*Python\_Tkinter\_Programming\_c01*

*Python and Tkinter Programming*

JOHN E. GRAYSON

p26

**PART** 1 *Basic concepts*

In part 1, I’ll introduce Python, Tkinter and application programming. Since I assume you’re already somewhat familiar with Python, chapter 1 is intended to illustrate the most important features of the language that will be used throughout the book. Additionally, I’ll discuss features of Python’s support for object-oriented programming so that those of you familiar with C++ or Java can under- stand how your experience may be applied to Python.

Chapter 2 quickly introduces Tkinter and explains how it relates to Tcl/Tk. You will find details of mapping Tk to Tkinter, along with a brief introduction to the widgets and their appearance.

Chapter 3 illustrates application development with Tkinter using two calculator examples. The first is a simple no-frills calculator that demonstrates basic principles. The second is a partially finished application that shows you how powerful applications may be developed using Python’s and Tkinter’s capabilities.

**CHAPTER 1**

*Python*

1.1 Introduction to Python programming and a feature review 3

1.2 Key data types: lists, tuples and dictionaries 5

1.3 Classes 9

This chapter defines the key features of Python that make the language ideal for rapid proto- typing of systems and for fully-functional applications. *Python and Tkinter Programming* is not intended to be a learning resource for beginning Python programmers; several other publications are better-suited to this task: *Quick Python*, *Learning Python, Programming Python*, *Internet Programming in Python* and *The Python Pocket Reference* are all excellent texts. Further information is provided in the “References” section at the end of this book. In this chapter, the key features of Python will be highlighted in concise examples of code to illustrate some of the building blocks that will be used in examples throughout the book.

Este capítulo define las características clave de Python que hacen que el lenguaje sea ideal para la creación rápida de prototipos de sistemas y para aplicaciones completamente funcionales. La programación de Python y Tkinter no pretende ser un recurso de aprendizaje para programadores principiantes de Python; Varias otras publicaciones se adaptan mejor a esta tarea: Quick Python, Learning Python, Programming Python, Internet Programming in Python y The Python Pocket Reference son todos textos excelentes. Se proporciona más información en la sección "Referencias" al final de este libro. En este capítulo, las características clave de Python se resaltarán en ejemplos concisos de código para ilustrar algunos de los componentes básicos que se utilizarán en los ejemplos a lo largo del libro.

**1.1 Introduction to Python programming and a feature review**

As stated earlier, this book is not intended to be used to learn Python basics directly. Pro- grammers experienced in other languages will be able to analyze the examples and discover the key points to programming in Python. However, if you are relatively new to program- ming generally, then learning Python this way will be a tough, upward struggle.

This chapter is really not necessary for most readers, then, since the material will already be familiar. Its purpose is to provide a refresher course for readers who worked with Python in the early days and a map for Tcl/Tk programmers and those readers experienced with other languages.

Como se indicó anteriormente, este libro no está destinado a aprender los conceptos básicos de Python directamente. Los programadores con experiencia en otros lenguajes podrán analizar los ejemplos y descubrir los puntos clave de la programación en Python. Sin embargo, si usted es relativamente nuevo en la programación en general, entonces aprender Python de esta manera será una lucha difícil y ascendente. Entonces, este capítulo realmente no es necesario para la mayoría de los lectores, ya que el material ya les resultará familiar. Su propósito es proporcionar un curso de actualización para los lectores que trabajaron con Python en los primeros días y un mapa para los programadores de Tcl/Tk y aquellos lectores con experiencia en otros lenguajes.

Readers unfamiliar with object-oriented programming (OOP) may find section 1.3 use- ful as an introduction to OOP as it is implemented in Python. C++ or Java programmers who need to see how Python’s classes operate will benefit as well.

Los lectores que no estén familiarizados con la programación orientada a objetos (POO) pueden encontrar útil la sección 1.3 como introducción a la POO tal como se implementa en Python. Los programadores de C++ o Java que necesiten ver cómo funcionan las clases de Python también se beneficiarán.

I’m not going to explain the reasons why Python was developed or when, since this infor- mation is covered in every other Python book very well. I will state that Guido van Rossum, Python’s creator, has been behind the language since he invented it at Stichting Mathematisch Centrum (CWI) in Amsterdam, The Nederlands, around 1990; he is now at the Corporation for National Research Initiatives (CNRI), Reston, Virginia, USA. The fact that one person has taken control of the growth of the language has had a great deal to do with its stability and elegance, although Guido will be the first to thank all of the people who have contributed, in one way or another, to the language’s development.

No voy a explicar las razones por las que se desarrolló Python ni cuándo, ya que esta información se trata muy bien en todos los demás libros de Python. Diré que Guido van Rossum, el creador de Python, ha estado detrás del lenguaje desde que lo inventó en Stichting Mathematisch Centrum (CWI) en Ámsterdam, Países Bajos, alrededor de 1990; Actualmente trabaja en la Corporación para Iniciativas Nacionales de Investigación (CNRI), Reston, Virginia, EE.UU. El hecho de que una persona se haya hecho cargo del crecimiento de la lengua ha tenido mucho que ver con su estabilidad y elegancia, aunque Guido será el primero en agradecer a todas las personas que han contribuido, de una forma u otra, a el desarrollo del lenguaje.

Perhaps more important than any of the above information is the name of the language. This language has nothing to do with snakes. Python is named after *Monty Python’s Flying Cir- cus*, the BBC comedy series which was produced from 1969 to 1974. Like many university stu- dents around 1970, I was influenced by Monty Python, so when I started writing this book I could not resist the temptation to add bits of Python other than the language. Now, all of you that skipped the boring beginning bit of this book, or decided that you didn’t need to read this paragraph are in for a surprise. Scattered through the examples you’ll find bits of Python. If you have never experienced Monty Python, then I can only offer the following advice: if some- thing about the example looks weird, it’s probably Python. As my Yugoslavian college friend used to say “You find *that* funny”?

Quizás más importante que cualquiera de la información anterior sea el nombre del idioma. Este idioma no tiene nada que ver con las serpientes. Python lleva el nombre de Monty Python's Flying Circus, la serie de comedia de la BBC que se produjo entre 1969 y 1974. Como muchos estudiantes universitarios alrededor de 1970, fui influenciado por Monty Python, así que cuando comencé a escribir este libro no pude resistirme. la tentación de agregar partes de Python distintas al lenguaje. Ahora, todos aquellos que se saltaron el aburrido comienzo de este libro, o decidieron que no necesitaban leer este párrafo, se llevarán una sorpresa. Distribuidos por los ejemplos, encontrará fragmentos de Python. Si nunca ha experimentado Monty Python, sólo puedo ofrecerle el siguiente consejo: si algo en el ejemplo parece extraño, probablemente sea Python. Como solía decir mi amigo de la universidad yugoslavo: “¿Te parece gracioso?”

**1.1.1 Why Python?**

Several key features make Python an ideal language for a wide range of applications. Adding Tkinter to the mix widens the possibilities dramatically. Here are some of the highlights that make Python what it is:

Varias características clave hacen de Python un lenguaje ideal para una amplia gama de aplicaciones. Agregar Tkinter a la mezcla amplía dramáticamente las posibilidades. Éstos son algunos de los aspectos más destacados que hacen de Python lo que es:

* Automatic compile to bytecode
* High-level data types and operations
* Portability across architectures
* Wide (huge) range of supported extensions
* Object-oriented model
* Ideal prototyping system
* Readable code with a distinct C-like quality supports maintenance
* Easy to extend in C and C++ and embed in applications
* Large library of contributed applications and tools
* Excellent documentation  
     
  You might notice that I did not mention an interpreter explicitly. One feature of Python is that it is a bytecode engine written in C. The extension modules are written in C. With a little care in the way you design your code, most of your code will run using *compiled C* since many operations are built into the system. The remaining code will run in the bytecode engine.

Quizás notes que no mencioné explícitamente a un intérprete. Una característica de Python es que es un motor de código de bytes escrito en C. Los módulos de extensión están escritos en C. Con un poco de cuidado en la forma en que diseña su código, la mayor parte de su código se ejecutará utilizando C compilado, ya que muchas operaciones están integradas. el sistema. El código restante se ejecutará en el motor de código de bytes.

The result is a system that may be used as a scripting language to develop anything from some system administration scripts all the way to a complex GUI-based application (using database, client/server, CORBA or other techniques).

El resultado es un sistema que puede usarse como lenguaje de scripting para desarrollar cualquier cosa, desde algunos scripts de administración del sistema hasta una aplicación compleja basada en GUI (usando base de datos, cliente/servidor, CORBA u otras técnicas).

**1.1.2 Where can Python be used?**

Knowing where Python can be used is best understood by learning where it might *not* be the best choice. Regardless of what I just said about the bytecode engine, Python has an interpre- tive nature, so if you can’t keep within the C-extensions, there has to be a performance pen- alty. Therefore, real-time applications for high-speed events would be a poor match. A set of extensions to Python have been developed specifically for numerical programming (see “NumPy” on page 626). These extensions help support compute-bound applications, but Python is not the best choice for huge computation-intensive applications unless time *isn’t* a factor. Similarly, graphics-intensive applications which involve real-time observation are not a good match (but see “Speed drawing” on page 271 for an example of what *can* be done).

La mejor manera de comprender dónde se puede utilizar Python es saber dónde podría no ser la mejor opción. Independientemente de lo que acabo de decir sobre el motor de código de bytes, Python tiene una naturaleza interpretativa, por lo que si no puedes mantenerte dentro de las extensiones C, tiene que haber una penalización en el rendimiento. Por lo tanto, las aplicaciones en tiempo real para eventos de alta velocidad no serían una buena opción. Se ha desarrollado un conjunto de extensiones de Python específicamente para programación numérica (consulte “NumPy” en la página 626). Estas extensiones ayudan a admitir aplicaciones vinculadas a la computación, pero Python no es la mejor opción para aplicaciones de gran uso intensivo de computación, a menos que el tiempo no sea un factor. De manera similar, las aplicaciones con uso intensivo de gráficos que implican observación en tiempo real no son una buena opción (pero consulte “Dibujo rápido” en la página 271 para ver un ejemplo de lo que se puede hacer).

**1.2 Key data types: lists, tuples and dictionaries**

Three key data types give Python the power to produce effective applications: two *sequence* classes—lists and tuples—and a *mapping* class—dictionaries. When they are used together, they can deliver surprising power in a few lines of code.

Tres tipos de datos clave le dan a Python el poder de producir aplicaciones efectivas: dos clases de secuencia (listas y tuplas) y una clase de mapeo (diccionarios). Cuando se usan juntos, pueden ofrecer un poder sorprendente en unas pocas líneas de código.

*Lists* and *tuples* have a lot in common. The major difference is that the elements of a list can be modified in place but a tuple is *immutable*: you have to deconstruct and then reconstruct a tuple to change individual elements. There are several good reasons why we should care about this distinction; if you want to use a tuple as the *key* to a dictionary, it’s good to know that it can’t be changed arbitrarily. A small advantage of tuples is that they are a slightly cheaper resource since they do not carry the additional operations of a list.

Las listas y tuplas tienen mucho en común. La principal diferencia es que los elementos de una lista se pueden modificar en el lugar, pero una tupla es inmutable: hay que deconstruir y luego reconstruir una tupla para cambiar elementos individuales. Hay varias buenas razones por las que deberíamos preocuparnos por esta distinción; Si desea utilizar una tupla como clave de un diccionario, es bueno saber que no se puede cambiar arbitrariamente. Una pequeña ventaja de las tuplas es que son un recurso ligeramente más económico ya que no llevan a cabo las operaciones adicionales de una lista.

If you want an in-depth view of these data types take a look at chapters 6 and 8 of *Quick Python*.

Si desea una visión en profundidad de estos tipos de datos, consulte los capítulos 6 y 8 de Quick Python.

**1.2.1 Lists**

Let’s look at lists first. If you are new to Python, remember to look at the tutorial that is avail- able in the standard documentation, which is available at www.python.org.

Veamos primero las listas. Si es nuevo en Python, recuerde consultar el tutorial que está disponible en la documentación estándar, que está disponible en www.python.org.

**Initializing lists**

Lists are easy to create and use. To initialize a list:

lst = []

lst = ['a', 'b', 'c']

lst = [1, 2, 3, 4]

lst = [[1,2,3], ['a','b','c']]

lst = [(1,'a'),(2,'b'),(3,'c')]

**Appending to lists**

Lists have an append method built in:

lst.append('e')

lst.append((5,'e'))

**Concatenating lists**

**Combining lists works well:**

lst = [1, 2, 3] + [4, 5, 6]

print lst

**[1, 2, 3, 4, 5, 6]**

**Iterating through members**

Iterating through a list is easy:

lst = ['first', 'second', 'third']

for str in lst:

print 'this entry is %s' % str

set = [(1, 'uno'), (2, 'due'), (3, 'tres')]

for integer, str in set:

print 'Numero "%d" in Italiano: è "%s"' % (integer, str)

Sorting and reversing

Lists have built-in sort and reverse methods:

lst = [4, 5, 1, 9, 2]

lst.sort()

print lst

[1, 2, 4, 5, 9]

lst.reverse() print lst

**[9, 5, 4, 2, 1]**

Indexing

Finding an entry in a list:

lst = [1, 2, 4, 5, 9] print lst.index(5)

**3**

Member

Checking membership of a list is convenient:

if 'jeg' in ['abc', 'tuv', 'kie', 'jeg']:

...

if '\*' in '123\*abc':

...

Modifying members

A list member may be modified in place:

lst = [1, 2, 4, 5, 9] lst[3] = 10

print lst

**[1, 2, 4, 10, 9]**

Inserting and deleting members

To insert a member in a list:

lst = [1, 2, 3, 4, 10, 9] lst.insert(4, 5)

print lst

**[1, 2, 3, 4, 5, 10, 9]**

To delete a member:

lst = [1, 2, 3, 4, 10, 9] del lst(4)

print lst

**[1, 2, 3, 4, 9]**

**1.2.2 Tuples**

Tuples are similar to lists but they are *immutable* (meaning they cannot be modified). Tuples are a convenient way of collecting data that may be passed as a single entity or stored in a list or dictionary; the entity is then *unpacked* when needed.

**Initializing tuples**

With the exception of a tuple containing *one* element, tuples are initialized in a similar man- ner to lists (lists and tuples are really related sequence types and are readily interchangeable).

tpl = ()

tpl = **(1,)**

tpl = ('a', 'b', 'c')

tpl = (1, 2, 3, 4)

tpl = ([1,2,3], ['a','b','c']) tpl = ((1,'a'),(2,'b'),(3,'c'))

**Iterating through members**

for i in tpl:

...

# Empty tuple

# Singleton tuple

# String tuple

# Integer tuple

# Tuple of lists

# Tuple of tuples

for i,a in ((1, 'a'), (2, 'b'), (3, 'c')):

...

Modifying tuples

(But you said tuples were immutable!)

a = 1, 2, 3

a = a[0], a[1], 10, a[2] a

**(1, 2, 10, 3)**

Note that you are not modifying the original tuple but you are creating a new name bind- ing for a.

**1.2.3 Dictionaries**

*Dictionaries* are arrays of data indexed by *keys*. I think that they give Python the edge in designing compact systems. If you use lists and tuples as data contained within dictionaries you have a powerful mix (not to say that mixing code objects, dictionaries and abstract objects isn’t powerful!).

**Initializing dictionaries**

Dictionaries may be initialized by providing key:value pairs:

dict = {}

dict = {'a'': 1, 'b': 2, 'c': 3}

dict = {1: 'a', 2: 'b', 3: 'c'}

dict = {1: [1,2,3], 2: [4,5,6]}

Modifying dictionaries

Dictionaries are readily modifiable:

dict['a'] = 10

dict[10] = 'Larch'

Accessing dictionaries

# Empty dictionary

# String key

# Integer key

# List data

Recent versions of Python facilitate lookups where the key may not exist. First, the old way:

or:

if dict.has\_key('a'):

value = dict['a']

else:

value = None

try:

value = dict['a']

except KeyError:

value = None

This is the current method:

value = dict.get('a', None)

Iterating through entries

Get the keys and then iterate through them:

keys = dict.keys()

for key in keys:

...

Sorting dictionaries

Dictionaries have arbitrary order so you must sort the keys if you want to access the keys in order:

keys = dict.keys().sort()

for key in keys:

**1.3 Classes**

I’m including a short section on Python classes largely for C++ programmers who may need to learn some of the details of Python’s implementation and for Python programmers who have yet to discover OOP in Python.

Incluyo una breve sección sobre clases de Python principalmente para programadores de C++ que puedan necesitar aprender algunos de los detalles de la implementación de Python y para programadores de Python que aún no han descubierto la programación orientada a objetos en Python.

**1.3.1 How do classes describe objects?**

A class provides the following object descriptions:

* The attributes (data-members) of the object
* The behavior of the object (methods)
* Where behavior is inherited from other classes (superclasses)  
     
  Having said all that, C++ programmers will probably be tuning out at this point—but hold on for a little longer. There are some valuable features of Python classes, some of which may come as a bit of a surprise for someone who is not fully up to speed with Python OOP.

Most of the examples of applications in this book rely heavily on building class libraries to create a wide range of objects. The classes typically create instances with multiple formats (see LEDs and Switches in chapter 7). Before we start building these objects, let’s review the rules and features that apply to Python classes.

**1.3.2 Defining classes**    
A Python class is a user-defined data type which is defined with a class statement:  
   
 class AClass:

statements

*Statements* are any valid Python statements defining attributes and member functions. In fact, any Python statement can be used, including a pass statement, as we will see in the next section. Calling the class as a function creates an instance of the class:  
   
 anInstanceOfAClass = AClass()



**1.3.3 Neat Python trick #10**    
A class instance can be used like a C structure or Pascal record. However, unlike C and Pascal, the members of the structure do not need to be declared before they are used—they can be created dynamically. We can use this ability to access arbitrary data objects across modules; examples using class instances to support global data will be shown later.  
   
 class DummyClass:

pass

Colors = DummyClass()

Colors.alarm = 'red'

Colors.warning = 'orange'

Colors.normal = 'green'

* If the preceding lines are stored in a file called programdata.py, the following is a possible code sequence.

from programdata import Colors

...

Button(parent, bg=Colors.alarm, text='Pressure\nVessel',

command=evacuateBuilding)

Alternately, if you apply a little knowledge about how Python manages data internally, you can use the following construction.

class Record:

def \_\_init\_\_(self, \*\*kw):

self.\_\_dict\_\_.update(kw)

Colors = Record(alarm='red', warning='orange', normal='green')

**1.3.4 Initializing an instance**

Fields (instance variables) of an instance may be initialized by including an \_\_init\_\_ method in the class body. This method is executed automatically when a new instance of the class is created. Python passes the instance as the first argument. It is a convention to name it self (it’s called *this* in C++). In addition, methods may be called to complete initialization. The \_\_init\_\_ methods of inherited classes may also be called, when necessary.

class ASX200(Frame):

def \_\_init\_\_(self, master=None):

Frame.\_\_init\_\_(self, master)

Pack.config(self)

self.state = NORMAL

self.set\_hardware\_data(FORE)

self.createWidgets()

...

...

switch = ASX200()

*I*

Note Touseinstancevariablesyoumustreferencethecontainingobject(intheprevi- ous example it is switch.state, not self.state). If you make a reference to a variable by itself, it is to a local variable within the executing function, not an instance

variable.

**1.3.5 Methods**

We have already encountered the \_\_init\_\_ method that is invoked when an instance is cre- ated. Other methods are defined similarly with def statements. Methods may take argu- ments: self is always the first or only argument.

You will see plenty of examples of methods, so little discussion is really necessary. Note that Python accepts named arguments, in addition to positional arguments, in both methods and function calls. This can make supplying default values for methods very easy, since omis- sion of an argument will result in the default value being supplied. Take care when mixing posi- tional and named arguments as it is very easy to introduce problems in class libraries this way.

**1.3.6 Private and public variables and methods**

Unless you take special action, all variables and methods are public and virtual. If you make use of name mangling, however, you can emulate private variables and methods. You mangle the name this way: Any name which begins with a double-underscore (\_\_) is private and is not exported to a containing environment. Any name which begins with a single underscore (\_) indicates *private by convention*, which is similar to *protected* in C++ or Java. In fact, Python usually is more intuitive than C++ or other languages, since it is immediately obvious if a ref- erence is being made to a private variable or method.

**1.3.7 Inheritance**

**1.3.8**

**1.3.9**

The rules of inheritance in Python are really quite simple:

* Classes inherit behavior from the classes specified in their header and from any classes above these classes.
* Instances inherit behavior from the class from which they are created and from all the classes above this class.  
     
  When Python searches for a reference it searches in the immediate namespace (the instance) and then in each of the higher namespaces. The first occurrence of the reference is used; this means that a class can easily redefine attributes and methods of its superclasses. If the reference cannot be found Python reports an error.  
     
  Note that inherited methods are not automatically called. To initialize the base class, a subclass must call the \_\_init\_\_ method explicitly.  
     
  **Multiple inheritance**    
  Multiple inheritance in Python is just an extension of inheritance. If more than one class is specified in a class’s header then we have multiple inheritance. Unlike C++, however, Python does not report errors if attributes of classes are multiple defined; the basic rule is that the first occurrence found is the one that is used.  
     
  **Mixin classes**    
  A class that collects a number of common methods and can be freely inherited by subclasses is usually referred to as a *mixin* class (some standard texts may use base, generalized or abstract classes, but that may not be totally correct). Such methods could be contained in a Python module, but the advantage of employing a mixin class is that the methods have access to the instance self and thus can modify the behavior of an instance. We will see examples of mixin classes throughout this book.