        Frame.__init__(self, master)
        self.grid()
        self.bttn_clicks = 0
        self.create_widget()

    def create_widget(self):
        self.bttn = Button(self)
        self.bttn["text"]= "Total Clicks: 0"
        self.bttn["command"] = self.update_count
        self.bttn.grid()

    def update_count(self):
        self.bttn_clicks += 1
        self.bttn["text"] = "Total Clicks: " + str(self.bttn_clicks)

root = Tk()
root.geometry("200x50")
app = Application(root)
root.mainloop()
```

Getting buttons to do something

This is where we write event handler routines to do the actual work of the program. An event handler is a method that handles events when they occur.

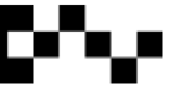
In tkinter, the way you create this is through the bind() method, example:

widget.bind(event_type_name, event_handler_name)

Before we begin, we need to point out a possible point of confusion. The word "button" can be used to refer to two entirely different things: (1) a button widget -- a GUI component that is displayed on the computer monitor -- and (2) a button on your mouse -- the kind of button that you press with your finger. In order to avoid confusion, I will usually try to distinguish them by referring to "button widget" or "mouse button" rather than simply to "button".

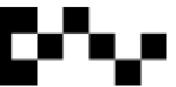
When we bind events onto widget buttons we use an “<eventName>” tag. It looks similar to HTML code.

The type of binding we will use is a "<Button-1>" event that reads the click of the left mouse button.



Type this code into a new script file. Save it as **tk_bind.py**

```
from tkinter import *

def showPosEvent(event):
    print('Widget = %s   X=%s  Y=%s ' % (event.widget, event.x, event.y))

def onKeyPress(event):
    print('Got key press: ', event.char)

def onArrowKey(event):
    print('Arrow pressed')

def onReturnKey(event):
    print('Enter pressed')

def onLeftClick(event):
    print('Got left mouse button click: ', showPosEvent(event))

def onRightClick(event):
    print('Right click: ', showPosEvent(event))

def onMiddleClick(event):
    print('Middle: ', showPosEvent(event))

def onLeftDrag(event):
    print('Left drag: ',showPosEvent(event))

def onDoubleLeftClick(event):
    print('Double click: ', showPosEvent(event), tkroot.quit())

tkroot = Tk()
labelfont=('courier', 20, 'bold')
widget = Label(tkroot, text='Hello binding')
widget.config(bg='red', font=labelfont)
widget.config(height=5, width=20)
widget.pack(expand=YES, fill=BOTH)

widget.bind('<Button-1>', onLeftClick)
widget.bind('<Button-3>', onRightClick)
widget.bind('<Button-2>', onMiddleClick)
widget.bind('<Double-1>', onDoubleLeftClick)
widget.bind('<B1-Motion>', onLeftDrag)

widget.bind('<KeyPress>', onKeyPress)
widget.bind('<Up>', onArrowKey)
widget.bind('Return>', onReturnKey)
widget.focus()
tkroot.title('Click Me')
tkroot.mainloop()
```



Binding events

All key binding events are case sensitive. Use the bindings shown below, don’t capitalise bindings like <KEY-A>, Python won’t understand them.

Mouse Bindings

<Button-1> catches the click of the left mouse button.
<Button-2> catches the click of the middle mouse button.
<Button-3> catches the click of the right mouse button.

<B1-Motion> catches the mouse movement while the left mouse button is held down.
<B2-Motion> catches the mouse movement while the middle mouse button is held down.
<B3-Motion> catches the mouse movement while the right mouse button is held down.

<B1- ButtonPress> fires when the left button is pressed down.
<B2- ButtonPress> fires when the middle button is pressed down.
<B3- ButtonPress> fires when the right button is pressed down.

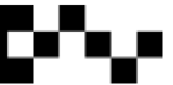
<B1- ButtonRelease> fires when the left button is released.
<B2- ButtonRelease> fires when the middle button is released.
<B3- ButtonRelease> fires when the right button is released.

<B1-Enter> and <B1-Leave> intercept the mouse entry and exit in a window’s display area (useful for automatically highlighting a widget)
<B2-Enter> and <B2-Leave>
<B3-Enter> and <B3-Leave>

Keyboard Bindings

<KeyPress> catches a single key of the keyboard.
<Escape>, <Backspace> and <tab> catch other special keys.
<Up>, <Down>, <Left>, <Right> catch arrow key presses.

<Return> catches the press of the ENTER key on the keyboard
<Key-1> - <Key-0> catches the number keys
<Key-a> - <Key-z> catches the alphabetical keys



List box with scroll bar

```
from tkinter import *

s = Scrollbar()
l = Listbox()

s.pack(side=RIGHT, fill=Y)
l.pack(side=LEFT, fill=Y)

s.config(command=l.yview)
l.config(yscrollcommand=s.set)

for i in range(30):
   l.insert(END, str(i)*3)

mainloop()
```