

Michael Driscoll

Python 101

Michael Driscoll

This book is for sale at http://leanpub.com/python_101

This version was published on 2016-10-27



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2014 Michael Driscoll

Also By Michael Driscoll

[Python 201](#)

[wxPython Cookbook](#)

Contents

Introduction	1
A Brief History of Python	3
About the Author	3
Conventions	4
Requirements	4
Reader Feedback	4
Errata	5
Part I - Learning the Basics	6
Chapter 1 - IDLE Programming	8
Using IDLE	8
Other Tips	12
Wrapping Up	13
Chapter 2 - All About Strings	14
How to Create a String	14
String Concatenation	16
String Methods	16
String Slicing	18
String Formatting	19
Wrapping Up	23
Chapter 3 - Lists, Tuples and Dictionaries	24
Lists	24
Tuples	26
Dictionaries	26
Wrapping Up	28
Chapter 4 - Conditional Statements	29
The if statement	29
Boolean Operations	31
Checking for Nothing	32
Special Characters	34
if __name__ == "__main__"	35

CONTENTS

Wrapping Up	35
Chapter 5 - Loops	36
The for Loop	36
The while Loop	38
What else is for in loops	40
Wrapping Up	41
Chapter 6 - Python Comprehensions	42
List Comprehensions	42
Dictionary Comprehensions	43
Set Comprehensions	44
Wrapping Up	44
Chapter 7 - Exception Handling	45
Common Exceptions	45
How to Handle Exceptions	46
The finally Statement	48
try, except, or else!	49
Wrapping Up	50
Chapter 8 - Working with Files	51
How to Read a File	51
How To Read Files Piece by Piece	52
How to Read a Binary File	53
Writing Files in Python	53
Using the with Operator	54
Catching Errors	54
Wrapping Up	55
Chapter 9 - Importing	56
import this	56
Using from to import	57
Importing Everything!	57
Wrapping Up	58
Chapter 10 - Functions	59
An Empty Function (the stub)	59
Passing Arguments to a Function	59
Keyword Arguments	60
*args and **kwargs	62
A Note on Scope and Globals	62
Coding Tips	63
Wrapping Up	64

CONTENTS

Chapter 11 - Classes	65
Creating a Class	65
What is self?	67
Subclasses	69
Wrapping Up	70
Part II - Learning from the Library	71
Chapter 12 - Introspection	73
The Python Type	73
The Python Dir	74
Python Help!	75
Wrapping Up	75
Chapter 13 - The csv Module	76
Reading a CSV File	76
Writing a CSV File	77
Wrapping Up	80
Chapter 14 - configparser	81
Creating a Config File	81
How to Read, Update and Delete Options	82
How to Use Interpolation	85
Wrapping Up	85
Chapter 15 - Logging	86
Creating a Simple Logger	86
How to log From Multiple Modules (and Formatting too!)	87
Configuring Logs for Work and Pleasure	90
Wrapping Up	93
Chapter 16 - The os Module	94
os.name	94
os.environ, os.getenv() and os.putenv()	95
os.chdir() and os.getcwd()	97
os.mkdir() and os.makedirs()	97
os.remove() and os.rmdir()	98
os.rename(src, dst)	98
os.startfile()	99
os.walk()	99
os.path	100
os.path.basename	100
os.path.dirname	100
os.path.exists	101

CONTENTS

os.path.isdir / os.path.isfile	101
os.path.join	102
os.path.split	102
Wrapping Up	103
Chapter 17 - The email / smtplib Module	104
Email Basics - How to Send an Email with smtplib	104
Sending Multiple Emails at Once	107
Send email using the TO, CC and BCC lines	109
Add an attachment / body using the email module	110
Wrapping Up	112
Chapter 18 - The sqlite Module	113
How to Create a Database and INSERT Some Data	113
Updating and Deleting Records	114
Basic SQLite Queries	115
Wrapping Up	116
Chapter 19 - The subprocess Module	117
The call function	117
The Popen Class	119
Learning to Communicate	119
Wrapping Up	121
Chapter 20 - The sys Module	122
sys.argv	122
sys.executable	123
sys.exit	123
sys.path	125
sys.platform	125
sys.stdin / stdout / stderr	126
Wrapping Up	126
Chapter 21 - The threading module	127
Using Threads	127
Writing a Threaded Downloader	128
Using Queues	131
Wrapping Up	133
Chapter 22 - Working with Dates and Time	135
The datetime Module	135
datetime.date	135
datetime.datetime	136
datetime.timedelta	137

CONTENTS

The time Module	138
time.ctime	138
time.sleep	139
time.strftime	139
time.time	140
Wrapping Up	140
Chapter 23 - The xml module	141
Working with minidom	141
Parsing with ElementTree	146
How to Create XML with ElementTree	146
How to Edit XML with ElementTree	148
How to Parse XML with ElementTree	150
Wrapping Up	152
Part III - Intermediate Odds and Ends	153
Chapter 24 - The Python Debugger	155
How to Start the Debugger	155
Stepping Through the Code	157
Setting breakpoints	158
Wrapping Up	158
Chapter 25 - Decorators	159
A Simple Function	159
Creating a Logging Decorator	161
Built-in Decorators	162
@classmethod and @staticmethod	163
Python Properties	164
Replacing Setters and Getters with a Python property	165
Wrapping Up	169
Chapter 26 - The lambda	170
Tkinter + lambda	170
Wrapping Up	171
Chapter 27 - Code Profiling	173
Profiling Your Code with cProfile	173
Wrapping Up	176
Chapter 28 - An Intro to Testing	177
Testing with doctest	177
Running doctest via the Terminal	177
Running doctest Inside a Module	179
Running doctest From a Separate File	180

CONTENTS

Test Driven Development with unittest	182
The First Test	183
The Second Test	185
The Third (and Final) Test	187
Other Notes	189
Wrapping Up	189
Part IV - Tips, Tricks and Tutorials	190
Chapter 29 - Installing Modules	192
Installing from Source	192
Using easy_install	193
Using pip	193
A Note on Dependencies	194
Wrapping Up	194
Chapter 30 - ConfigObj	195
Getting Started	195
Using a configspec	196
Wrapping Up	198
Chapter 31 - Parsing XML with lxml	199
Parsing XML with lxml	200
Parsing the Book Example	201
Parsing XML with lxml.objectify	202
Creating XML with lxml.objectify	205
Wrapping Up	208
Chapter 32 - Python Code Analysis	209
Getting Started with pylint	209
Analyzing Your Code	209
Getting Started with pyflakes	212
Wrapping Up	213
Chapter 33 - The requests package	214
Using requests	214
How to Submit a Web Form	215
Wrapping Up	216
Chapter 34 - SQLAlchemy	217
How to Create a Database	217
How to Insert / Add Data to Your Tables	220
How to Modify Records with SQLAlchemy	222
How to Delete Records in SQLAlchemy	223
The Basic SQL Queries of SQLAlchemy	223

CONTENTS

Wrapping Up	225
Chapter 35 - virtualenv	226
Installation	226
Creating a Virtual Environment	226
Wrapping Up	227
Part V - Packaging and Distribution	228
Chapter 36 - Creating Modules and Packages	230
How to Create a Python Module	230
How to Create a Python Package	232
Wrapping Up	234
Chapter 37 - How to Add Your Code to PyPI	235
Creating a setup.py File	235
Registering Packages	236
Uploading Packages to PyPI	237
Wrapping Up	238
Chapter 38 - The Python egg	239
Creating an egg	239
Wrapping Up	240
Chapter 39 - Python wheels	241
Getting Started	241
Creating a wheel	241
Installing a Python wheel	242
Wrapping Up	243
Chapter 40 - py2exe	244
Creating a Simple GUI	244
The py2exe setup.py file	246
Creating an Advanced setup.py File	248
Wrapping Up	249
Chapter 41 - bbfreeze	250
Getting Started with bbfreeze	250
Using bbfreeze's Advanced Configuration	252
Wrapping Up	255
Chapter 42 - cx_Freeze	256
Getting Started with cx_Freeze	256
Advanced cx_Freeze - Using a setup.py File	258
Wrapping Up	260

CONTENTS

Chapter 43 - PyInstaller	261
Getting Started with PyInstaller	261
PyInstaller and wxPython	263
Wrapping Up	266
Chapter 44 - Creating an Installer	267
Getting Started with GUI2Exe	268
Let's Make an Installer!	270
Wrapping Up	273
Appendix A: Putting It All Together	275
The Background	275
The Specification	275
Breaking the Specification Down	276
Turning Chapters Into a Book	276
Reading the Customer CSV File	278
Emailing the PDF	279
Putting it all Together	281
Wrapping Up	284

Introduction

Welcome to Python 101! I wrote this book to help you learn Python 3. It is not meant to be an exhaustive reference book. Instead, the object is to get you acquainted with the building blocks of Python so that you can actually write something useful yourself. A lot of programming textbooks only teach you the language, but do not go much beyond that. I will endeavour to not only get you up to speed on the basics, but also to show you how to create useful programs. Now you may be wondering why just learning the basics isn't enough. In my experience, when I get finished reading an introductory text, I want to then create something, but I don't know how! I've got the learning, but not the glue to get from point A to point B. I think it's important to not only teach you the basics, but also cover intermediate material.

Thus, this book will be split into five parts:

- Part one will cover Python's basics
- Part two will be on a small subset of Python's Standard Library
- Part three will be intermediate material
- Part four will be a series of small tutorials
- Part five will cover Python packaging and distribution

|

Let me spend a few moments explaining what each part has to offer. In part one, we will cover the following:

- Python types (strings, lists, dicts, etc)
- Conditional statements
- Loops
- List and dictionary comprehensions
- Exception Handling
- File I/O
- Functions and Classes

|

Part two will talk about some of Python's standard library. The standard library is what comes pre-packaged with Python. It is made up of modules that you can import to get added functionality. For example, you can import the `math` module to gain some high level math functions. I will be cherry picking the modules I use the most as a day-to-day professional and explaining how they work. The reason I think this is a good idea is that they are common, every day modules that I think you will benefit knowing about at the beginning of your Python education. This section will also cover

various ways to install 3rd party modules. Finally, I will cover how to create your own modules and packages and why you'd want to do that in the first place. Here are some of the modules we will be covering:

- csv
- ConfigParser
- logging
- os
- smtplib / email
- subprocess
- sys
- thread / queues
- time / datetime

|

Part three will cover intermediate odds and ends. These are topics that are handy to know, but not necessarily required to be able to program in Python. The topics covered are:

- the Python debugger (pdb)
- decorators
- the lambda function
- code profiling
- a testing introduction

|

Part four will be made up of small tutorials that will help you to learn how to use Python in a practical way. In this way, you will learn how to create Python programs that can actually do something useful! You can take the knowledge in these tutorials to create your own scripts. Ideas for further enhancements to these mini-applications will be provided at the end of each tutorial so you will have something that you can try out on your own. Here are a few of the 3rd party packages that we'll be covering:

- pip and easy_install
- configobj
- lxml
- requests
- virtualenv
- pylint / pychecker
- SQLAlchemy

|

Part five is going to cover how to take your code and give it to your friends, family and the world! You will learn the following:

- How to turn your reusable scripts into Python “eggs”, “wheels” and more
- How to upload your creation to the Python Package Index (PyPI)
- How to create binary executables so you can run your application without Python
- How to create an installer for your application

|

The chapters and sections may not all be the same length. While every topic will be covered well, not every topic will require the same page count.

A Brief History of Python

I think it helps to know the background of the Python programming language. Python was created in the late 1980s¹. Everyone agrees that its creator is Guido van Rossum when he wrote it as a successor to the ABC programming language that he was using. Guido named the language after one of his favorite comedy acts: Monty Python. The language wasn't released until 1991 and it has grown a lot in terms of the number of included modules and packages included. At the time of this writing, there are two major versions of Python: the 2.x series and the 3.x (sometimes known as Python 3000) . The 3.x series is not backwards compatible with 2.x because the idea when creating 3.x was to get rid of some of the idiosyncrasies in the original. The current versions are 2.7.12 and 3.5.2. Most of the features in 3.x have been backported to 2.x; however, 3.x is getting the majority of Python's current development, so it is the version of the future.

Some people think Python is just for writing little scripts to glue together “real” code, like C++ or Haskell. However you will find Python to be useful in almost any situation. Python is used by lots of big name companies such as Google, NASA, LinkedIn, Industrial Light & Magic, and many others. Python is used not only on the backend, but also on the front. In case you're new to the computer science field, backend programming is the stuff that's behind the scenes; things like database processing, document generation, etc. Frontend processing is the pretty stuff most users are familiar with, such as web pages or desktop user interfaces. For example, there are some really nice Python GUI toolkits such as wxPython, PySide, and Kivy. There are also several web frameworks like Django, Pyramid, and Flask. You might find it surprising to know that Django is used for Instagram and Pinterest. If you have used these or many other websites, then you have used something that's powered by Python without even realizing it!

About the Author

You may be wondering about who I am and why I might be knowledgeable enough about Python to write about it, so I thought I'd give you a little information about myself. I started programming in Python in the Spring of 2006 for a job. My first assignment was to port Windows login scripts from

¹<http://www.artima.com/intv/pythonP.html>

Kixtart to Python. My second project was to port VBA code (basically a GUI on top of Microsoft Office products) to Python, which is how I first got started in wxPython. I've been using Python ever since, doing a variation of backend programming and desktop front end user interfaces.

I realized that one way for me to remember how to do certain things in Python was to write about them and that's how my Python blog came about: <http://www.blog.pythonlibrary.org/>. As I wrote, I would receive feedback from my readers and I ended up expanding the blog to include tips, tutorials, Python news, and Python book reviews. I work regularly with Packt Publishing as a technical reviewer, which means that I get to try to check for errors in the books before they're published. I also have written for the Developer Zone (DZone) and i-programmer websites as well as the Python Software Foundation. In November 2013, DZone published **The Essential Core Python Cheat Sheet** that I co-authored.

Conventions

As with most technical books, this one includes a few conventions that you need to be aware of. New topics and terminology will be in **bold**. You will also see some examples that look like the following:

```
1 >>> myString = "Welcome to Python!"
```

The `>>>` is a Python prompt symbol. You will see this in the Python **interpreter** and in IDLE. You will learn more about each of these in the first chapter. Other code examples will be shown in a similar manner, but without the `>>>`.

Requirements

You will need a working Python 3 installation. The examples should work in either Python 2.x or 3.x unless specifically marked otherwise. Most Linux and Mac machines come with Python already installed. However, if you happen to find yourself without Python, you can go download a copy from <http://python.org/download/>². There are up-to-date installation instructions on their website, so I won't include any installation instructions in this book. Any additional requirements will be explained later on in the book.

Reader Feedback

I welcome feedback about my writings. If you'd like to let me know what you thought of the book, you can send comments to the following address:

comments@pythonlibrary.org

²<http://python.org/download/>

Errata

I try my best not to publish errors in my writings, but it happens from time to time. If you happen to see an error in this book, feel free to let me know by emailing me at the following:

errata@pythonlibrary.org

Part I - Learning the Basics

In Part I, we will learn the basics of the Python programming language. This section of the book should get you ready to use all the building blocks of Python so that you will be ready to tackle the following sections confidently.

Let's go over what we'll be covering:

- IDLE
- Strings
- Lists, Dictionaries and Tuples
- Conditional statements
- Loops
- Comprehensions
- Exception Handling
- File I/O
- Importing modules and packages
- Functions
- Classes

The first chapter in this section will familiarize you with Python's built-in development environment that is known as IDLE. The next couple of chapters will go over some of Python's types, such as strings, lists, and dictionaries. After that we'll look at conditional statements in Python and looping using Python's **for** and **while** loops.

In the second half of this section, we will go into comprehensions, such as list and dictionary comprehensions. Then we'll look at Python's exception handling capabilities and how Python's file operations work. Next up is learning how to import pre-made modules and packages. The last two chapters cover Python functions and classes.

Let's get started!



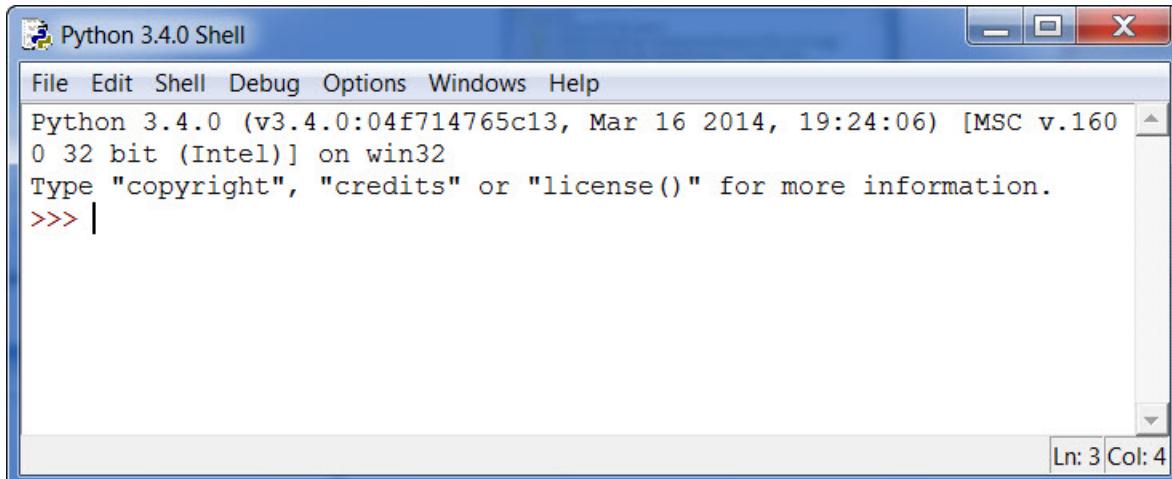
image

Chapter 1 - IDLE Programming

Using IDLE

Python comes with its own code editor: **IDLE**. There is some unconfirmed lore that the name for IDLE comes from Eric Idle, an actor in *Monty Python*. I have no idea if that's true or not, but it would make sense in this context as it appears to be a pun on the acronym IDE or Integrated Development Environment. An IDE is an editor for programmers that provides color highlighting of key words in the language, auto-complete, a debugger and lots of other fun things. You can find an IDE for most popular languages and a number of IDEs will work with multiple languages. IDLE is kind of a lite IDE, but it does have all those items mentioned. It allows the programmer to write Python and debug their code quite easily. The reason I call it "lite" is the debugger is very basic and it's missing other features that programmers who have a background using products like *Visual Studio* will miss. You might also like to know that IDLE was created using Tkinter, a Python GUI toolkit that comes with Python.

To open up IDLE, you will need to find it and you'll see something like this:



image

Yes, it's a Python shell where you can type short scripts and see their output immediately and even interact with code in real time. There is no compiling of the code as Python is an interpretive language and runs in the Python interpreter. Let's write your first program now. Type the following after the command prompt (>>>) in IDLE:

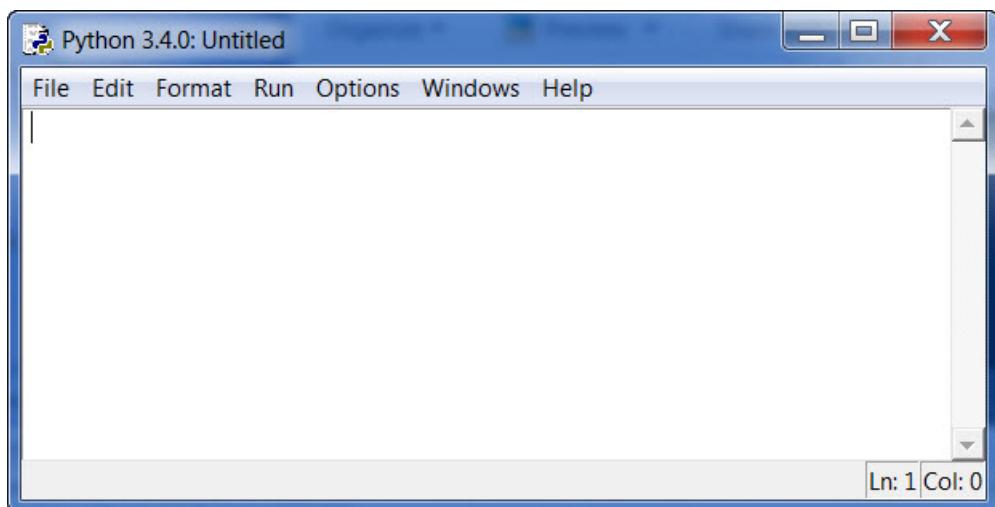
```
1 print("Hello from Python!")
```

You have just written your first program! All your program does is write a string to the screen, but you'll find that very helpful later on. Please note that the **print** statement has changed in Python 3.x. In Python 2.x, you would have written the above like this:

```
1 print "Hello from Python!"
```

In Python 3, the **print** statement was turned into a **print function**, which is why parentheses are required. You will learn what functions are in chapter 10.

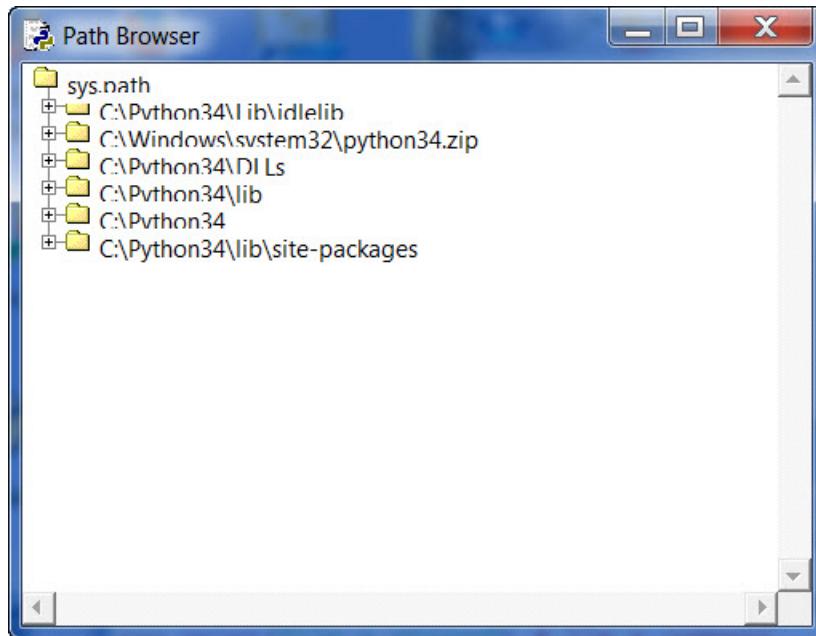
If you want to save your code into a file, go to the File menu and choose New Window (or press CTRL+N). Now you can type in your program and save it here. The primary benefit of using the Python shell is that you can experiment with small snippets to see how your code will behave before you put the code into a real program. The code editor screen looks a little different than the IDLE screenshot above:



image

Now we'll spend a little time looking at IDLE's other useful features.

Python comes with lots of modules and packages that you can import to add new features. For example, you can import the **math** module for all kinds of good math functions, like square roots, cosines, etcetera. In the **File** menu, you'll find a **Path Browser** which is useful for figuring out where Python looks for module imports. You see, Python first looks in the same directory as the script that is running to see if the file it needs to import is there. Then it checks a predefined list of other locations. You can actually add and remove locations as well. The Path Browser will show you where these files are located on your hard drive, if you have imported anything. My Path Browser looks like this:



image

Next there's a **Class Browser** that will help you navigate your code. This is actually something that won't be very useful to you right now, but will be in the future. You'll find it helpful when you have lots of lines of code in a single file as it will give you a "tree-like" interface for your code. Note that you won't be able to load the Class Browser unless you have actually saved your program.

The **Edit** menu has your typical features, such as Copy, Cut, Paste, Undo, Redo and Select All. It also contains various ways to search your code and do a search and replace. Finally, the Edit menu has some menu items that will Show you various things, such as highlighting parentheses or displaying the auto-complete list.

The **Format** menu has lots of useful functionality. It has some helpful items for **indenting** and **dedenting** your code, as well as commenting out your code. I find that pretty helpful when I'm testing my code. Commenting out your code can be very helpful. One way it can be helpful is when you have a lot of code and you need to find out why it's not working correctly. Commenting out portions of it and re-running the script can help you figure out where you went wrong. You just go along slowly uncommenting out stuff until you hit your bug. Which reminds me; you may have noticed that the main IDLE screen has a **Debugger** menu. That is nice for debugging, but only in the **Shell** window. Sadly you cannot use the debugger in your main editing menu. If you need a more versatile debugger, you should either find a different IDE or try Python's debugger found in the **pdb** library.

What are Comments?

A comment is a way to leave un-runnable code that documents what you are doing in your code. Every programming language uses a different symbol to demarcate where a comment starts and ends. What do comments look like in Python though? A comment is anything that begins with an octothorpe (i.e. a hash or pound sign). The following is an example of some comments in action:

```
1 # This is a comment before some code
2 print("Hello from Python!")
3 print("Winter is coming") # this is an in-line comment
```

You can write comments on a line all by themselves or following a statement, like the second `print` statement above. The Python interpreter ignores comments, so you can write anything you want in them. Most programmers I have met don't use comments very much. However, I highly recommend using comments liberally not just for yourself, but for anyone else who might have to maintain or enhance your code in the future. I have found my own comments useful when I come back to a script that I wrote 6 months ago and I have found myself working with code that didn't have comments and wishing that it did so I could figure it out faster.

Examples of good comments would include explanations about complex code statements, or adding an explanation for acronyms in your code. Sometimes you'll need to leave a comment to explain why you did something a certain way because it's just not obvious.

Now we need to get back to going over the menu options of IDLE:

The **Run** menu has a couple of handy options. You can use it to bring up the Python Shell, check your code for errors, or run your code. The Options menu doesn't have very many items. It does have a Configure option that allows you to change the code highlighting colors, fonts and key shortcuts. Other than that, you get a Code Context option that is helpful in that it puts an overlay in the editing window which will show you which class or function you're currently in. We will be explaining functions and classes near the end of Part I. You will find this feature is useful whenever you have a lot of code in a function and the name has scrolled off the top of the screen. With this option enabled, that doesn't happen. Of course, if the function is too large to fit on one screen, then it may be getting too long and it could be time to break that function down into multiple functions.

The **Windows** menu shows you a list of currently open Windows and allows you to switch between them.

Last but not least is the **Help** menu where you can learn about IDLE, get help with IDLE itself or load up a local copy of the Python documentation. The documentation will explain how each piece of Python works and is pretty exhaustive in its coverage. The Help menu is probably the most helpful in that you can get access to the docs even when you're not connected to the internet. You can search

the documentation, find HOWTOs, read about any of the builtin libraries, and learn so much your head will probably start spinning.

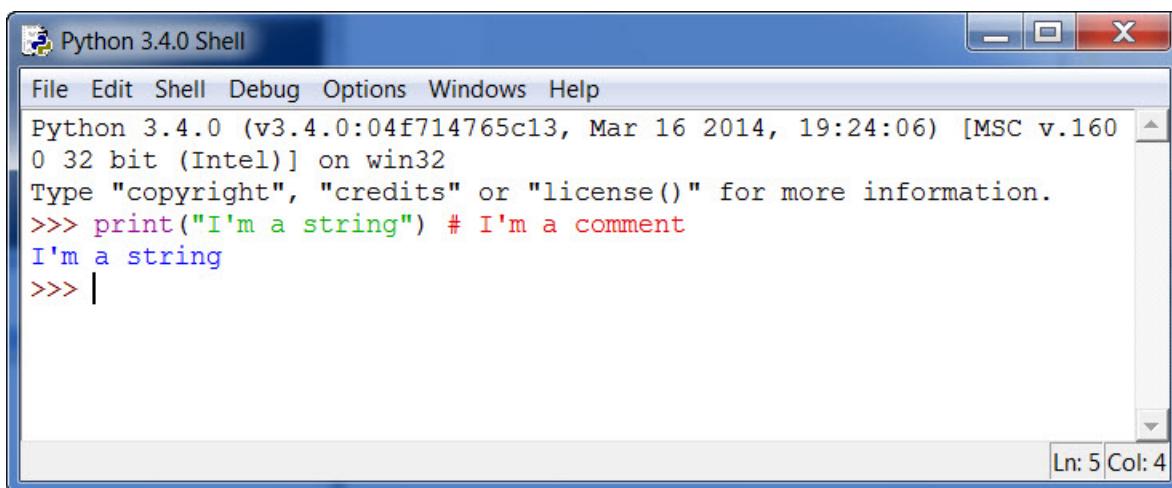
Other Tips

When you see code examples in the following chapters, you can write and run them in IDLE. I wrote all my programs in IDLE for the first couple of years of my Python programming life and I was pretty happy with it. There are lots of free Python IDEs out there though and several IDEs that you have to pay for. If you want to go cheap, you might want to take a look at Eclipse+PyDev, Editra or even Notepad++. For a paid IDE, I would recommend WingWare's IDE or possibly PyCharm. They have many more features such as integration with code repositories, better debuggers, refactoring help, etc.

In this book, we will be using IDLE in our examples because it comes with Python and will provide a common test bed. I still think IDLE has the best, most consistent code highlighting of any IDE I have used. Code highlighting is important in my mind in that it helps prevent me from using one of Python's keywords (or built-ins) for a variable name. In case you're wondering, here is a list of those key words:

```
1 and      del      from     not      while
2 as       elif     global   or       with
3 assert   else     if       pass    yield
4 break    except   import   print
5 class    exec    in       raise
6 continue finally is      return
7 def     for     lambda  try
```

Let's see what happens as we type out a few things in Python:



image

As you can see, IDLE color coded everything. A key word is magenta, a string of text is in green, a comment is in red and the output from the print function is in blue.

Wrapping Up

In this chapter we learned how to use Python's integrated development environment, IDLE. We also learned what **comments** are and how to use them. At this point, you should be familiar enough with IDLE to use it in the rest of this book. There are many other integrated development environments (IDEs) for Python. There are free ones like PyDev and Editra, and there are some others that you have to pay for, such as WingWare and PyCharm. There are also plug-ins for regular text editors that allow you to code in Python too. I think IDLE is a good place to start, but if you already have a favorite editor, feel free to continue using that.

At this point, we are ready to move on and start learning about Python's various data types. We will start with Strings in the following chapter.

Chapter 2 - All About Strings

There are several data types in Python. The main data types that you'll probably see the most are string, integer, float, list, dict and tuple. In this chapter, we'll cover the string data type. You'll be surprised how many things you can do with strings in Python right out of the box. There's also a string module that you can import to access even more functionality, but we won't be looking at that in this chapter. Instead, we will be covering the following topics:

- How to create strings
- String concatenation
- String methods
- String slicing
- String substitution

How to Create a String

Strings are usually created in one of three ways. You can use single, double or triple quotes. Let's take a look!

```
1 >>> my_string = "Welcome to Python!"  
2 >>> another_string = 'The bright red fox jumped the fence.'  
3 >>> a_long_string = '''This is a  
4 multi-line string. It covers more than  
5 one line'''
```

The triple quoted line can be done with three single quotes or three double quotes. Either way, they allow the programmer to write strings over multiple lines. If you print it out, you will notice that the output retains the line breaks. If you need to use single quotes in your string, then wrap it in double quotes. See the following example.

```
1 >>> my_string = "I'm a Python programmer!"  
2 >>> otherString = 'The word "python" usually refers to a snake'  
3 >>> tripleString = """Here's another way to embed "quotes" in a string"""
```

The code above demonstrates how you could put single quotes or double quotes into a string. There's actually one other way to create a string and that is by using the `str` method. Here's how it works:

```
1 >>> my_number = 123
2 >>> my_string = str(my_number)
```

If you type the code above into your interpreter, you'll find that you have transformed the integer value into a string and assigned the string to the variable *my_string*. This is known as **casting**. You can cast some data types into other data types, like numbers into strings. But you'll also find that you can't always do the reverse, such as casting a string like 'ABC' into an integer. If you do that, you'll end up with an error like the one in the following example:

```
1 >>> int('ABC')
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 ValueError: invalid literal for int() with base 10: 'ABC'
```

We will look at exception handling in a later chapter, but as you may have guessed from the message, this means that you cannot convert a literal into an integer. However, if you had done

```
1 >>> x = int("123")
```

then that would have worked fine.

It should be noted that a string is one of Python immutable types. What this means is that you cannot change a string's content after creation. Let's try to change one to see what happens:

```
1 >>> my_string = "abc"
2 >>> my_string[0] = "d"
3 Traceback (most recent call last):
4   File "<string>", line 1, in <fragment>
5 TypeError: 'str' object does not support item assignment
```

Here we try to change the first character from an "a" to a "d"; however this raises a `TypeError` that stops us from doing so. Now you may think that by assigning a new string to the same variable that you've changed the string. Let's see if that's true:

```
1 >>> my_string = "abc"
2 >>> id(my_string)
3 19397208
4 >>> my_string = "def"
5 >>> id(my_string)
6 25558288
7 >>> my_string = my_string + "ghi"
8 >>> id(my_string)
9 31345312
```

By checking the `id` of the object, we can determine that any time we assign a new value to the variable, its identity changes.

Note that in Python 2.x, strings can only contain **ASCII** characters. If you require **unicode** in Python 2.x, then you will need to precede your string with a **u**. Here's an example:

```
1 my_unicode_string = u"This is unicode!"
```

The example above doesn't actually contain any unicode, but it should give you the general idea. In Python 3.x, all strings are unicode.

String Concatenation

Concatenation is a big word that means to combine or add two things together. In this case, we want to know how to add two strings together. As you might suspect, this operation is very easy in Python:

```
1 >>> string_one = "My dog ate "
2 >>> string_two = "my homework!"
3 >>> string_three = string_one + string_two
```

The `'+'` operator concatenates the two strings into one.

String Methods

A string is an object in Python. In fact, everything in Python is an object. However, you're not really ready for that. If you want to know more about how Python is an object oriented programming language, then you'll need to skip to that chapter. In the meantime, it's enough to know that strings have their very own methods built into them. For example, let's say you have the following string:

```
1 >>> my_string = "This is a string!"
```

Now you want to cause this string to be entirely in uppercase. To do that, all you need to do is call its **upper()** method, like this:

```
1 >>> my_string.upper()
```

If you have your interpreter open, you can also do the same thing like this:

```
1 >>> "This is a string!".upper()
```

There are many other string methods. For example, if you wanted everything to be lowercase, you would use the **lower()** method. If you wanted to remove all the leading and trailing white space, you would use **strip()**. To get a list of all the string methods, type the following command into your interpreter:

```
1 >>> dir(my_string)
```

You should end up seeing something like the following:

```
[ '__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format__',  
 '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__getslice__', '__gt__',  
 '__hash__', '__init__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',  
 '__sizeof__', '__str__', '__subclasshook__', '__formatter_field_name_split__',  
 '__formatter_parser__', 'capitalize', 'center', 'count', 'decode', 'encode', 'endswith', 'expandtabs',  
 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower', 'isspace',  
 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex',  
 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase',  
 'title', 'translate', 'upper', 'zfill']
```

You can safely ignore the methods that begin and end with double-underscores, such as `__add__`. They are not used in every day Python coding. Focus on the other ones instead. If you'd like to know what one of them does, just ask for **help**. For example, say you want to learn what **capitalize** is for. To find out, you would type

```
1 >>> help(my_string.capitalize)
```

This would return the following information:

Help on built-in function capitalize:

```
capitalize(...)  
S.capitalize() -> string
```

Return a copy of the string S with only its first character capitalized.

You have just learned a little bit about a topic called **introspection**. Python allows easy introspection of all its objects, which makes it very easy to use. Basically, introspection allows you to ask Python about itself. In an earlier section, you learned about casting. You may have wondered how to tell what type the variable was (i.e. an int or a string). You can ask Python to tell you that!

```
1 >>> type(my_string)  
2 <type 'str'>
```

As you can see, the my_string variable is of type str!

String Slicing

One subject that you'll find yourself doing a lot of in the real world is string slicing. I have been surprised how often I have needed to know how to do this in my day-to-day job. Let's take a look at how slicing works with the following string:

```
1 >>> my_string = "I like Python!"
```

Each character in a string can be accessed using slicing. For example, if I want to grab just the first character, I could do this:

```
1 >>> my_string[0:1]
```

This grabs the first character in the string up to, but **not** including, the 2nd character. Yes, Python is zero-based. It's a little easier to understand if we map out each character's position in a table:

0 1 2 3 4 5 6 7 8 9 10 11 12 13 - - - - - I l i k e P y t h o n !

Thus we have a string that is 14 characters long, starting at zero and going through thirteen. Let's do a few more examples to get these concepts into our heads better.

```
1 >>> my_string[:1]
2 'I'
3 >>> my_string[0:12]
4 'I like Pytho'
5 >>> my_string[0:13]
6 'I like Python'
7 >>> my_string[0:14]
8 'I like Python!'
9 >>> my_string[0:-5]
10 'I like Py'
11 >>> my_string[:]
12 'I like Python!'
13 >>> my_string[2:]
14 'like Python!'
```

As you can see from these examples, we can do a slice by just specifying the beginning of the slice (i.e. `my_string[2:]`), the ending of the slice (i.e. `my_string[:1]`) or both (i.e. `my_string[0:13]`). We can even use negative values that start at the end of the string. So the example where we did `my_string[0:-5]` starts at zero, but ends 5 characters before the end of the string.

You may be wondering where you would use this. I find myself using it for parsing fixed width records in files or occasionally for parsing complicated file names that follow a very specific naming convention. I have also used it in parsing out values from binary-type files. Any job where you need to do text file processing will be made easier if you understand slicing and how to use it effectively.

You can also access individual characters in a string via indexing. Here is an example:

```
1 >>> print(my_string[0])
```

The code above will print out the first character in the string.

String Formatting

String formatting (AKA substitution) is the topic of substituting values into a base string. Most of the time, you will be inserting strings within strings; however you will also find yourself inserting integers and floats into strings quite often as well. There are two different ways to accomplish this task. We'll start with the old way of doing things and then move on to the new.

Ye Olde Way of Substituting Strings

The easiest way to learn how to do this is to see a few examples. So here we go:

```
1 >>> my_string = "I like %s" % "Python"
2 >>> my_string
3 'I like Python'
4 >>> var = "cookies"
5 >>> newString = "I like %s" % var
6 >>> newString
7 'I like cookies'
8 >>> another_string = "I like %s and %s" % ("Python", var)
9 >>> another_string
10 'I like Python and cookies'
```

As you've probably guessed, the `%s` is the important piece in the code above. It tells Python that you may be inserting text soon. If you follow the string with a percent sign and another string or variable, then Python will attempt to insert it into the string. You can insert multiple strings by putting multiple instances of `%s` inside your string. You'll see that in the last example. Just note that when you insert more than one string, you have to enclose the strings that you're going to insert with parentheses.

Now let's see what happens if we don't insert enough strings:

```
1 >>> another_string = "I like %s and %s" % "Python"
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4     TypeError: not enough arguments for format string
```

Oops! We didn't pass enough arguments to format the string! If you look carefully at the example above, you'll notice it has two instances of `%s`, so to insert strings into it, you have to pass it the same number of strings! Now we're ready to learn about inserting integers and floats. Let's take a look!

```
1 >>> my_string = "%i + %i = %i" % (1,2,3)
2 >>> my_string
3 '1 + 2 = 3'
4 >>> float_string = "%f" % (1.23)
5 >>> float_string
6 '1.230000'
7 >>> float_string2 = "%.2f" % (1.23)
8 >>> float_string2
9 '1.23'
10 >>> float_string3 = "%.2f" % (1.237)
11 >>> float_string3
12 '1.24'
```

The first example above is pretty obvious. We create a string that accept three arguments and we pass them in. Just in case you hadn't figured it out yet, no, Python isn't actually doing any addition in that first example. For the second example, we pass in a float. Note that the output includes a lot of extra zeroes. We don't want that, so we tell Python to limit it to two decimal places in the 3rd example ("%.2f"). The last example shows you that Python will do some rounding for you if you pass it a float that's greater than two decimal places.

Now let's see what happens if we pass it bad data:

```
1 >>> int_float_err = "%i + %f" % ("1", "2.00")
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 TypeError: %d format: a number is required, not str
```

In this example, we pass it two strings instead of an integer and a float. This raises a `TypeError` and tells us that Python was expecting a number. This refers to not passing an integer, so let's fix that and see if that fixes the issue:

```
1 >>> int_float_err = "%i + %f" % (1, "2.00")
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 TypeError: float argument required, not str
```

Nope. We get the same error, but a different message that tells us we should have passed a float. As you can see, Python gives us pretty good information about what went wrong and how to fix it. If you fix the inputs appropriately, then you should be able to get this example to run.

Let's move on to the new method of string formatting!

Templates and the New String Formatting Methodology

This new method was actually added back in Python 2.4 as string templates, but was added as a regular string method via the `format` method in Python 2.6. So it's not really a new method, just newer. Anyway, let's start with templates!

```
1 >>> print("%(lang)s is fun!" % {"lang": "Python"})
2 Python is fun!
```

This probably looks pretty weird, but basically we just changed our `%s` into `%(lang)s`, which is basically the `%s` with a variable inside it. The second part is actually called a Python dictionary that we will be studying in the next section. Basically it's a key:value pair, so when Python sees the key "lang" in the string AND in the key of the dictionary that is passed in, it replaces that key with its value. Let's look at some more samples:

```

1 >>> print("%(value)s %(value)s %(value)s !" % {"value":"SPAM"})
2 SPAM SPAM SPAM !
3 >>> print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2})
4 Traceback (most recent call last):
5   File "<string>", line 1, in <fragment>
6 KeyError: 'z'
7 >>> print("%(x)i + %(y)i = %(z)i" % {"x":1, "y":2, "z":3})
8 1 + 2 = 3

```

In the first example, you'll notice that we only passed in one value, but it was inserted 3 times! This is one of the advantages of using templates. The second example has an issue in that we forgot to pass in a key, namely the "z" key. The third example rectifies this issue and shows the result. Now let's look at how we can do something similar with the string's format method!

```

1 >>> "Python is as simple as {0}, {1}, {2}".format("a", "b", "c")
2 'Python is as simple as a, b, c'
3 >>> "Python is as simple as {1}, {0}, {2}".format("a", "b", "c")
4 'Python is as simple as b, a, c'
5 >>> xy = {"x":0, "y":10}
6 >>> print("Graph a point at where x={x} and y={y}".format(**xy))
7 Graph a point at where x=0 and y=10

```

In the first two examples, you can see how we can pass items positionally. If we rearrange the order, we get a slightly different output. The last example uses a dictionary like we were using in the templates above. However, we have to extract the dictionary using the double asterisk to get it to work correctly here.

There are lots of other things you can do with strings, such as specifying a width, aligning the text, converting to different bases and much more. Be sure to take a look at some of the references below for more information.

- Python's official documentation on the str type³
- String Formatting⁴
- More on String Formatting⁵
- Python 2.x documentation on unicode⁶

³<https://docs.python.org/3/library/functions.html#func-str>

⁴<https://docs.python.org/3/library/string.html#string-formatting>

⁵<https://docs.python.org/3/library/string.html#formatexamples>

⁶<http://docs.python.org/2/library/functions.html#unicode>

Wrapping Up

We have covered a lot in this chapter. Let's review:

First we learned how to create strings themselves, then we moved on to the topic of string concatenation. After that we looked at some of the methods that the string object gives us. Next we looked at string slicing and we finished up by learning about string substitution.

In the next chapter, we will look at three more of Python's built-in data types: lists, tuples and dictionaries. Let's get to it!

Chapter 3 - Lists, Tuples and Dictionaries

Python has several other important data types that you'll probably use every day. They are called lists, tuples and dictionaries. This chapter's aim is to get you acquainted with each of these data types. They are not particularly complicated, so I expect that you will find learning how to use them very straight forward. Once you have mastered these three data types plus the string data type from the previous chapter, you will be quite a ways along in your education of Python. You'll be using these four building blocks in 99% of all the applications you will write.

Lists

A Python list is similar to an array in other languages. In Python, an empty list can be created in the following ways.

```
1 >>> my_list = []
2 >>> my_list = list()
```

As you can see, you can create the list using square brackets or by using the Python built-in, `list`. A list contains a list of elements, such as strings, integers, objects or a mixture of types. Let's take a look at some examples:

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list2 = ["a", "b", "c"]
3 >>> my_list3 = ["a", 1, "Python", 5]
```

The first list has 3 integers, the second has 3 strings and the third has a mixture. You can also create lists of lists like this:

```
1 >>> my_nested_list = [my_list, my_list2]
2 >>> my_nested_list
3 [[1, 2, 3], ['a', 'b', 'c']]
```

Occasionally, you'll want to combine two lists together. The first way is to use the `extend` method:

```
1 >>> combo_list = []
2 >>> one_list = [4, 5]
3 >>> combo_list.extend(one_list)
4 >>> combo_list
5 [4, 5]
```

A slightly easier way is to just add two lists together.

```
1 >>> my_list = [1, 2, 3]
2 >>> my_list2 = ["a", "b", "c"]
3 >>> combo_list = my_list + my_list2
4 >>> combo_list
5 [1, 2, 3, 'a', 'b', 'c']
```

Yes, it really is that easy. You can also sort a list. Let's spend a moment to see how to do that:

```
1 >>> alpha_list = [34, 23, 67, 100, 88, 2]
2 >>> alpha_list.sort()
3 >>> alpha_list
4 [2, 23, 34, 67, 88, 100]
```

Now there is a got-cha above. Can you see it? Let's do one more example to make it obvious:

```
1 >>> alpha_list = [34, 23, 67, 100, 88, 2]
2 >>> sorted_list = alpha_list.sort()
3 >>> sorted_list
4 >>> print(sorted_list)
5 None
```

In this example, we try to assign the sorted list to a variable. However, when you call the `sort()` method on a list, it sorts the list in-place. So if you try to assign the result to another variable, then you'll find out that you'll get a `None` object, which is like a Null in other languages. Thus when you want to sort something, just remember that you sort them in-place and you cannot assign it to a different variable.

You can slice a list just like you do with a string:

```
1 >>> alpha_list[0:3]
2 [2, 23, 34]
```

This code returns a list of just the first 3 elements.

Tuples

A tuple is similar to a list, but you create them with parentheses instead of square brackets. You can also use the **tuple** built-in. The main difference is that a tuple is immutable while the list is mutable. Let's take a look at a few examples:

```
1 >>> my_tuple = (1, 2, 3, 4, 5)
2 >>> my_tuple[0:3]
3 (1, 2, 3)
4 >>> another_tuple = tuple()
5 >>> abc = tuple([1, 2, 3])
```

The code above demonstrates one way to create a tuple with five elements. It also shows that you can do tuple slicing. However, you cannot sort a tuple! The last two examples shows how to create tuples using the **tuple** keyword. The first one just creates an empty tuple whereas the second example has three elements inside it. Notice that it has a list inside it. This is an example of **casting**. We can change or **cast** an item from one data type to another. In this case, we cast a list into a tuple. If you want to turn the **abc** tuple back into a list, you can do the following:

```
1 >>> abc_list = list(abc)
```

To reiterate, the code above casts the tuple (**abc**) into a list using the **list** function.

Dictionaries

A Python dictionary is basically a **hash table** or a hash mapping. In some languages, they might be referred to as **associative memories** or **associative arrays**. They are indexed with keys, which can be any immutable type. For example, a string or number can be a key. You need to be aware that a dictionary is an unordered set of key:value pairs and the keys must be unique. You can get a list of keys by calling a dictionary instance's **keys** method. To check if a dictionary has a key, you can use Python's **in** keyword. In some of the older versions of Python (2.3 and older to be specific), you will see the **has_key** keyword used for testing if a key is in a dictionary. This keyword is deprecated in Python 2.x and removed entirely from Python 3.x.

Let's take a moment to see how we create a dictionary.

```

1 >>> my_dict = {}
2 >>> another_dict = dict()
3 >>> my_other_dict = {"one":1, "two":2, "three":3}
4 >>> my_other_dict
5 {'three': 3, 'two': 2, 'one': 1}

```

The first two examples show how to create an empty dictionary. All dictionaries are enclosed with curly braces. The last line is printed out so you can see how unordered a dictionary is. Now it's time to find out how to access a value in a dictionary.

```

1 >>> my_other_dict["one"]
2 1
3 >>> my_dict = {"name":"Mike", "address":"123 Happy Way"}
4 >>> my_dict["name"]
5 'Mike'

```

In the first example, we use the dictionary from the previous example and pull out the value associated with the key called "one". The second example shows how to acquire the value for the "name" key. Now let's see how to tell if a key is in a dictionary or not:

```

1 >>> "name" in my_dict
2 True
3 >>> "state" in my_dict
4 False

```

So, if the key is in the dictionary, Python returns a **Boolean True**. Otherwise it returns a Boolean **False**. If you need to get a listing of all the keys in a dictionary, then you do this:

```

1 >>> my_dict.keys()
2 dict_keys(['name', 'address'])

```

In Python 2, the **keys** method returns a list. But in Python 3, it returns a *view object*. This gives the developer the ability to update the dictionary and the view will automatically update too. Also note that when using the **in** keyword for dictionary membership testing, it is better to do it against the dictionary instead of the list returned from the **keys** method. See below:

```

1 >>> "name" in my_dict      # this is good
2 >>> "name" in my_dict.keys() # this works too, but is slower

```

While this probably won't matter much to you right now, in a real job situation, seconds matter. When you have thousands of files to process, these little tricks can save you a lot of time in the long run!

Wrapping Up

In this chapter you just learned how to construct a Python list, tuple and dictionary. Make sure you understand everything in this section before moving on. These concepts will assist you in designing your programs. You will be building complex data structures using these building blocks every day if you choose to pursue employment as a Python programmer. Each of these data types can be nested inside the others. For example, you can have a nested dictionary, a dictionary of tuples, a tuple made up of several dictionaries, and on and on.

When you are ready to move on, we will learn about Python's support for conditional statements.

Chapter 4 - Conditional Statements

Every computer language I have ever used has had at least one conditional statement. Most of the time that statement is the **if/elif/else** structure. This is what Python has. Other languages also include the **case/switch** statement which I personally enjoy, however Python does not include it. You can make your own if you really want to, but this book is focused on learning Python fundamentals, so we're going to be only focusing on what's included with Python in this chapter.

The conditional statement checks to see if a statement is True or False. That's really all it does. However we will also be looking at the following Boolean operations: **and**, **or**, and **not**. These operations can change the behaviour of the conditional in simple and complex ways, depending on your project.

The if statement

Python's if statement is pretty easy to use. Let's spend a few minutes looking at some examples to better acquaint ourselves with this construct.

```
1 >>> if 2 > 1:  
2     print("This is a True statement!")  
3 This is a True Statement!
```

This conditional tests the “truthfulness” of the following statement: $2 > 1$. Since this statement evaluates to True, it will cause the last line in the example to print to the screen or **standard out** (stdout).

Python Cares About Space

The Python language cares a lot about space. You will notice that in our conditional statement above, we indented the code inside the **if** statement four spaces. This is very important! If you do not indent your blocks of code properly, the code will not execute properly. It may not even run at all.

Also, do **not** mix tabs and spaces. IDLE will complain that there is an issue with your file and you will have trouble figuring out what the issue is. The recommended number of spaces to indent a block of code is four. You can actually indent your code any number of spaces as long as you are consistent. However, the 4-space rule is one that is recommended by the Python Style Guide and is the rule that is followed by the Python code developers.

Let's look at another example:

```
1 >>> var1 = 1
2 >>> var2 = 3
3 >>> if var1 > var2:
4     print("This is also True")
```

In this one, we compare two variables that translate to the question: Is `1 > 3`? Obviously one is not greater than three, so it doesn't print anything. But what is we wanted it to print something? That's where the `else` statement comes in. Let's modify the conditional to add that piece:

```
1 if var1 > var2:
2     print("This is also True")
3 else:
4     print("That was False!")
```

If you run this code, it will print the string that follows the `else` statement. Let's change gears here and get some information from the user to make this more interesting. In Python 2.x, you can get information using a built-in called `raw_input`. If you are using Python 3.x, then you'll find that `raw_input` no longer exists. It was renamed to just `input`. They function in the same way though. To confuse matters, Python 2.x actually has a built-in called `input` as well; however it tries to execute what is entered as a Python expression whereas `raw_input` returns a string. Anyway, we'll be using Python 2.x's `raw_input` for this example to get the user's age.

```
1 # Python 2.x code
2 value = raw_input("How much is that doggy in the window? ")
3 value = int(value)
4
5 if value < 10:
6     print("That's a great deal!")
7 elif 10 <= value <= 20:
8     print("I'd still pay that...")
9 else:
10    print("Wow! That's too much!")
```

|
Let's break this down a bit. The first line asks the user for an amount. In the next line, it converts the user's input into an integer. So if you happen to type a floating point number like `1.23`, it will get truncated to just `1`. If you happen to type something other than a number, then you'll receive an

exception. We'll be looking at how to handle exceptions in a later chapter, so for now, just enter an integer.

In the next few lines, you can see how we check for 3 different cases: less than 10, greater than or equal to 10 but less than or equal to 20 or something else. For each of these cases, a different string is printed out. Try putting this code into IDLE and save it. Then run it a few times with different inputs to see how it works.

You can add multiple `elif` statements to your entire conditional. The `else` is optional, but makes a good default.

Boolean Operations

Now we're ready to learn about Boolean operations (and, or, not). According to the Python documentation, their order of priority is first **or**, then **and**, then **not**. Here's how they work:

- **or** means that if any conditional that is “ored” together is True, then the following statement runs
- **and** means that all statements must be True for the following statement to run
- **not** means that if the conditional evaluates to False, it is True. This is the most confusing, in my opinion.

Let's take a look at some examples of each of these. We will start with **or**.

```
1 x = 10
2 y = 20
3
4 if x < 10 or y > 15:
5     print("This statement was True!")
```

Here we create a couple of variables and test if one is less than ten or if the other is greater than 15. Because the latter is greater than 15, the print statement executes. As you can see, if one or both of the statements are True, it will execute the statement. Let's take a look at how **and** works:

```
1 x = 10
2 y = 10
3 if x == 10 and y == 15:
4     print("This statement was True")
5 else:
6     print("The statement was False!")
```

If you run the code above, you will see that first statement does not get run. Instead, the statement under the **else** is executed. Why is that? Well, it is because what we are testing is both **x** and **y** are 10 and 15 respectively. In this case, they are not, so we drop to the **else**. Thus, when you **and** two statements together, **both** statements have to evaluate to True for it to execute the following code. Also note that to test equality in Python, you have to use a double equal sign. A single equals sign is known as the **assignment operator** and is only for assigning a value to a variable. If you had tried to run the code above with one of those statement only having one equals sign, you would have received a message about invalid syntax.

Note that you can also **or** and **and** more than two statements together. However, I do not recommend that as the more statements that you do that too, the harder it can be to understand and debug.

Now we're ready to take a look at the **not** operation.

```
1 my_list = [1, 2, 3, 4]
2 x = 10
3 if x not in my_list:
4     print("'x' is not in the list, so this is True!")
```

In this example, we create a list that contains four integers. Then we write a test that asks if “**x**” is not in that list. Because “**x**” equals 10, the statement evaluates to True and the message is printed to the screen. Another way to test for **not** is by using the exclamation point, like this:

```
1 x = 10
2 if x != 11:
3     print("x is not equal to 11!")
```

If you want to, you can combine the **not** operation with the other two to create more complex conditional statements. Here is a simple example:

```
1 my_list = [1, 2, 3, 4]
2 x = 10
3 z = 11
4 if x not in my_list and z != 10:
5     print("This is True!")
```

Checking for Nothing

Because we are talking about statements that evaluate to True, we probably need to cover what evaluates to False. Python has the keyword **False** which I've mentioned a few times. However an empty string, tuple or list also evaluates to False. There is also another keyword that basically evaluates to False which is called **None**. The **None** value is used to represent the absence of value. It's kind of analogous to Null, which you find in databases. Let's take a look at some code to help us better understand how this all works:

```
1 empty_list = []
2 empty_tuple = ()
3 empty_string = ""
4 nothing = None
5
6 if empty_list == []:
7     print("It's an empty list!")
8
9 if empty_tuple:
10    print("It's not an empty tuple!")
11
12 if not empty_string:
13    print("This is an empty string!")
14
15 if not nothing:
16    print("Then it's nothing!")
```

The first four lines set up four variables. Next we create four conditionals to test them. The first one checks to see if the `empty_list` is really empty. The second conditional checks to see if the `empty_tuple` has something in it. Yes, you read that right, The second conditional only evaluates to True if the tuple is **not** empty! The last two conditionals are doing the opposite of the second. The third is checking if the string **is** empty and the fourth is checking if the `nothing` variable is really `None`.

The `not` operator means that we are checking for the opposite meaning. In other words, we are checking if the value is NOT True. So in the third example, we check if the empty string is REALLY empty. Here's another way to write the same thing:

```
1 if empty_string == "":
2     print("This is an empty string!")
```

To really nail this down, let's set the `empty_string` variable to actually contain something:

```
1 >>> empty_string = "something"
2 >>> if empty_string == "":
3     print("This is an empty string!")
```

If you run this, you will find that nothing is printed as we will only print something if the variable is an empty string.

Please note that none of these variables equals the other. They just evaluate the same way. To prove this, we'll take a look at a couple of quick examples:

```
1 >>> empty_list == empty_string
2 False
3 >>> empty_string == nothing
4 False
```

As you can see, they do not equal each other. You will find yourself checking your data structures for data a lot in the real world. Some programmers actually like to just wrap their structures in an exception handler and if they happen to be empty, they'll catch the exception. Others like to use the strategy mentioned above where you actually test the data structure to see if it has something in it. Both strategies are valid.

Personally, I find the **not** operator a little confusing and don't use it that much. But you will find it useful from time to time.

Special Characters

Strings can contain special characters, like tabs or new lines. We need to be aware of those as they can sometimes crop up and cause problems. For example, the new line character is defined as "n", while the tab character is defined as "t". Let's see a couple of examples so you will better understand what these do:

```
1 >>> print("I have a \n new line in the middle")
2 I have a
3     new line in the middle
4 >>> print("This sentence is \ttabbed!")
5 This sentence is      tabbed!
```

Was the output as you expected? In the first example, we have a "n" in the middle of the sentence, which forces it to print out a new line. Because we have a space after the new line character, the second line is indented by a space. The second example shows what happens when we have a tab character inside of a sentence.

Sometimes you will want to use escape characters in a string, such as a backslash. To use escape characters, you have to actually use a backslash, so in the case of a backslash, you would actually type two backslashes. Let's take a look:

```
1 >>> print("This is a backslash \\")
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 EOL while scanning string literal: <string>, line 1, pos 30
5 >>> print("This is a backslash \\\\")
6 This is a backslash \\\\"
```

You will notice that the first example didn't work so well. Python thought we were escaping the double-quote, so it couldn't tell where the end of the line (EOL) was and it threw an error. The second example has the backslash appropriately escaped.

if __name__ == "__main__"

You will see a very common conditional statement used in many Python examples. This is what it looks like:

```
1 if __name__ == "__main__":
2     # do something!
```

You will see this at the end of a file. This tells Python that you only want to run the following code if this program is executed as a standalone file. I use this construct a lot to test that my code works in the way I expect it to. We will be discussing this later in the book, but whenever you create a Python script, you create a Python module. If you write it well, you might want to import it into another module. When you do import a module, it will **not** run the code that's under the conditional because `__name__` will no longer equal "`__main__`". We will look at this again in **Chapter 11** when we talk about **classes**.

Wrapping Up

We've covered a fair bit of ground in this chapter. You have learned how to use conditional statements in several different ways. We have also spent some time getting acquainted with Boolean operators. As you have probably guessed, each of these chapters will get slightly more complex as we'll be using each building block that we learn to build more complicated pieces of code. In the next chapter, we will continue that tradition by learning about Python's support of loops!

Chapter 5 - Loops

Every programming language I have tried has some kind of looping construct. Most have more than one. The Python world has two types of loops:

- the **for** loop and
- the **while** loop

|

You will find that the **for** loop is by far the most popular of the two. Loops are used when you want to do something many times. Usually you will find that you need to do some operation or a set of operations on a piece of data over and over. This is where loops come in. They make it really easy to apply this sort of logic to your data.

Let's get started learning how these fun structures work!

The **for** Loop

As mentioned above, you use a loop when you want to iterate over something n number of times. It's a little easier to understand if we see an example. Let's use Python's builtin **range** function. The **range** function will create a list that is n in length. In Python 2.x, there is actually another function called **xrange** that is a number generator and isn't as resource intensive as **range**. They basically changed **xrange** into **range** in Python 3. Here is an example:

```
1 >>> range(5)
2 range(0, 5)
```

As you can see, the **range** function above took an integer and returned a **range** object. The **range** function also accepts a beginning value, an end value and a step value. Here are two more examples:

```
1 >>> range(5,10)
2 range(5, 10)
3 >>> list(range(1, 10, 2))
4 [1, 3, 5, 7, 9]
```

The first example demonstrates that you can pass a beginning and end value and the **range** function will return the numbers from the beginning value up to but not including the end value. So in the case of 5-10, we get 5-9. The second example shows how to use the **list** function to cause the **range** function to return every second element between 1 and 10. So it starts with one, skips two, etc. Now you're probably wondering what this has to do with loops. Well one easy way to show how a loop works is if we use the **range** function! Take a look:

```
1 >>> for number in range(5):
2     print(number)
3
4 0
5 1
6 2
7 3
8 4
```

What happened here? Let's read it from left to right to figure it out. For each number in a range of 5, print the number. We know that if we call range with a value of 5, it will return a list of 5 elements. So each time through the loop, it prints out each of the elements. The for loop above would be the equivalent of the following:

```
1 >>> for number in [0, 1, 2, 3, 4]:
2     print(number)
```

The range function just makes it a little bit smaller. The for loop can loop over any kind of Python iterator. We've already seen how it can iterate over a list. Let's see if it can also iterate over a dictionary.

```
1 >>> a_dict = {"one":1, "two":2, "three":3}
2 >>> for key in a_dict:
3     print(key)
4
5 three
6 two
7 one
```

When you use a **for** loop with a dictionary, you'll see that it automatically loops over the keys. We didn't have to say **for key in a_dict.keys()** (although that would have worked too). Python just did the right thing for us. You may be wondering why the keys printed in a different order than they were defined in the dictionary. As you may recall from chapter 3, dictionaries are unordered, so when we iterate over it, the keys could be in any order.

Now if you know that the keys can be sorted, then you can do that before you iterate over them. Let's change the dictionary slightly to see how that works.

```
1 >>> a_dict = {1:"one", 2:"two", 3:"three"}  
2 >>> keys = a_dict.keys()  
3 >>> keys = sorted(keys)  
4 >>> for key in keys:  
5         print(key)  
6  
7 1  
8 2  
9 3
```

Let's take a moment to figure out what this code does. First off, we create a dictionary that has integers for keys instead of strings. Then we extract the keys from the dictionary. Whenever you call the `keys()` method, it will return an unordered list of the keys. If you print them out and find them to be in ascending order, then that's just happenstance. Now we have a view of the dictionary's keys that are stored in a variable called `keys`. We sort it and then we use the `for` loop to loop over it.

Now we're ready to make things a little bit more interesting. We are going to loop over a range, but we want to print out only the even numbers. To do this, we want to use a conditional statement instead of using the range's step parameter. Here's one way you could do this:

```
1 >>> for number in range(10):  
2     if number % 2 == 0:  
3         print(number)  
4  
5 0  
6 2  
7 4  
8 6  
9 8
```

You're probably wondering what's going on here. What's up with the percent sign? In Python, the `%` is called a modulus operator. When you use the modulus operator, it will return the remainder. There is no remainder when you divide an even number by two, so we print those numbers out. You probably won't use the modulus operator a lot in the wild, but I have found it useful from time to time.

Now we're ready to learn about the `while` loop.

The `while` Loop

The while loop is also used to repeat sections of code, but instead of looping n number of times, it will only loop until a specific condition is met. Let's look at a very simple example:

```
1 >>> i = 0
2 >>> while i < 10:
3         print(i)
4         i = i + 1
```

The while loop is kind of like a conditional statement. Here's what this code means: while the variable `i` is less than ten, print it out. Then at the end, we increase `i`'s value by one. If you run this code, it should print out 0-9, each on its own line and then stop. If you remove the piece where we increment `i`'s value, then you'll end up with an infinite loop. This is usually a bad thing. Infinite loops are to be avoided and are known as logic errors.

There is another way to break out of a loop. It is by using the `break` builtin. Let's see how that works:

```
1 >>> while i < 10:
2         print(i)
3         if i == 5:
4             break
5         i += 1
6
7 0
8 1
9 2
10 3
11 4
12 5
```

In this piece of code, we add a conditional to check if the variable `i` ever equals 5. If it does, then we break out of the loop. As you can see from the sample output, as soon as it reaches 5, the code stops even though we told the while loop to keep looping until it reached 10. You will also note that we changed how we increment the value by using `+=`. This is a handy shortcut that you can also use with other math operations, like subtraction (`-=`) and multiplication (`*=`).

The `break` builtin is known as a **flow control tool**. There is another one called `continue` that is used to basically skip an iteration or continue with the next iteration. Here's one way to use it:

```

1 i = 0
2
3 while i < 10:
4     if i == 3:
5         i += 1
6         continue
7
8     print(i)
9
10    if i == 5:
11        break
12    i += 1

```

This is a little confusing, no? Basically we added a second conditional that checks if `i` equals 3. If it does, we increment the variable and continue with the next loop, which effectively skips printing the value 3 to the screen. As before, when we reach a value of 5, we break out of the loop.

There's one more topic we need to cover regarding loops and that's the `else` statement.

What else is for in loops

The `else` statement in loops only executes if the loop completes successfully. The primary use of the `else` statement is for searching for items:

```

1 my_list = [1, 2, 3, 4, 5]
2
3 for i in my_list:
4     if i == 3:
5         print("Item found!")
6         break
7     print(i)
8 else:
9     print("Item not found!")

```

In this code, we break out of the loop when `i` equals 3. This causes the `else` statement to be skipped. If you want to experiment, you can change the conditional to look for a value that's not in the list, which will cause the `else` statement to execute. To be honest, I have never seen anyone use this structure in all my years as a programmer. Most of the examples I have seen are bloggers trying to explain what it is used for. I have seen several who use it to raise an error if an item is not found in the iterable that you were searching. You can read a fairly in depth article by one of the Python core developers [here⁷](#).

⁷https://ncoghlan_devs-python-notes.readthedocs.org/en/latest/python_concepts/break_else.html

Wrapping Up

Hopefully at this point you can see the value in Python loops. They make repetition easier and pretty easy to understand. You will likely see the `for` loop much more often than the `while` loop. In fact, we are going to look at another way `for` loops are used in the next chapter when we learn about comprehensions! If you're still not quite sure how all this works, you may want to re-read this chapter before continuing.

Chapter 6 - Python Comprehensions

The Python language has a couple of methods for creating lists and dictionaries that are known as comprehensions. There is also a third type of comprehension for creating a Python set. In this chapter we will learn how to use each type of comprehension. You will find that the comprehension constructs build on the knowledge you have acquired from the previous chapters as they contain loops and conditionals themselves.

List Comprehensions

List comprehensions in Python are very handy. They can also be a little hard to understand when and why you would use them. List comprehensions tend to be harder to read than just using a simple `for` loop as well. You may want to review the looping chapter before you continue.

If you are ready, then we'll spend some time looking at how to construct list comprehensions and learn how they can be used. A list comprehension is basically a one line `for` loop that produces a Python list data structure. Here's a simple example:

```
1 >>> x = [i for i in range(5)]
```

Let's break this down a bit. Python comes with a `range` function that can return a list of numbers. By default, it returns integers starting at 0 and going up to but not including the number you pass it. So in this case, it returns a list containing the integers 0-4. This can be useful if you need to create a list very quickly. For example, say you're parsing a file and looking for something in particular. You could use a list comprehension as a kind of filter:

```
1 if [i for i in line if "SOME TERM" in i]:  
2     # do something
```

I have used code similar to this to look through a file quickly to parse out specific lines or sections of the file. When you throw functions into the mix, you can start doing some really cool stuff. Say you want to apply a function to every element in a list, such as when you need to cast a bunch of strings into integers:

```

1 >>> x = ['1', '2', '3', '4', '5']
2 >>> y = [int(i) for i in x]
3 >>> y
4 [1, 2, 3, 4, 5]

```

This sort of thing comes up more often than you'd think. I have also had to loop over a list of strings and call a string method, such as strip on them because they had all kinds of leading or ending white space:

```

1 >>> myStrings = [s.strip() for s in myList]

```

There are also occasions where one needs to create a nested list comprehension. One reason to do that is to flatten multiple lists into one. This example comes from the Python documentation:

```

1 >>> vec = [[1,2,3], [4,5,6], [7,8,9]]
2 >>> [num for elem in vec for num in elem]
3 [1, 2, 3, 4, 5, 6, 7, 8, 9]

```

The documentation shows several other interesting examples for nested list comprehensions as well. I highly recommend taking a look at it! At this point, you should now be capable of using list comprehensions in your own code and use them well. Just use your imagination and you'll start seeing lots of good places where you too can use them.

Now we're ready to move on to Python's dictionary comprehensions!

Dictionary Comprehensions

Dictionary comprehensions started life in Python 3.0, but were backported to Python 2.7. They were originally proposed in the [Python Enhancement Proposal 274 \(PEP 274\)](#)⁸ back in 2001. They are pretty similar to a list comprehension in the way that they are organized.

The best way to understand is to just do one!

```

1 >>> print( {i: str(i) for i in range(5)} )
2 {0: '0', 1: '1', 2: '2', 3: '3', 4: '4'}

```

This is a pretty straightforward comprehension. Basically it is creating an integer key and string value for each item in the range. Now you may be wondering how you could use a dictionary comprehension in real life. [Mark Pilgrim](#)⁹ mentioned that you could use a dictionary comprehension for swapping the dictionary's keys and values. Here's how you would do that:

⁸<http://www.python.org/dev/peps/pep-0274/>

⁹<http://www.diveintopython3.net/comprehensions.html>

```
1 >>> my_dict = {1:"dog", 2:"cat", 3:"hamster"}  
2 >>> print( {value:key for key, value in my_dict.items()} )  
3 {'hamster': 3, 'dog': 1, 'cat': 2}
```

This will only work if the dictionary values are of a non-mutable type, such as a string. Otherwise you will end up causing an exception to be raised.

I could also see a dictionary comprehension being useful for creating a table out of class variables and their values. However, we haven't covered classes at this point, so I won't confuse you with that here.

Set Comprehensions

Set comprehensions are created in much the same way as dictionary comprehensions. Now a Python set is much like a mathematical set in that it doesn't have any repeated elements. You can create a normal set like this:

```
1 >>> my_list = [1, 2, 2, 3, 4, 5, 5, 7, 8]  
2 >>> my_set = set(my_list)  
3 >>> my_set  
4 set([1, 2, 3, 4, 5, 7, 8])
```

As you can see from the example above, the call to set has removed the duplicates from the list. Now let's rewrite this code to use a set comprehension:

```
1 >>> my_list = [1, 2, 2, 3, 4, 5, 5, 7, 8]  
2 >>> my_set = {x for x in my_list}  
3 >>> my_set  
4 set([1, 2, 3, 4, 5, 7, 8])
```

You will notice that to create a set comprehension, we basically changed the square brackets that a list comprehension uses to the curly braces that the dictionary comprehension has.

Wrapping Up

Now you know how to use the various Python comprehensions. You will probably find the list comprehension the most useful at first and also the most popular. If you start using your imagination, I am sure you will be able to find uses for all three types of comprehensions. Now we're ready to move on and learn about exception handling!

Chapter 7 - Exception Handling

What do you do when something bad happens in your program? Let's say you try to open a file, but you typed in the wrong path or you ask the user for information and they type in some garbage. You don't want your program to crash, so you implement exception handling. In Python, the construct is usually wrapped in what is known as a **try/except**. We will be looking at the following topics in this chapter:

- Common exception types
- Handling exceptions with **try/except**
- Learn how **try/except/finally** works
- Discover how the **else** statement works in conjunction with the **try/except**

Let's start out by learning about some of the most common exceptions that you'll see in Python. Note: an error and an exception are just different words that describe the same thing when we are talking about exception handling.

Common Exceptions

You have seen a few exceptions already. Here is a list of the most common built-in exceptions (definitions from the [Python documentation¹⁰](#)):

- **Exception** (this is what almost all the others are built off of)
- **AttributeError** - Raised when an attribute reference or assignment fails.
- **IOError** - Raised when an I/O operation (such as a print statement, the built-in `open()` function or a method of a file object) fails for an I/O-related reason, e.g., "file not found" or "disk full".
- **ImportError** - Raised when an import statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.
- **IndexError** - Raised when a sequence subscript is out of range.
- **KeyError** - Raised when a mapping (dictionary) key is not found in the set of existing keys.
- **KeyboardInterrupt** - Raised when the user hits the interrupt key (normally Control-C or Delete).
- **NameError** - Raised when a local or global name is not found.
- **OSError** - Raised when a function returns a system-related error.
- **SyntaxError** - Raised when the parser encounters a syntax error.

¹⁰<http://docs.python.org/2/library/exceptions.html>

- **TypeError** - Raised when an operation or function is applied to an object of inappropriate type. The associated value is a string giving details about the type mismatch.
- **ValueError** - Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as IndexError.
- **ZeroDivisionError** - Raised when the second argument of a division or modulo operation is zero.

There are a lot of other exceptions as well, but you probably won't see them all that often. However, if you are interested, you can go and read all about them in the [Python documentation¹¹](#).

How to Handle Exceptions

Handling exceptions in Python is really easy. Let's spend some time writing some examples that will cause exceptions. We will start with one of the most common computer science problems: division by zero.

```
1  >>> 1 / 0
2  Traceback (most recent call last):
3      File "<string>", line 1, in <fragment>
4  ZeroDivisionError: integer division or modulo by zero
5
6  >>> try:
7      1 / 0
8  except ZeroDivisionError:
9      print("You cannot divide by zero!")
10
11 You cannot divide by zero!
```

If you think back to elementary math class, you will recall that you cannot divide by zero. In Python, this operation will cause an error, as you can see in the first half of the example. To catch the error, we wrap the operation with a **try/except** statement.

Bare Excepts

Here's another way to catch the error:

¹¹<http://docs.python.org/2/library/exceptions.html>

```
1 >>> try:  
2     1 / 0  
3 except:  
4     print("You cannot divide by zero!")
```

This is **not** recommended! In Python, this is known as a **bare except**, which means it will catch any and all exceptions. The reason this is not recommended is that you don't know which exception you are catching. When you have something like `except ZeroDivisionError`, you are obviously trying to catch a division by zero error. In the code above, you cannot tell what you are trying to catch.

Let's take a look at a couple of other examples.

```
1 >>> my_dict = {"a":1, "b":2, "c":3}  
2 >>> try:  
3     value = my_dict["d"]  
4 except KeyError:  
5     print("That key does not exist!")  
6  
7 That key does not exist!  
8 >>> my_list = [1, 2, 3, 4, 5]  
9 >>> try:  
10    my_list[6]  
11 except IndexError:  
12    print("That index is not in the list!")  
13  
14 That index is not in the list!
```

In the first example, we create a 3-element dictionary. Then we try to access a key that is not in the dictionary. Because the key is not in the dictionary, it raises a **KeyError**, which we catch. The second example shows a list that is 5 items in length. We try to grab the 7th item from the index. Remember, Python lists are zero-based, so when you say [6], you're asking for the 7th item. Anyway, because there are only 5 items, it raises an **IndexError**, which we also catch.

You can also catch multiple exceptions with a single statement. There are a couple of different ways to do this. Let's take a look:

```
1 my_dict = {"a":1, "b":2, "c":3}
2 try:
3     value = my_dict["d"]
4 except IndexError:
5     print("This index does not exist!")
6 except KeyError:
7     print("This key is not in the dictionary!")
8 except:
9     print("Some other error occurred!")
```

This is a fairly standard way to catch multiple exceptions. First we try to access a key that doesn't exist in the dictionary. The try/except checks to see if you are catching a `KeyError`, which you are in the second `except` statement. You will also note that we have a bare `except` at the end. This is usually not recommended, but you'll probably see it from time to time, so it's good to know about it. Also note that most of the time, you won't need to wrap a block of code in multiple `except` handlers. You normally just need to wrap it in one.

Here's another way to catch multiple exceptions:

```
1 try:
2     value = my_dict["d"]
3 except (IndexError, KeyError):
4     print("An IndexError or KeyError occurred!")
```

Notice that in this example, we are putting the errors that we want to catch inside of parentheses. The problem with this method is that it's hard to tell which error has actually occurred, so the previous example is recommended.

Most of the time when an exception occurs, you will need to alert the user by printing to the screen or logging the message. Depending on the severity of the error, you may need to exit your program. Sometimes you will need to clean up before you exit the program. For example, if you have opened a database connection, you will want to close it before you exit your program or you may end up leaving connections open. Another example is closing a file handle that you have been writing to. You will learn more about file handling in the next chapter. But first, we need to learn how to clean up after ourselves. This is facilitated with the `finally` statement.

The `finally` Statement

The `finally` statement is really easy to use. Let's take a look at a silly example:

```
1 my_dict = {"a":1, "b":2, "c":3}
2
3 try:
4     value = my_dict["d"]
5 except KeyError:
6     print("A KeyError occurred!")
7 finally:
8     print("The finally statement has executed!")
```

If you run the code above, it will print the statement in the **except** and the **finally**. This is pretty simple, right? Now you can use the **finally** statement to clean up after yourself. You would also put the exit code at the end of the **finally** statement.

try, except, or else!

The **try/except** statement also has an **else** clause. The **else** will only run if there are no errors raised. We will spend a few moments looking at a couple examples:

```
1 my_dict = {"a":1, "b":2, "c":3}
2
3 try:
4     value = my_dict["a"]
5 except KeyError:
6     print("A KeyError occurred!")
7 else:
8     print("No error occurred!")
```

Here we have a dictionary with 3 elements and in the **try/except** we access a key that exists. This works, so the **KeyError** is **not** raised. Because there is no error, the **else** executes and “No error occurred!” is printed to the screen. Now let’s add in the **finally** statement:

```
1 my_dict = {"a":1, "b":2, "c":3}
2
3 try:
4     value = my_dict["a"]
5 except KeyError:
6     print("A KeyError occurred!")
7 else:
8     print("No error occurred!")
9 finally:
10    print("The finally statement ran!")
```

If you run this example, it will execute the `else` and `finally` statements. Most of the time, you won't see the `else` statement used as any code that follows a `try/except` will be executed if no errors were raised. The only good usage of the `else` statement that I've seen mentioned is where you want to execute a **second** piece of code that can **also** raise an error. Of course, if an error is raised in the `else`, then it won't get caught.

Wrapping Up

Now you should be able to handle exceptions in your code. If you find your code raising an exception, you will know how to wrap it in such a way that you can catch the error and exit gracefully or continue without interruption.

Now we're ready to move on and learn about how to work with files in Python.

Chapter 8 - Working with Files

This chapter introduces the topic of reading and writing data to files on your hard drive. You will find that reading and writing files in Python is very easy to do. Let's get started!

How to Read a File

Python has a builtin function called **open** that we can use to open a file for reading. Create a text file name "test.txt" with the following contents:

```
1 This is a test file
2 line 2
3 line 3
4 this line intentionally left blank
```

Here are a couple of examples that show how to use **open** for reading:

```
1 handle = open("test.txt")
2 handle = open(r"C:\Users\mike\py101book\data\test.txt", "r")
```

The first example will open a file named **test.txt** in read-only mode. This is the default mode of the **open** function. Note that we didn't pass a fully qualified path to the file that we wanted to open in the first example. Python will automatically look in the folder that the script is running in for **test.txt**. If it doesn't find it, then you will receive an **IOError**.

The second example does show a fully qualified path to the file, but you'll notice that it begins with an "r". This means that we want Python to treat the string as a raw string. Let's take a moment to see the difference between specifying a raw string versus a regular string:

```
1 >>> print("C:\Users\mike\py101book\data\test.txt")
2 C:\Users\mike\py101book\data      est.txt
3 >>> print(r"C:\Users\mike\py101book\data\test.txt")
4 C:\Users\mike\py101book\data\test.txt
```

As you can see, when we don't specify it as a raw string, we get an invalid path. Why does this happen? Well, as you might recall from the strings chapter, there are certain special characters that

need to be escaped, such as “n” or “t”. In this case, we see there’s a “t” (i.e. a tab), so the string obediently adds a tab to our path and screws it up for us.

The second argument in the second example is also an “r”. This tells `open` that we want to open the file in read-only mode. In other words, it does the same thing as the first example, but it’s more explicit. Now let’s actually read the file!

Put the following lines into a Python script and save it in the same location as your `test.txt` file:

```
1 handle = open("test.txt", "r")
2 data = handle.read()
3 print(data)
4 handle.close()
```

If you run this, it will open the file and read the entire file as a string into the `data` variable. Then we print that data and close the file handle. You should always close a file handle as you never know when another program will want to access it. Closing the file will also help save memory and prevent strange bugs in your programs. You can tell Python to just read a line at a time, to read all the lines into a Python list or to read the file in chunks. The last option is very handy when you are dealing with really large files and you don’t want to read the whole thing in, which might fill up the PC’s memory.

Let’s spend some time looking at different ways to read files.

```
1 handle = open("test.txt", "r")
2 data = handle.readline() # read just one line
3 print(data)
4 handle.close()
```

If you run this example, it will only read the first line of your text file and print it out. That’s not too useful, so let’s try the file handle’s `readlines()` method:

```
1 handle = open("test.txt", "r")
2 data = handle.readlines() # read ALL the lines!
3 print(data)
4 handle.close()
```

After running this code, you will see a Python list printed to the screen because that’s what the `readlines` method returns: a list! Let’s take a moment to learn how to read a file in smaller chunks.

How To Read Files Piece by Piece

The easiest way to read a file in chunks is to use a loop. First we will learn how to read a file line by line and then we will learn how to read it a kilobyte at a time. We will use a `for` loop for our first example:

```
1 handle = open("test.txt", "r")
2
3 for line in handle:
4     print(line)
5
6 handle.close()
```

Here we open up a read-only file handle and then we use a **for** loop to iterate over it. You will find that you can iterate over all kinds of objects in Python (strings, lists, tuples, keys in a dictionary, etc). That was pretty simple, right? Now let's do it in chunks!

```
1 handle = open("test.txt", "r")
2
3 while True:
4     data = handle.read(1024)
5     print(data)
6     if not data:
7         break
```

In this example, we use Python's **while** loop to read a kilobyte of the file at a time. As you probably know, a kilobyte is 1024 bytes or characters. Now let's pretend that we want to read a binary file, like a PDF.

How to Read a Binary File

Reading a binary file is very easy. All you need to do is change the file mode:

```
1 handle = open("test.pdf", "rb")
```

So this time we changed the file mode to **rb**, which means **read-binary**. You will find that you may need to read binary files when you download PDFs from the internet or transfer files from PC to PC.

Writing Files in Python

If you have been following along, you can probably guess what the file-mode flag is for writing files: "**w**" and "**wb**" for write-mode and write-binary-mode. Let's take a look at a simple example, shall we?

CAUTION: When using "**w**" or "**wb**" modes, if the file already exists, it will be overwritten with no warning! You can check if a file exists before you open it by using Python's **os** module. See the **os.path.exists** section in [Chapter 16](#).

```
1 handle = open("output.txt", "w")
2 handle.write("This is a test!")
3 handle.close()
```

That was easy! All we did here was change the file mode to “w” and we called the file handle’s **write** method to write some text to the file. The file handle also has a **writelines** method that will accept a list of strings that the handle will then write to disk in order.

Using the with Operator

Python has a neat little builtin called **with** which you can use to simplify reading and writing files. The **with** operator creates what is known as a **context manager** in Python that will automatically close the file for you when you are done processing it. Let’s see how this works:

```
1 with open("test.txt") as file_handler:
2     for line in file_handler:
3         print(line)
```

The syntax for the **with** operator is a little strange, but you’ll pick it up pretty quickly. Basically what we’re doing is replacing:

```
1 handle = open("test.txt")
```

with this:

```
1 with open("test.txt") as file_handler:
```

You can do all the usual file I/O operations that you would normally do as long as you are within the **with** code block. Once you leave that code block, the file handle will close and you won’t be able to use it any more. Yes, you read that correctly. You no longer have to close the file handle explicitly as the **with** operator does it automatically! See if you can change some of the earlier examples from this chapter so that they use the **with** method too.

Catching Errors

Sometimes when you are working with files, bad things happen. The file is locked because some other process is using it or you have some kind of permission error. When this happens, an **IOError** will probably occur. In this section, we will look at how to catch errors the normal way and how to catch them using the **with** operator. Hint: the idea is basically the same in both!

```
1 try:  
2     file_handler = open("test.txt")  
3     for line in file_handler:  
4         print(line)  
5 except IOError:  
6     print("An IOError has occurred!")  
7 finally:  
8     file_handler.close()
```

In the example above, we wrap the usual code inside of a **try/except** construct. If an error occurs, we print out a message to the screen. Note that we also make sure we close the file using the **finally** statement. Now we're ready to look at how we would do this same thing using **with**:

```
1 try:  
2     with open("test.txt") as file_handler:  
3         for line in file_handler:  
4             print(line)  
5 except IOError:  
6     print("An IOError has occurred!")
```

As you might have guessed, we just wrapped the **with** block in the same way as we did in the previous example. The difference here is that we do not need the **finally** statement as the context manager handles that for us.

Wrapping Up

At this point you should be pretty well versed in dealing with files in Python. Now you know how to read and write files using the older style and the newer **with** style. You will most likely see both styles in the wild. In the next chapter, we will learn how to import other modules that come with Python. This will allow us to create programs using pre-built modules. Let's get started!

Chapter 9 - Importing

Python comes with lots of pre-made code baked in. These pieces of code are known as modules and packages. A module is a single importable Python file whereas a package is made up of two or more modules. A package can be imported the same way a module is. Whenever you save a Python script of your own, you have created a module. It may not be a very useful module, but that's what it is. In this chapter, we will learn how to import modules using several different methods. Let's get started!

import this

Python provides the `import` keyword for importing modules. Let's give it a try:

```
1 import this
```

If you run this code in your interpreter, you should see something like the following as your output:

```
1 The Zen of Python, by Tim Peters
2
3 Beautiful is better than ugly.
4 Explicit is better than implicit.
5 Simple is better than complex.
6 Complex is better than complicated.
7 Flat is better than nested.
8 Sparse is better than dense.
9 Readability counts.
10 Special cases aren't special enough to break the rules.
11 Although practicality beats purity.
12 Errors should never pass silently.
13 Unless explicitly silenced.
14 In the face of ambiguity, refuse the temptation to guess.
15 There should be one-- and preferably only one --obvious way to do it.
16 Although that way may not be obvious at first unless you're Dutch.
17 Now is better than never.
18 Although never is often better than *right* now.
19 If the implementation is hard to explain, it's a bad idea.
20 If the implementation is easy to explain, it may be a good idea.
21 Namespaces are one honking great idea -- let's do more of those!
```

You have found an “Easter egg” in Python known as the “Zen of Python”. It’s actually a sort of an unofficial best practices for Python. The **this** module doesn’t actually do anything, but it provided a fun little way to show how to import something. Let’s actually import something we can use, like the **math** module:

```
1 >>> import math  
2 >>> math.sqrt(4)  
3 2.0
```

Here we imported the **math** module and then we did something kind of new. We called one of its functions, **sqrt** (i.e. square root). To call a method of an imported module, we have to use the following syntax: **module_name.method_name(argument)**. In this example, we found the square root of 4. The **math** module has many other functions that we can use, such as **cos** (cosine), **factorial**, **log** (logarithm), etc. You can call these functions in much the same way you did **sqrt**. The only thing you’ll need to check is if they accept more arguments or not. Now let’s look at another way to import.

Using from to import

Some people don’t like having to preface everything they type with the module name. Python has a solution for that! You can actually import just the functions you want from a module. Let’s pretend that we want to just import the **sqrt** function:

```
1 >>> from math import sqrt  
2 >>> sqrt(16)  
3 4.0
```

This works pretty much exactly how it is read: **from the math module, import the sqrt function**. Let me explain it another way. We use Python’s **from** keyword to import the **sqrt** function **from** the **math** module. You can also use this method to import multiple functions from the **math** function:

```
1 >>> from math import pi, sqrt
```

In this example, we import both **pi** and **sqrt**. If you tried to access **pi** you may have noticed that it’s actually a value and not a function that you can call. It just returns the value of pi. When you do an import, you may end up importing a value, a function or even another module! There’s one more way to import stuff that we need to cover. Let’s find out how to import everything!

Importing Everything!

Python provides a way to import all the functions and values from a module as well. This is actually a **bad** idea as it can contaminate your **namespace**. A namespace is where all your variables live during the life of the program. So let’s say you have your own variable named **sqrt**, like this:

```
1 >>> from math import sqrt  
2 >>> sqrt = 5
```

Now you have just changed the `sqrt` function into a variable that holds the value of 5. This is known as **shadowing**. This becomes especially tricky when you import everything from a module. Let's take a look:

```
>>> from math import * >>> sqrt = 5 >>> sqrt(16) Traceback (most recent call last): File  
<string>, line 1, in <fragment> TypeError: 'int' object is not callable
```

To import everything, instead of specifying a list of items, we just use the “`*`” *wildcard which means we want to import everything*. If we don’t know what’s in the `math`* module, we won’t realize that we’ve just clobbered one of the functions we imported. When we try to call the `sqrt` function after reassigning it to an integer, we find out that it no longer works.

Thus it is recommended that in most cases, you should import items from modules using one of the previous methods mentioned in this chapter. There are a few exceptions to this rule. Some modules are made to be imported using the “`*`” method. One prominent example is Tkinter, a toolkit included with Python that allows you to create desktop user interfaces. The reason that it is supposedly okay to import Tkinter in this way is that the modules are named so that it is unlikely you would reuse one yourself.

Wrapping Up

Now you know all about Python imports. There are dozens of modules included with Python that you can use to give extra functionality to your programs. You can use the builtin modules to query your OS, get information from the Windows Registry, set up logging utilities, parse XML, and much, much more. We will be covering a few of these modules in Part II of this book.

In the next chapter, we will be looking at building our own functions. I think you’ll find this next topic to be very helpful.

Chapter 10 - Functions

A function is a structure that you define. You get to decide if they have arguments or not. You can add keyword arguments and default arguments too. A function is a block of code that starts with the `def` keyword, a name for the function and a colon. Here's a simple example:

```
1 >>> def a_function():
2     print("You just created a function!")
```

This function doesn't do anything except print out some text. To call a function, you need to type out the name of the function followed by an open and close parentheses:

```
1 >>> a_function()
2 You just created a function!
```

Simple, eh?

An Empty Function (the stub)

Sometimes when you are writing out some code, you just want to write the function definitions without putting any code in them. I've done this as kind of an outline. It helps you to see how your application is going to be laid out. Here's an example:

```
1 >>> def empty_function():
2     pass
```

Here's something new: the `pass` statement. It is basically a null operation, which means that when `pass` is executed, nothing happens.

Passing Arguments to a Function

Now we're ready to learn about how to create a function that can accept arguments and also learn how to pass said arguments to the function. Let's create a simple function that can add two numbers together:

```
1 >>> def add(a, b):
2         return a + b
3
4 >>> add(1, 2)
5 3
```

All functions return something. If you don't tell it to return something, then it will return None. In this case, we tell it to return `a + b`. As you can see, we can call the function by passing in two values. If you don't pass enough or you pass too many arguments, then you'll get an error:

```
1 >>> add(1)
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 TypeError: add() takes exactly 2 arguments (1 given)
```

You can also call the function by passing the name of the arguments:

```
1 >>> add(a=2, b=3)
2 5
3 >>> total = add(b=4, a=5)
4 >>> print(total)
5 9
```

You'll notice that it doesn't matter what order you pass them to the function as long as they are named correctly. In the second example, you can see that we assign the result of the function to a variable named `total`. This is the usual way of calling a function as you'll want to do something with the result. You are probably wondering what would happen if we passed in arguments with the wrong names attached. Would it work? Let's find out:

```
1 >>> add(c=5, d=2)
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4 TypeError: add() got an unexpected keyword argument 'c'
```

Whoops! We received an error. This means that we passed in a keyword argument that the function didn't recognize. Coincidentally, keyword arguments are our next topic!

Keyword Arguments

Functions can also accept keyword arguments! They can actually accept both regular arguments and keyword arguments. What this means is that you can specify which keywords are which and pass them in. You saw this behavior in a previous example.

```
1 >>> def keyword_function(a=1, b=2):  
2     return a+b  
3  
4 >>> keyword_function(b=4, a=5)  
5 9
```

You could have also called this function without specifying the keywords. This function also demonstrates the concept of default arguments. How? Well, try calling the function without any arguments at all!

```
1 >>> keyword_function()  
2 3
```

The function returned the number 3! Why? The reason is that **a** and **b** have default values of 1 and 2 respectively. Now let's create a function that has both a regular argument and a couple keyword arguments:

```
1 >>> def mixed_function(a, b=2, c=3):  
2     return a+b+c  
3  
4 >>> mixed_function(b=4, c=5)  
5 Traceback (most recent call last):  
6   File "<string>", line 1, in <fragment>  
7 TypeError: mixed_function() takes at least 1 argument (2 given)  
8 >>> mixed_function(1, b=4, c=5)  
9 10  
10 >>> mixed_function(1)  
11 6
```

There are 3 example cases in the above code. Let's go over each of them. In the first example, we try calling our function using just the keyword arguments. This will give us a confusing error. The Traceback says that our function accepts at least one argument, but that two were given. What's going on here? The fact is that the first argument is required because it's not set to anything, so if you only call the function with the keyword arguments, that causes an error.

For the second example, we call the mixed function with 3 values, naming two of them. This works and gives us the expected result, which was $1+4+5=10$. The third example shows what happens if we only call the function by passing in just one value...the one that didn't have a default. This also works by taking the "1" and adding it to the two default values of "2" and "3" to get a result of "6"! Isn't that cool?

*args and **kwargs

You can also set up functions to accept any number of arguments or keyword arguments by using a special syntax. To get infinite arguments, use `*args` and for infinite keyword arguments, use `**kwargs`. *The “args” and “kwargs” words are not important. That’s just convention. You could have called them *bill and *ted and it would work the same way.* The key here is in the number of asterisks.

*Note: in addition to the convention of `*args` and `**kwargs`, you will also see `a` and `kw` from time to time.*

Let’s take a look at a quick example:

```
1 >>> def many(*args, **kwargs):
2         print(args)
3         print(kwargs)
4
5 >>> many(1, 2, 3, name="Mike", job="programmer")
6 (1, 2, 3)
7 {'job': 'programmer', 'name': 'Mike'}
```

First we create our function using the new syntax and then we call it with three regular arguments and two keyword arguments. The function itself will print out both types of arguments. As you can see, the `args` parameter turns into a tuple and `kwargs` turns into a dictionary. You will see this type of coding used in the Python source and in many 3rd party Python packages.

A Note on Scope and Globals

Python has the concept of **scope** just like most programming languages. Scope will tell us when a variable is available to use and where. If we define the variables inside of a function, those variables can only be used inside that function. Once that function ends, they can no longer be used because they are **out of scope**. Let’s take a look at an example:

```
1 def function_a():
2     a = 1
3     b = 2
4     return a+b
5
6 def function_b():
7     c = 3
8     return a+c
9
10 print(function_a())
11 print(function_b())
```

If you run this code, you will receive the following error:

```
1 NameError: global name 'a' is not defined
```

This is caused because the variable `a` is only defined in the first function and is not available in the second. You can get around this by telling Python that `a` is a **global** variable. Let's take a look at how that's done:

```
1 def function_a():
2     global a
3     a = 1
4     b = 2
5     return a+b
6
7 def function_b():
8     c = 3
9     return a+c
10
11 print(function_a())
12 print(function_b())
```

This code will work because we told Python to make `a` global, which means that that variable is available everywhere in our program. This is usually a bad idea and not recommended. The reason it is not recommended is that it makes it difficult to tell when the variable is defined. Another problem is that when we define a global in one place, we may accidentally redefine its value in another which may cause logic errors later that are difficult to debug.

Coding Tips

One of the biggest problems that new programmers need to learn is the idea of “Don’t Repeat Yourself (DRY)”. This concept is that you should avoid writing the same code more than once. When you find yourself doing that, then you know that chunk of code should go into a function. One great reason to do this is that you will almost certainly need to change that piece of code again in the future and if it’s in multiple places, then you will need to remember where all those locations are AND change them.

Copying and pasting the same chunk of code all over is an example of **spaghetti code**. Try to avoid this as much as possible. You will regret it at some point either because you’ll be the one having to fix it or because you’ll find someone else’s code that you have to maintain with these sorts of problems.

Wrapping Up

You now have the foundational knowledge necessary to use functions effectively. You should practice creating some simple functions and try calling them in different ways. Once you've played around with functions a bit or you just think you thoroughly understand the concepts involved, you can turn to the next chapter on classes.

Chapter 11 - Classes

Everything in Python is an object. That's a very vague statement unless you've taken a computer programming class or two. What this means is that every *thing* in Python has methods and values. The reason is that everything is based on a class. A class is the blueprint of an object. Let's take a look at what I mean:

```
1 >>> x = "Mike"
2 >>> dir(x)
3 ['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
4 '__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
5 '__getslice__', '__gt__', '__hash__', '__init__', '__le__', '__len__', '__lt__',
6 '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__rep\
7 r__',
8 '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__',
9 '_',
10 '__formatter_field_name_split', '__formatter_parser', 'capitalize', 'center', 'cou\
11 nt',
12 'decode', 'encode', 'endswith', 'expandtabs', 'find', 'format', 'index', 'isalnu\
13 m',
14 'isalpha', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper', 'join', 'ljust\
15 ',
16 'lower', 'lstrip', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartiti\
17 on',
18 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 't\
19 itle',
20 'translate', 'upper', 'zfill']
```

Here we have a string assigned to the variable x. It might not look like much, but that string has a lot of methods. If you use Python's `dir` keyword, you can get a list of all the methods you can call on your string. There are 71 methods here! Technically we're not supposed to call the methods that start with underscores directly, so that reduces the total to 38, but that's still a lot of methods! What does this mean though? It means that a string is based on a class and x is an **instance** of that class!

In Python we can create our own classes. Let's get started!

Creating a Class

Creating a class in Python is very easy. Here is a very simple example:

```
1 # Python 2.x syntax
2 class Vehicle(object):
3     """docstring"""
4
5     def __init__(self):
6         """Constructor"""
7         pass
```

This class doesn't do anything in particular, however it is a very good learning tool. For example, to create a class, we need to use Python's `class` keyword, followed by the name of the class. In Python, convention says that the class name should have the first letter capitalized. Next we have an open parentheses followed by the word `object` and a closed parentheses. The `object` is what the class is based on or inheriting from. This is known as the base class or parent class. Most classes in Python are based on `object`. Classes have a special method called `__init__` (for initialization). This method is called whenever you create (or instantiate) an object based on this class. The `__init__` method is only called once and is not to be called again inside the program. Another term for `__init__` is `constructor`, although this term isn't used that much in Python.

You may be wondering why I keep saying `method` instead of `function`. A function changes its name to "method" when it is within a class. You will also notice that every method has to have at least one argument (i.e. `self`), which is not true with a regular function.

In Python 3, we don't need to explicitly say we're inheriting from `object`. Instead, we could have written the above like this:

```
1 # Python 3.x syntax
2 class Vehicle:
3     """docstring"""
4
5     def __init__(self):
6         """Constructor"""
7         pass
```

You will notice that the only difference is that we no longer need the parentheses if we're basing our class on `object`. Let's expand our class definition a bit and give it some attributes and methods.

```
1 class Vehicle(object):
2     """docstring"""
3
4     def __init__(self, color, doors, tires):
5         """Constructor"""
6         self.color = color
7         self.doors = doors
8         self.tires = tires
9
10    def brake(self):
11        """
12            Stop the car
13        """
14        return "Braking"
15
16    def drive(self):
17        """
18            Drive the car
19        """
20        return "I'm driving!"
```

The code above added three attributes and two methods. The three attributes are:

```
1 self.color = color
2 self.doors = doors
3 self.tires = tires
```

Attributes describe the vehicle. So the vehicle has a color, some number of doors and some number of tires. It also has two methods. A method describes what a class does. So in this case, a vehicle can **brake** and **drive**. You may have noticed that all of the methods, including the first one have a funny argument called **self**. Let's talk about that!

What is **self**?

Python classes need a way to refer to themselves. This isn't some kind of narcissistic navel-gazing on the part of the class. Instead, it's a way to tell one instance from another. The word **self** is a way to describe itself, literally. Let's take a look at an example as I always find that helpful when I'm learning something new and strange:

Add the following code to the end of that class you wrote above and save it:

```
1 if __name__ == "__main__":
2     car = Vehicle("blue", 5, 4)
3     print(car.color)
4
5     truck = Vehicle("red", 3, 6)
6     print(truck.color)
```

The conditional statement above is a common way of telling Python that you only want to run the following code if this code is executed as a standalone file. If you had imported your module into another script, then the code underneath the conditional would not run. Anyway, if you run this code, you will create two instances of the Vehicle class: a car instance and a truck instance. Each instance will have its own attributes and methods. This is why when we print out the color of each instance, they are different. The reason is that the class is using that `self` argument to tell itself which is which. Let's change the class a bit to make the methods more unique:

```
1 class Vehicle(object):
2     """docstring"""
3
4     def __init__(self, color, doors, tires, vtype):
5         """Constructor"""
6         self.color = color
7         self.doors = doors
8         self.tires = tires
9         self.vtype = vtype
10
11    def brake(self):
12        """
13        Stop the car
14        """
15        return "%s braking" % self.vtype
16
17    def drive(self):
18        """
19        Drive the car
20        """
21        return "I'm driving a %s %s!" % (self.color, self.vtype)
22
23 if __name__ == "__main__":
24     car = Vehicle("blue", 5, 4, "car")
25     print(car.brake())
26     print(car.drive())
27
```

```
28     truck = Vehicle("red", 3, 6, "truck")
29     print(truck.drive())
30     print(truck.brake())
```

In this example, we pass in another parameter to tell the class which vehicle type we're creating. Then we call each method for each instance. If you run this code, you should see the following output:

```
1 car braking
2 I'm driving a blue car!
3 I'm driving a red truck!
4 truck braking
```

This demonstrates how the instance keeps track of its “self”. You will also notice that we are able to get the values of the attributes from the `__init__` method into the other methods. The reason is because all those attributes are prepended with `self.`. If we hadn't done that, the variables would have gone out of scope at the end of the `__init__` method.

Subclasses

The real power of classes becomes apparent when you get into subclasses. You may not have realized it, but we've already created a subclass when we created a class based on `object`. In other words, we subclassed `object`. Now because `object` isn't very interesting, the previous examples don't really demonstrate the power of subclassing. So let's subclass our `Vehicle` class and find out how all this works.

```
1 class Car(Vehicle):
2     """
3     The Car class
4     """
5
6     #-----
7     def brake(self):
8         """
9         Override brake method
10        """
11        return "The car class is breaking slowly!"
12
13 if __name__ == "__main__":
14     car = Car("yellow", 2, 4, "car")
15     car.brake()
```

```
16     'The car class is breaking slowly!'
17     car.drive()
18     "I'm driving a yellow car!"
```

For this example, we subclassed our **Vehicle** class. You will notice that we didn't include an `__init__` method or a `drive` method. The reason is that when you subclass **Vehicle**, you get all its attributes and methods unless you override them. Thus you will notice that we did override the `brake` method and made it say something different from the default. The other methods we left the same. So when you tell the car to brake, it uses the original method and we learn that we're driving a yellow car. Wasn't that neat?

Using the default values of the parent class is known as **inheriting** or **inheritance**. This is a big topic in Object Oriented Programming (OOP). This is also a simple example of **polymorphism**. Polymorphic classes typically have the same interfaces (i.e. methods, attributes), but they are not aware of each other. In Python land, polymorphism isn't very rigid about making sure the interfaces are exactly the same. Instead, it follows the concept of **duck typing**. The idea of **duck typing** is that if it walks like a duck and talks like a duck, it must be a duck. So in Python, as long as the class has method names that are the same, it doesn't matter if the implementation of the methods are different.

Anyway, you really don't need to know the nitty gritty details of all that to use classes in Python. You just need to be aware of the terminology so if you want to dig deeper, you will be able to. You can find lots of good examples of Python polymorphism that will help you figure out if and how you might use that concept in your own applications.

Now, when you subclass, you can override as much or as little from the parent class as you want. If you completely override it, then you would probably be just as well off just creating a new class.

Wrapping Up

Classes are a little complicated, but they are very powerful. They allow you to use variables across methods which can make code reuse even easier. I would recommend taking a look at Python's source for some excellent examples of how classes are defined and used.

We are now at the end of Part I. Congratulations for making it this far! You now have the tools necessary to build just about anything in Python! In Part II, we will spend time learning about using some of the wonderful modules that Python includes in its distribution. This will help you to better understand the power of Python and familiarize yourself with using the Standard Library. Part II will basically be a set of tutorials to help you on your way to becoming a great Python programmer!

Part II - Learning from the Library

In Part II, you will get an abbreviated tour of some of the Python Standard Library. The reason it's abbreviated is that the Python Standard Library is **HUGE!** So this section is to get you acquainted with using the modules that come with Python. I will be going over the modules I use the most in my day-to-day job and the modules I've seen used by my co-workers. I think this mini-tour will prepare you for digging in on your own.

Let's take a look at what we'll be covering:

- Introspection
- csv
- ConfigParser
- logging
- os
- smtplib / email
- subprocess
- sys
- thread / queues
- time / datetime

|

The first chapter in this section will give you a quick tutorial into Python's **introspection** abilities; basically you'll learn how to make Python tell you about itself, which sounds kind of weird but is really quite valuable to know about. Next we'll learn how to use **ConfigParser**, a neat little module that lets you read and write config files. After that we'll take a look at **logging**. The **os** module can do lots of fun things, but we'll try to focus on the ones that I think you'll find most useful. The **subprocess** allows you to open other processes.

You will find the **sys** module allows you to exit a program, get the Python path, acquire version information, redirect stdout and a whole lot more. The **thread** module allows you to create **threads** in your program. We won't dive too deep into that subject as it can get confusing pretty quickly. The **time** and **datetime** modules allow you to manipulate dates and time in Python, which has many applications when it comes to developing programs.

Let's get started!



image

Chapter 12 - Introspection

Whether you're new to Python, been using it for a few years or you're an expert, knowing how to use Python's introspection capabilities can help your understanding of your code and that new package you just downloaded with the crappy documentation. Introspection is a fancy word that means to observe oneself and ponder one's thoughts, senses, and desires. In Python world, introspection is actually kind of similar. Introspection in this case is to use Python to figure out Python. In this chapter, you can learn how to use Python to give yourself a clue about the code you're working on or trying to learn. Some might even call it a form of debugging.

Here's what we're going to cover:

- type
- dir
- help

The Python Type

You may not know this, but Python may just be your type. Yes, Python can tell you what type of variable you have or what type is returned from a function. It's a very handy little tool. Let's look at a few examples to make it all clear:

```
1 >>> x = "test"
2 >>> y = 7
3 >>> z = None
4 >>> type(x)
5 <class 'str'>
6 >>> type(y)
7 <class 'int'>
8 >>> type(z)
9 <class 'NoneType'>
```

As you can see, Python has a keyword called **type** that can tell you what is what. In my real-life experiences, I've used **type** to help me figure out what is going on when my database data is corrupt or not what I expect. I just add a couple lines and print out each row's data along with its type. This has helped me a lot when I've been dumbfounded by some stupid code I wrote.

The Python Dir

What is dir? Is it something you say when someone says or does something stupid? Not in this context! No, here on Planet Python, the dir keyword is used to tell the programmer what attributes and methods there are in the passed in object. If you forget to pass in an object, dir will return a list of names in the current scope. As usual, this is easier to understand with a few examples.

```

1  >>> dir("test")
2  ['__add__', '__class__', '__contains__', '__delattr__',
3   '__doc__', '__eq__', '__ge__', '__getattribute__',
4   '__getitem__', '__getnewargs__', '__getslice__', '__gt__',
5   '__hash__', '__init__', '__le__', '__len__', '__lt__',
6   '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
7   '__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
8   '__setattr__', '__str__', 'capitalize', 'center',
9   'count', 'decode', 'encode', 'endswith', 'expandtabs',
10  'find', 'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
11  'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
12  'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rsplit',
13  'rstrip', 'split', 'splitlines', 'startswith', 'strip',
14  'swapcase', 'title', 'translate', 'upper', 'zfill']

```

Since everything in Python is an object, we can pass a string to dir and find out what methods it has. Pretty neat, huh? Now let's try it with an imported module:

```

1  >>> import sys
2  >>> dir(sys)
3  ['__displayhook__', '__doc__', '__egginject', '__excepthook__',
4   '__name__', '__plen', '__stderr__', '__stdin__', '__stdout__',
5   '__getframe', 'api_version', 'argv', 'builtin_module_names',
6   'byteorder', 'call_tracing', 'callstats', 'copyright',
7   'displayhook', 'dllhandle', 'exc_clear', 'exc_info',
8   'exc_traceback', 'exc_type', 'exc_value', 'excepthook',
9   'exec_prefix', 'executable', 'exit', 'exitfunc',
10  'getcheckinterval', 'getdefaultencoding', 'getfilesystemencoding',
11  'getrecursionlimit', 'getrefcount', 'getwindowsversion', 'hexversion',
12  'maxint', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
13  'path_importer_cache', 'platform', 'prefix', 'setcheckinterval',
14  'setprofile', 'setrecursionlimit', 'settrace', 'stderr', 'stdin',
15  'stdout', 'version', 'version_info', 'warnoptions', 'winver']

```

Now, that's handy! If you haven't figured it out yet, the `dir` function is extremely handy for those 3rd party packages that you have downloaded (or will soon download) that have little to no documentation. How do you find out about what methods are available in these cases? Well, `dir` will help you figure it out. Of course, sometimes the documentation is in the code itself, which brings us to the builtin help utility.

Python Help!

Python comes with a handy help utility. Just type "help()" (minus the quotes) into a Python shell and you'll see the following directions (Python version may vary)

```
1  >>> help()
2
3  Welcome to Python 2.6!  This is the online help utility.
4
5  If this is your first time using Python, you should definitely check out
6  the tutorial on the Internet at http://www.python.org/doc/tut/.
7
8  Enter the name of any module, keyword, or topic to get help on writing
9  Python programs and using Python modules.  To quit this help utility and
10 return to the interpreter, just type "quit".
11
12 To get a list of available modules, keywords, or topics, type "modules",
13 "keywords", or "topics".  Each module also comes with a one-line summary
14 of what it does; to list the modules whose summaries contain a given word
15 such as "spam", type "modules spam".
16
17 help>
```

Note that you now have a "help>" prompt instead of the ">>>". When you are in help mode, you can explore the various modules, keywords and topics found in Python. Also note that when typing the word "modules", you will see a delay as Python searches its library folder to acquire a list. If you have installed a lot of 3rd party modules, this can take quite a while, so be prepared to go fix yourself a mocha while you wait. Once it's done, just follow the directions and play around with it and I think you'll get the gist.

Wrapping Up

Now you know how to take an unknown module and learn a lot about it by just using some of Python's built-in functionality. You will find yourself using these commands over and over again to help you learn Python. As I mentioned earlier, you will find these tools especially helpful for the 3rd Party modules that don't believe in documentation.

Chapter 13 - The csv Module

The csv module gives the Python programmer the ability to parse CSV (Comma Separated Values) files. A CSV file is a human readable text file where each line has a number of fields, separated by commas or some other delimiter. You can think of each line as a row and each field as a column. The CSV format has no standard, but they are similar enough that the csv module will be able to read the vast majority of CSV files. You can also write CSV files using the csv module.

Reading a CSV File

There are two ways to read a CSV file. You can use the csv module's **reader** function or you can use the **DictReader** class. We will look at both methods. But first, we need to get a CSV file so we have something to parse. There are many websites that provide interesting information in CSV format. We will be using the World Health Organization's (WHO) website to download some information on Tuberculosis. You can go here to get it: <http://www.who.int/tb/country/data/download/en/>. Once you have the file, we'll be ready to start. Ready? Then let's look at some code!

```
1 import csv
2
3 def csv_reader(file_obj):
4     """
5         Read a csv file
6     """
7     reader = csv.reader(file_obj)
8     for row in reader:
9         print(" ".join(row))
10
11 if __name__ == "__main__":
12     csv_path = "TB_data_dictionary_2014-02-26.csv"
13     with open(csv_path, "r") as f_obj:
14         csv_reader(f_obj)
```

Let's take a moment to break this down a bit. First off, we have to actually import the **csv** module. Then we create a very simple function called **csv_reader** that accepts a file object. Inside the function, we pass the file object into the **csv.reader** function, which returns a reader object. The reader object allows iteration, much like a regular file object does. This lets us iterate over each row in the reader object and print out the line of data, minus the commas. This works because each row is a list and we can join each element in the list together, forming one long string.

Now let's create our own CSV file and feed it into the **DictReader** class. Here's a really simple one:

```
1 first_name,last_name,address,city,state,zip_code
2 Tyrese,Hirthe,1404 Turner Ville,Strackeport,NY,19106-8813
3 Jules,Dicki,2410 Estella Cape Suite 061,Lake Nickolasville,ME,00621-7435
4 Dedric,Medhurst,6912 Dayna Shoal,Stiedemannberg,SC,43259-2273
```

Let's save this in a file named `data.csv`. Now we're ready to parse the file using the `DictReader` class. Let's try it out:

```
1 import csv
2
3 def csv_dict_reader(file_obj):
4     """
5         Read a CSV file using csv.DictReader
6     """
7     reader = csv.DictReader(file_obj, delimiter=',')
8     for line in reader:
9         print(line["first_name"]),
10        print(line["last_name"])
11
12 if __name__ == "__main__":
13     with open("data.csv") as f_obj:
14         csv_dict_reader(f_obj)
```

In the example above, we open a file and pass the file object to our function as we did before. The function passes the file object to our `DictReader` class. We tell the `DictReader` that the delimiter is a comma. This isn't actually required as the code will still work without that keyword argument. However, it's a good idea to be explicit so you know what's going on here. Next we loop over the reader object and discover that each line in the reader object is a dictionary. This makes printing out specific pieces of the line very easy.

Now we're ready to learn how to write a csv file to disk.

Writing a CSV File

The `csv` module also has two methods that you can use to write a CSV file. You can use the `writer` function or the `DictWriter` class. We'll look at both of these as well. We will be with the `writer` function. Let's look at a simple example:

```
1 import csv
2
3 def csv_writer(data, path):
4     """
5         Write data to a CSV file path
6     """
7     with open(path, "w", newline='') as csv_file:
8         writer = csv.writer(csv_file, delimiter=',')
9         for line in data:
10             writer.writerow(line)
11
12 if __name__ == "__main__":
13     data = ["first_name,last_name,city".split(","),
14             "Tyrese,Hirthe,Strackeport".split(","),
15             "Jules,Dicki,Lake Nickolasville".split(","),
16             "Dedric,Medhurst,Stiedemannberg".split(",")]
17
18     path = "output.csv"
19     csv_writer(data, path)
```

In the code above, we create a `csv_writer` function that accepts two arguments: data and path. The data is a list of lists that we create at the bottom of the script. We use a shortened version of the data from the previous example and split the strings on the comma. This returns a list. So we end up with a nested list that looks like this:

```
1 [[['first_name', 'last_name', 'city'],
2  ['Tyrese', 'Hirthe', 'Strackeport'],
3  ['Jules', 'Dicki', 'Lake Nickolasville'],
4  ['Dedric', 'Medhurst', 'Stiedemannberg']]
```

The `csv_writer` function opens the path that we pass in and creates a csv writer object. Then we loop over the nested list structure and write each line out to disk. Note that we specified what the delimiter should be when we created the writer object. If you want the delimiter to be something besides a comma, this is where you would set it.

Now we're ready to learn how to write a CSV file using the `DictWriter` class! We're going to use the data from the previous version and transform it into a list of dictionaries that we can feed to our hungry `DictWriter`. Let's take a look:

```
1 import csv
2
3 def csv_dict_writer(path, fieldnames, data):
4     """
5         Writes a CSV file using DictWriter
6     """
7     with open(path, "w", newline='') as out_file:
8         writer = csv.DictWriter(out_file, delimiter=',', fieldnames=fieldnames)
9         writer.writeheader()
10        for row in data:
11            writer.writerow(row)
12
13 if __name__ == "__main__":
14    data = ["first_name,last_name,city",
15            "Tyrese,Hirthe,Strackeport",
16            "Jules,Dicki,Lake Nickolasville",
17            "Dedric,Medhurst,Stiedemannberg"]
18    ]
19    my_list = []
20    fieldnames = data[0]
21    for values in data[1:]:
22        inner_dict = dict(zip(fieldnames, values))
23        my_list.append(inner_dict)
24
25 path = "dict_output.csv"
26 csv_dict_writer(path, fieldnames, my_list)
```

We will start in the second section first. As you can see, we start out with the nested list structure that we had before. Next we create an empty list and a list that contains the field names, which happens to be the first list inside the nested list. Remember, lists are zero-based, so the first element in a list starts at zero! Next we loop over the nested list construct, starting with the second element:

```
1 for values in data[1:]:
2     inner_dict = dict(zip(fieldnames, values))
3     my_list.append(inner_dict)
```

Inside the `for` loop, we use Python builtins to create dictionary. The `zip` method will take two iterators (lists in this case) and turn them into a list of tuples. Here's an example:

```
1 zip(fieldnames, values)
2 [('first_name', 'Dedric'), ('last_name', 'Medhurst'), ('city', 'Stiedemannberg')]
```

Now when you wrap that call in **dict**, it turns that list of tuples into a dictionary. Finally we append the dictionary to the list. When the **for** finishes, you'll end up with a data structure that looks like this:

```
[{'city': 'Strakeport', 'first_name': 'Tyrese', 'last_name': 'Hirthe'},
 {'city': 'Lake Nickolasville', 'first_name': 'Jules', 'last_name': 'Dicki'}, {'city': 'Stiedemannberg', 'first_name': 'Dedric', 'last_name': 'Medhurst'}]
```

At the end of the second session, we call our **csv_dict_writer** function and pass in all the required arguments. Inside the function, we create a **DictWriter** instance and pass it a file object, a delimiter value and our list of field names. Next we write the field names out to disk and loop over the data one row at a time, writing the data to disk. The **DictWriter** class also supports the **writerows** method, which we could have used instead of the loop. The **csv.writer** function also supports this functionality.

You may be interested to know that you can also create **Dialects** with the **csv** module. This allows you to tell the **csv** module how to read or write a file in a very explicit manner. If you need this sort of thing because of an oddly formatted file from a client, then you'll find this functionality invaluable.

Wrapping Up

Now you know how to use the **csv** module to read and write CSV files. There are many websites that put out their data in this format and it is used a lot in the business world. In our next chapter, we will begin learning about the **ConfigParser** module.

Chapter 14 - configparser

Configuration files are used by both users and programmers. They are usually used for storing your application's settings or even your operating system's settings. Python's core library includes a module called configparser that you can use for creating and interacting with configuration files. We'll spend a few minutes learning how it works in this chapter.

Creating a Config File

Creating a config file with configparser is extremely simple. Let's create some code to demonstrate:

```
1 import configparser
2
3 def createConfig(path):
4     """
5     Create a config file
6     """
7     config = configparser.ConfigParser()
8     config.add_section("Settings")
9     config.set("Settings", "font", "Courier")
10    config.set("Settings", "font_size", "10")
11    config.set("Settings", "font_style", "Normal")
12    config.set("Settings", "font_info",
13               "You are using %(font)s at %(font_size)s pt")
14
15    with open(path, "w") as config_file:
16        config.write(config_file)
17
18
19 if __name__ == "__main__":
20     path = "settings.ini"
21     createConfig(path)
```

The code above will create a config file with one section labelled Settings that will contain four options: font, font_size, font_style and font_info. Also note that in Python 3 we only need to specify that we're writing the file in write-only mode, i.e. "w". Back in Python 2, we had to use "wb" to write in binary mode.

How to Read, Update and Delete Options

Now we're ready to learn how to read the config file, update its options and even how to delete options. In this case, it's easier to learn by actually writing some code! Just add the following function to the code that you wrote above.

```
1 import configparser
2 import os
3
4 def crudConfig(path):
5     """
6         Create, read, update, delete config
7     """
8     if not os.path.exists(path):
9         createConfig(path)
10
11    config = configparser.ConfigParser()
12    config.read(path)
13
14    # read some values from the config
15    font = config.get("Settings", "font")
16    font_size = config.get("Settings", "font_size")
17
18    # change a value in the config
19    config.set("Settings", "font_size", "12")
20
21    # delete a value from the config
22    config.remove_option("Settings", "font_style")
23
24    # write changes back to the config file
25    with open(path, "w") as config_file:
26        config.write(config_file)
27
28
29 if __name__ == "__main__":
30     path = "settings.ini"
31     crudConfig(path)
```

This code first checks to see if the path for the config file exists. If it does not, then it uses the createConfig function we created earlier to create it. Next we create a ConfigParser object and pass it the config file path to read. To read an option in your config file, we call our ConfigParser object's get method, passing it the section name and the option name. This will return the option's value. If

you want to change an option's value, then you use the `set` method, where you pass the section name, the option name and the new value. Finally, you can use the `remove_option` method to remove an option.

In our example code, we change the value of `font_size` to `12` and we remove the `font_style` option completely. Then we write the changes back out to disk.

This isn't really a good example though as you should never have a function that does everything like this one does. So let's split it up into a series of functions instead:

```
1 import configparser
2 import os
3
4 def create_config(path):
5     """
6     Create a config file
7     """
8     config = configparser.ConfigParser()
9     config.add_section("Settings")
10    config.set("Settings", "font", "Courier")
11    config.set("Settings", "font_size", "10")
12    config.set("Settings", "font_style", "Normal")
13    config.set("Settings", "font_info",
14               "You are using %(font)s at %(font_size)s pt")
15
16    with open(path, "w") as config_file:
17        config.write(config_file)
18
19
20 def get_config(path):
21     """
22     Returns the config object
23     """
24     if not os.path.exists(path):
25         create_config(path)
26
27     config = configparser.ConfigParser()
28     config.read(path)
29     return config
30
31
32 def get_setting(path, section, setting):
33     """
34     Print out a setting
```

```
35      """
36      config = get_config(path)
37      value = config.get(section, setting)
38      msg = "{section} {setting} is {value}".format(
39          section=section, setting=setting, value=value)
40      print(msg)
41      return value
42
43
44 def update_setting(path, section, setting, value):
45     """
46     Update a setting
47     """
48     config = get_config(path)
49     config.set(section, setting, value)
50     with open(path, "w") as config_file:
51         config.write(config_file)
52
53
54 def delete_setting(path, section, setting):
55     """
56     Delete a setting
57     """
58     config = get_config(path)
59     config.remove_option(section, setting)
60     with open(path, "w") as config_file:
61         config.write(config_file)
62
63 if __name__ == "__main__":
64     path = "settings.ini"
65     font = get_setting(path, 'Settings', 'font')
66     font_size = get_setting(path, 'Settings', 'font_size')
67
68     update_setting(path, "Settings", "font_size", "12")
69
70     delete_setting(path, "Settings", "font_style")
```

This example is heavily refactored compared to the first one. I even went so far as to name the functions following PEP8. Each function should be self-explanatory and self-contained. Instead of putting all the logic into one function, we separate it out into multiple functions and then demonstrate their functionality within the bottom if statement. Now you can import the module and use it yourself.

Please note that this example has the section hard-coded, so you will want to update this example further to make it completely generic.

How to Use Interpolation

The configparser module also allows **interpolation**, which means you can use some options to build another option. We actually do this with the font_info option in that its value is based on the font and font_size options. We can change an interpolated value using a Python dictionary. Let's take a moment to demonstrate both of these facts.

```
1 import configparser
2 import os
3
4 def interpolationDemo(path):
5     if not os.path.exists(path):
6         createConfig(path)
7
8     config = configparser.ConfigParser()
9     config.read(path)
10
11    print(config.get("Settings", "font_info"))
12
13    print(config.get("Settings", "font_info",
14                      vars={"font": "Arial", "font_size": "100"}))
15
16
17 if __name__ == "__main__":
18     path = "settings.ini"
19     interpolationDemo(path)
```

If you run this code, you should see output similar to the following:

```
1 You are using Courier at 12 pt
2 You are using Arial at 100 pt
```

Wrapping Up

At this point, you should know enough about the **configparser**'s capabilities that you can use it for your own projects. There's another project called **ConfigObj** that isn't a part of Python that you might also want to check out. **ConfigObj** is more flexible and has more features than **configparser**. But if you're in a pinch or your organization doesn't allow 3rd party packages, then **configparser** will probably fit the bill.

Chapter 15 - Logging

Python provides a very powerful logging library in its standard library. A lot of programmers use print statements for debugging (myself included), but you can also use logging to do this. It's actually cleaner to use logging as you won't have to go through all your code to remove the print statements. In this chapter we'll cover the following topics:

- Creating a simple logger
- How to log from multiple modules
- Log formatting
- Log configuration

By the end of this chapter, you should be able to confidently create your own logs for your applications. Let's get started!

Creating a Simple Logger

Creating a log with the logging module is easy and straight-forward. It's easiest to just look at a piece of code and then explain it, so here's some code for you to read:

```
1 import logging
2
3 # add filemode="w" to overwrite
4 logging.basicConfig(filename="sample.log", level=logging.INFO)
5
6 logging.debug("This is a debug message")
7 logging.info("Informational message")
8 logging.error("An error has happened!")
```

As you might expect, to access the logging module you have to first import it. The easiest way to create a log is to use the logging module's basicConfig function and pass it some keyword arguments. It accepts the following: filename, filemode, format, datefmt, level and stream. In our example, we pass it a file name and the logging level, which we set to INFO. There are five levels of logging (in ascending order): DEBUG, INFO, WARNING, ERROR and CRITICAL. By default, if you run this code multiple times, it will append to the log if it already exists. If you would rather have your logger overwrite the log, then pass in a `filemode="w"` as mentioned in the comment in the code. Speaking of running the code, this is what you should get if you ran it once:

```
1 INFO:root:Informational message
2 ERROR:root>An error has happened!
```

Note that the debugging message isn't in the output. That is because we set the level at INFO, so our logger will only log if it's a INFO, WARNING, ERROR or CRITICAL message. The root part just means that this logging message is coming from the root logger or the main logger. We'll look at how to change that so it's more descriptive in the next section. If you don't use `basicConfig`, then the logging module will output to the console / stdout.

The logging module can also log some exceptions to file or wherever you have it configured to log to. Here's an example:

```
1 import logging
2
3 logging.basicConfig(filename="sample.log", level=logging.INFO)
4 log = logging.getLogger("ex")
5
6 try:
7     raise RuntimeError
8 except RuntimeError:
9     log.exception("Error!")
```

Let's break this down a bit. Here we use the `logging` module's `getLogger` method to return a logger object that has been named `ex`. This is handy when you have multiple loggers in one application as it allows you to identify which messages came from each logger. This example will force a `RuntimeError` to be raised, catch the error and log the entire traceback to file, which can be very handy when debugging.

How to log From Multiple Modules (and Formatting too!)

The more you code, the more often you end up creating a set of custom modules that work together. If you want them all to log to the same place, then you've come to the right place. We'll look at the simple way and then show a more complex method that's also more customizable. Here's one easy way to do it:

```
1 import logging
2 import otherMod
3
4 def main():
5     """
6     The main entry point of the application
7     """
8     logging.basicConfig(filename="mySnake.log", level=logging.INFO)
9     logging.info("Program started")
10    result = otherMod.add(7, 8)
11    logging.info("Done!")
12
13 if __name__ == "__main__":
14     main()
```

Here we import logging and a module of our own creation (“otherMod”). Then we create our log file as before. The other module looks like this:

```
1 # otherMod.py
2 import logging
3
4 def add(x, y):
5     """
6     logging.info("added %s and %s to get %s" % (x, y, x+y))
7     return x+y
```

If you run the main code, you should end up with a log that has the following contents:

```
1 INFO:root:Program started
2 INFO:root:added 7 and 8 to get 15
3 INFO:root:Done!
```

Do you see the problem with doing it this way? You can’t really tell very easily where the log messages are coming from. This will only get more confusing the more modules there are that write to this log. So we need to fix that. That brings us to the complex way of creating a logger. Let’s take a look at a different implementation:

```
1 import logging
2 import otherMod2
3
4 def main():
5     """
6     The main entry point of the application
7     """
8     logger = logging.getLogger("exampleApp")
9     logger.setLevel(logging.INFO)
10
11    # create the logging file handler
12    fh = logging.FileHandler("new_snake.log")
13
14    formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(me\
15 ssage)s')
16    fh.setFormatter(formatter)
17
18    # add handler to logger object
19    logger.addHandler(fh)
20
21    logger.info("Program started")
22    result = otherMod2.add(7, 8)
23    logger.info("Done!")
24
25 if __name__ == "__main__":
26     main()
```

Here we create a logger instance named “exampleApp”. We set its logging level, create a logging file handler object and a logging Formatter object. The file handler has to set the formatter object as its formatter and then the file handler has to be added to the logger instance. The rest of the code in main is mostly the same. Just note that instead of “logging.info”, it’s “logger.info” or whatever you call your logger variable. Here’s the updated otherMod2 code:

```
1 # otherMod2.py
2 import logging
3
4 module_logger = logging.getLogger("exampleApp.otherMod2")
5
6 def add(x, y):
7     """
8     logger = logging.getLogger("exampleApp.otherMod2.add")
9     logger.info("added %s and %s to get %s" % (x, y, x+y))
10    return x+y
```

Note that here we have two loggers defined. We don't do anything with the module_logger in this case, but we do use the other one. If you run the main script, you should see the following output in your file:

```
1 2012-08-02 15:37:40,592 - exampleApp - INFO - Program started
2 2012-08-02 15:37:40,592 - exampleApp.otherMod2.add - INFO - added 7 and 8 to get\
3 15
4 2012-08-02 15:37:40,592 - exampleApp - INFO - Done!
```

You will notice that all references to root have been removed. Instead it uses our **Formatter** object which says that we should get a human readable time, the logger name, the logging level and then the message. These are actually known as **LogRecord** attributes. For a full list of **LogRecord** attributes, see the documentation¹² as there are too many to list here.

Configuring Logs for Work and Pleasure

The logging module can be configured 3 different ways. You can configure it using methods (loggers, formatters, handlers) like we did earlier in this article; you can use a configuration file and pass it to fileConfig(); or you can create a dictionary of configuration information and pass it to the dictConfig() function. Let's create a configuration file first and then we'll look at how to execute it with Python. Here's an example config file:

```
1 [loggers]
2 keys=root,exampleApp
3
4 [handlers]
5 keys=fileHandler, consoleHandler
6
7 [formatters]
8 keys=myFormatter
9
10 [logger_root]
11 level=CRITICAL
12 handlers=consoleHandler
13
14 [logger_exampleApp]
15 level=INFO
16 handlers=fileHandler
17 qualname=exampleApp
```

¹²<https://docs.python.org/3/library/logging.html#logrecord-attributes>

```
18 [handler_consoleHandler]
19 class=StreamHandler
20 level=DEBUG
21 formatter=myFormatter
22 args=(sys.stdout,)
23
24 [handler_fileHandler]
25 class=FileHandler
26 formatter=myFormatter
27 args=("config.log",)
28
29 [formatter_myFormatter]
30 format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
31 datefmt=
32
```

You'll notice that we have two loggers specified: root and exampleApp. For whatever reason, "root" is required. If you don't include it, Python will raise a `ValueError` from config.py's `_install_loggers` function, which is a part of the logging module. If you set the root's handler to `fileHandler`, then you'll end up doubling the log output, so to keep that from happening, we send it to the console instead. Study this example closely. You'll need a section for every key in the first three sections. Now let's see how we load it in the code:

```
1 # log_with_config.py
2 import logging
3 import logging.config
4 import otherMod2
5
6 def main():
7     """
8     Based on http://docs.python.org/howto/logging.html#configuring-logging
9     """
10    logging.config.fileConfig('logging.conf')
11    logger = logging.getLogger("exampleApp")
12
13    logger.info("Program started")
14    result = otherMod2.add(7, 8)
15    logger.info("Done!")
16
17 if __name__ == "__main__":
18     main()
```

As you can see, all you need to do is pass the config file path to `logging.config.fileConfig`. You'll also notice that we don't need all that setup code any more as that's all in the config file. Also we can just import the `otherMod2` module with no changes. Anyway, if you run the above, you should end up with the following in your log file:

```
1 2012-08-02 18:23:33,338 - exampleApp - INFO - Program started
2 2012-08-02 18:23:33,338 - exampleApp.otherMod2.add - INFO - added 7 and 8 to get\
3 15
4 2012-08-02 18:23:33,338 - exampleApp - INFO - Done!
```

As you might have guessed, it's very similar to the other example. Now we'll move on to the other config method. The dictionary configuration method (`dictConfig`) wasn't added until Python 2.7, so make sure you have that or a later version, otherwise you won't be able to follow along. It's not well documented how this works. In fact, the examples in the documentation show YAML for some reason. Anyway, here's some working code for you to look over:

```
1 # log_with_config.py
2 import logging
3 import logging.config
4 import otherMod2
5
6 def main():
7     """
8     Based on http://docs.python.org/howto/logging.html#configuring-logging
9     """
10    dictLogConfig = {
11        "version":1,
12        "handlers":{
13            "fileHandler":{
14                "class":"logging.FileHandler",
15                "formatter":"myFormatter",
16                "filename":"config2.log"
17            }
18        },
19        "loggers":{
20            "exampleApp":{
21                "handlers":["fileHandler"],
22                "level":"INFO",
23            }
24        },
25
26        "formatters":{
```

```
27     "myFormatter":{  
28         "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"  
29     }  
30 }  
31 }  
32  
33 logging.config.dictConfig(dictLogConfig)  
34  
35 logger = logging.getLogger("exampleApp")  
36  
37 logger.info("Program started")  
38 result = otherMod2.add(7, 8)  
39 logger.info("Done!")  
40  
41 if __name__ == "__main__":  
42     main()
```

If you run this code, you'll end up with the same output as the previous method. Note that you don't need the "root" logger when you use a dictionary configuration.

Wrapping Up

At this point you should know how to get started using loggers and how to configure them in several different ways. You should also have gained the knowledge of how to modify the output using the Formatter object. The logging module is very handy for troubleshooting what went wrong in your application. Be sure to spend some time practicing with this module before writing a large application.

In the next chapter we will be looking at how to use the `os` module.

Chapter 16 - The os Module

The **os** module has many uses. We won't be covering everything that it can do. Instead, we will get an overview of its uses and we'll also take a look at one of its sub-modules, known as **os.path**. Specifically, we will be covering the following:

- `os.name`
- `os.environ`
- `os.chdir()`
- `os.getcwd()`
- `os.getenv()`
- `os.putenv()`
- `os.mkdir()`
- `os.makedirs()`
- `os.remove()`
- `os.rename()`
- `os.rmdir()`
- `os.startfile()`
- `os.walk()`
- `os.path`

That looks like a lot to cover, but there is at least ten times as many other actions that the **os** module can do. This chapter is just going to give you a little taste of what's available. To use any of the methods mentioned in this section, you will need to import the **os** module, like this:

```
1 import os
```

Let's start learning how to use this module!

os.name

The **os** module has both callable functions and normal values. In the case of **os.name**, it is just a value. When you access `os.name`, you will get information about what platform you are running on. You will receive one of the following values: 'posix', 'nt', 'os2', 'ce', 'java', 'riscos'. Let's see what we get when we run it on Windows 7:

```
1 >>> import os  
2 >>> os.name  
3 'nt'
```

This tells us that our Python instance is running on a Windows box. How do we know this? Because Microsoft started calling its operating system NT many years ago. For example, Windows 7 is also known as Windows NT 6.1.

os.environ, os.getenv() and os.putenv()

The `os.environ` value is known as a `mapping` object that returns a dictionary of the user's environmental variables. You may not know this, but every time you use your computer, some environment variables are set. These can give you valuable information, such as number of processors, type of CPU, the computer name, etc. Let's see what we can find out about our machine:

```
1 >>> import os  
2 >>> os.environ  
3 { 'ALLUSERSPROFILE': 'C:\\\\ProgramData',  
4   'APPDATA': 'C:\\\\Users\\\\mike\\\\AppData\\\\Roaming',  
5   'CLASSPATH': '.;C:\\\\Program Files\\\\QuickTime\\\\QTSystem\\\\QTJava.zip',  
6   'COMMONPROGRAMFILES': 'C:\\\\Program Files\\\\Common Files',  
7   'COMPUTERNAME': 'MIKE-PC',  
8   'COMSPEC': 'C:\\\\Windows\\\\system32\\\\cmd.exe',  
9   'FP_NO_HOST_CHECK': 'NO',  
10  'HOMEDRIVE': 'C:',  
11  'HOMEPATH': '\\\\Users\\\\mike',  
12  'LOCALAPPDATA': 'C:\\\\Users\\\\mike\\\\AppData\\\\Local',  
13  'LOGONSERVER': '\\\\\\\\MIKE-PC',  
14  'NUMBER_OF_PROCESSORS': '2',  
15  'OS': 'Windows_NT',  
16  'PATHEXT': '.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;.JSE;.WSF;.WSH;.MSC',  
17  'PROCESSOR_ARCHITECTURE': 'x86',  
18  'PROCESSOR_IDENTIFIER': 'x86 Family 6 Model 15 Stepping 13, GenuineIntel',  
19  'PROCESSOR_LEVEL': '6',  
20  'PROGRAMDATA': 'C:\\\\ProgramData',  
21  'PROGRAMFILES': 'C:\\\\Program Files',  
22  'PSMODULEPATH': 'C:\\\\Windows\\\\system32\\\\WindowsPowerShell\\\\v1.0\\\\Modules\\\\',  
23  'PUBLIC': 'C:\\\\Users\\\\Public',  
24  'PYTHONIOENCODING': 'cp437',  
25  'QTJAVA': 'C:\\\\Program Files\\\\QuickTime\\\\QTSystem\\\\QTJava.zip',  
26  'SESSIONNAME': 'Console',
```

```
27 'SYSTEMDRIVE': 'C:',
28 'SYSTEMROOT': 'C:\\Windows',
29 'TEMP': 'C:\\Users\\mike\\AppData\\Local\\Temp',
30 'TMP': 'C:\\Users\\mike\\AppData\\Local\\Temp',
31 'USERDOMAIN': 'mike-PC',
32 'USERNAME': 'mike',
33 'USERPROFILE': 'C:\\Users\\mike',
34 'VBOX_INSTALL_PATH': 'C:\\Program Files\\Oracle\\VirtualBox\\',
35 'VS90COMNTOOLS': 'C:\\Program Files\\Microsoft Visual Studio 9.0\\Common7\\Tool\\
36 s\\',
37 'WINDIR': 'C:\\Windows',
38 'WINDOWS_TRACING_FLAGS': '3',
39 'WINDOWS_TRACING_LOGFILE': 'C:\\BVTBin\\Tests\\installpackage\\csilogfile.log',
40 'WINGDB_ACTIVE': '1',
41 'WINGDB_PYTHON': 'c:\\python27\\python.exe',
42 'WINGDB_SPAWNCOOKIE': 'rvlxwsGdD7SHYIjm'}
```

Your output won't be the same as mine as everyone's PC configuration is a little different, but you'll see something similar. As you may have noticed, this returned a dictionary. That means you can access the environmental variables using your normal dictionary methods. Here's an example:

```
1 >>> print(os.environ["TMP"])
2 'C:\\Users\\mike\\AppData\\Local\\Temp'
```

You could also use the `os.getenv` function to access this environmental variable:

```
1 >>> os.getenv("TMP")
2 'C:\\Users\\mike\\AppData\\Local\\Temp'
```

The benefit of using `os.getenv()` instead of the `os.environ` dictionary is that if you happen to try to access an environmental variable that doesn't exist, the `getenv` function will just return `None`. If you did the same thing with `os.environ`, you would receive an error. Let's give it a try so you can see what happens:

```
1 >>> os.environ["TMP2"]
2 Traceback (most recent call last):
3   File "<pyshell#1>", line 1, in <module>
4     os.environ["TMP2"]
5   File "C:\Python27\lib\os.py", line 423, in __getitem__
6     return self.data[key.upper()]
7 KeyError: 'TMP2'
8
9 >>> print(os.getenv("TMP2"))
10 None
```

os.chdir() and os.getcwd()

The `os.chdir` function allows us to change the directory that we're currently running our Python session in. If you want to actually know what path you are currently in, then you would call `os.getcwd()`. Let's try them both out:

```
1 >>> os.getcwd()
2 'C:\\Python27'
3 >>> os.chdir(r"c:\\Users\\mike\\Documents")
4 >>> os.getcwd()
5 'c:\\Users\\mike\\Documents'
```

The code above shows us that we started out in the Python directory by default when we run this code in IDLE. Then we change folders using `os.chdir()`. Finally we call `os.getcwd()` a second time to make sure that we changed to the folder successfully.

os.mkdir() and os.makedirs()

You might have guessed this already, but the two methods covered in this section are used for creating directories. The first one is `os.mkdir()`, which allows us to create a single folder. Let's try it out:

```
1 >>> os.mkdir("test")
2 >>> path = r'C:\\Users\\mike\\Documents\\pytest'
3 >>> os.mkdir(path)
```

The first line of code will create a folder named `test` in the current directory. You can use the methods in the previous section to figure out where you just ran your code if you've forgotten. The second

example assigns a path to a variable and then we pass the path to `os.mkdir()`. This allows you to create a folder anywhere on your system that you have permission to.

The `os.makedirs()` function will create all the intermediate folders in a path if they don't already exist. Basically this means that you can create a path that has nested folders in it. I find myself doing this a lot when I create a log file that is in a dated folder structure, like Year/Month/Day. Let's look at an example:

```
1 >>> path = r'C:\Users\mike\Documents\pytest\2014\02\19'  
2 >>> os.makedirs(path)
```

What happened here? This code just created a bunch of folders! If you still had the `pytest` folder in your system, then it just added a `2014` folder with another folder inside of it which also contained a folder. Try it out for yourself using a valid path on your system.

os.remove() and os.rmdir()

The `os.remove()` and `os.rmdir()` functions are used for deleting files and directories respectively. Let's look at an example of `os.remove()`:

```
1 >>> os.remove("test.txt")
```

This code snippet will attempt to remove a file named `test.txt` from your current working directory. If it cannot find the file, you will likely receive some sort of error. You will also receive an error if the file is in use (i.e. locked) or you don't have permission to delete the file. You might also want to check out `os.unlink`, which does the same thing. The term `unlink` is the traditional Unix name for this procedure.

Now let's look at an example of `os.rmdir()`:

```
1 >>> os.rmdir("pytest")
```

The code above will attempt to remove a directory named `pytest` from your current working directory. If it's successful, you will see that the directory no longer exists. An error will be raised if the directory does not exist, you do not have permission to remove it or if the directory is not empty. You might also want to take a look at `os.removedirs()` which can remove nested empty directories recursively.

os.rename(src, dst)

The `os.rename()` function will rename a file or folder. Let's take a look at an example where we rename a file:

```
1 >>> os.rename("test.txt", "pytest.txt")
```

In this example, we tell **os.rename** to rename a file named **test.txt** to **pytest.txt**. This occurs in our current working directory. You will see an error occur if you try to rename a file that doesn't exist or that you don't have the proper permission to rename the file.

There is also an **os.renames** function that recursively renames a directory or file.

os.startfile()

The **os.startfile()** method allows us to “start” a file with its associated program. In other words, we can open a file with it’s associated program, just like when you double-click a PDF and it opens in Adobe Reader. Let’s give it a try!

```
1 >>> os.startfile(r'C:\Users\mike\Documents\labels.pdf')
```

In the example above, I pass a fully qualified path to **os.startfile** that tells it to open a file called **labels.pdf**. On my machine, this will open the PDF in Adobe Reader. You should try opening your own PDFs, MP3s, and photos using this method to see how it works.

os.walk()

The **os.walk()** method gives us a way to iterate over a root level path. What this means is that we can pass a path to this function and get access to all its sub-directories and files. Let’s use one of the Python folders that we have handy to test this function with. We’ll use: C:\Python27\Tools

```
1 >>> path = r'C:\Python27\Tools'
2 >>> for root, dirs, files in os.walk(path):
3     print(root)
4
5 C:\Python27\Tools
6 C:\Python27\Tools\i18n
7 C:\Python27\Tools\pynche
8 C:\Python27\Tools\pynche\X
9 C:\Python27\Tools\Scripts
10 C:\Python27\Tools\versioncheck
11 C:\Python27\Tools\webchecker
```

If you want, you can also loop over **dirs** and **files** too. Here’s one way to do it:

```
1 >>> for root, dirs, files in os.walk(path):
2     print(root)
3     for _dir in dirs:
4         print(_dir)
5     for _file in files:
6         print(_file)
```

This piece of code will print a lot of stuff out, so I won't be showing its output here, but feel free to give it a try. Now we're ready to learn about working with paths!

os.path

The **os.path** sub-module of the **os** module has lots of great functionality built into it. We'll be looking at the following functions:

- `basename`
- `dirname`
- `exists`
- `isdir` and `isfile`
- `join`
- `split`

There are lots of other functions in this sub-module. You are welcome to go read about them in the Python documentation, section 10.1.

os.path.basename

The **basename** function will return just the filename of a path. Here is an example:

```
1 >>> os.path.basename(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 'ChipViewer.py'
```

I have found this useful whenever I need to use a filename for naming some related file, such as a log file. This happens a lot when I'm processing a data file.

os.path.dirname

The **dirname** function will return just the directory portion of the path. It's easier to understand if we take a look at some code:

```
1 >>> os.path.dirname(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 'C:\\Python27\\Tools\\pynche'
```

In this example, we just get the directory path back. This is also useful when you want to store other files next to the file you're processing, like the aforementioned log file.

os.path.exists

The **exists** function will tell you if a path exists or not. All you have to do is pass it a path. Let's take a look:

```
1 >>> os.path.exists(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 True
3 >>> os.path.exists(r'C:\Python27\Tools\pynche\fake.py')
4 False
```

In the first example, we pass the **exists** function a real path and it returns **True**, which means that the path exists. In the second example, we passed it a bad path and it told us that the path did not exist by returning **False**.

os.path.isdir / os.path.isfile

The **isdir** and **.isfile** methods are closely related to the **exists** method in that they also test for existence. However, **isdir** only checks if the path is a directory and **.isfile** only checks if the path is a file. If you want to check if a path exists regardless of whether it is a file or a directory, then you'll want to use the **exists** method. Anyway, let's study some examples:

```
1 >>> os.path.isfile(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 True
3 >>> os.path.isdir(r'C:\Python27\Tools\pynche\ChipViewer.py')
4 False
5 >>> os.path.isdir(r'C:\Python27\Tools\pynche')
6 True
7 >>> os.path.isfile(r'C:\Python27\Tools\pynche')
8 False
```

Take a moment to study this set of examples. In the first one we pass a path to a file and check if the path is really a file. Then the second example checks the same path to see if it's a directory. You can see for yourself how that turned out. Then in the last two examples, we switched things up a bit by passing a path to a directory to the same two functions. These examples demonstrate how these two functions work.

os.path.join

The **join** method give you the ability to join one or more path components together using the appropriate separator. For example, on Windows, the separator is the backslash, but on Linux, the separator is the forward slash. Here's how it works:

```
1 >>> os.path.join(r'C:\Python27\Tools\pynche', 'ChipViewer.py')
2 'C:\\Python27\\Tools\\pynche\\ChipViewer.py'
```

In this example, we joined a directory path and a file path together to get a fully qualified path. Note however that the **join** method does **not** check if the result actually exists!

os.path.split

The **split** method will split a path into a tuple that contains the directory and the file. Let's take a look:

```
1 >>> os.path.split(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 ('C:\\Python27\\Tools\\pynche', 'ChipViewer.py')
```

This example shows what happens when we path in a path with a file. Let's see what happens if the path doesn't have a filename on the end:

```
1 >>> os.path.split(r'C:\Python27\Tools\pynche')
2 ('C:\\Python27\\Tools', 'pynche')
```

As you can see, it took the path and split it in such a way that the last sub-folder became the second element of the tuple with the rest of the path in the first element.

For our final example, I thought you might like to see a common use case of the **split**:

```
1 >>> dirname, fname = os.path.split(r'C:\Python27\Tools\pynche\ChipViewer.py')
2 >>> dirname
3 'C:\\Python27\\Tools\\pynche'
4 >>> fname
5 'ChipViewer.py'
```

This shows how to do multiple assignment. When you split the path, it returns a two-element tuple. Since we have two variables on the left, the first element of the tuple is assigned to the first variable and the second element to the second variable.

Wrapping Up

At this point you should be pretty familiar with the **os** module. In this chapter you learned the following:

- how to work with environment variables
- change directories and discover your current working directory
- create and remove folders and files
- rename files / folders
- start a file with its associated application
- walk a directory
- work with paths

There are a lot of other functions in the **os** module that are not covered here. Be sure to read the documentation to see what else you can do. In the next chapter, we will be learning about the **email** and **smtplib** modules.

Chapter 17 - The `email` / `smtplib` Module

Python provides a couple of really nice modules that we can use to craft emails with. They are the `email` and `smtplib` modules. Instead of going over various methods in these two modules, we'll spend some time learning how to actually use these modules. Specifically, we'll be covering the following:

- The basics of emailing
- How to send to multiple addresses at once
- How to send email using the TO, CC and BCC lines
- How to add an attachment and a body using the `email` module

|

Let's get started!

Email Basics - How to Send an Email with `smtplib`

The `smtplib` module is very intuitive to use. Let's write a quick example that shows how to send an email. Save the following code to a file on your hard drive:

```
1 import smtplib
2
3 HOST = "mySMTP.server.com"
4 SUBJECT = "Test email from Python"
5 TO = "mike@someAddress.org"
6 FROM = "python@mydomain.com"
7 text = "Python 3.4 rules them all!"
8
9 BODY = "\r\n".join((
10     "From: %s" % FROM,
11     "To: %s" % TO,
12     "Subject: %s" % SUBJECT ,
13     "",
14     text
15 ))
```

```
16
17 server = smtplib.SMTP(HOST)
18 server.sendmail(FROM, [TO], BODY)
19 server.quit()
```

We imported two modules, **smtplib** and the **string** module. Two thirds of this code is used for setting up the email. Most of the variables are pretty self-explanatory, so we'll focus on the odd one only, which is **BODY**. Here we use the **string** module to combine all the previous variables into a single string where each lines ends with a carriage return ("r") plus new line ("n"). If you print **BODY** out, it would look like this:

```
1 'From: python@mydomain.com\r\nTo: mike@mydomain.com\r\nSubject: Test email from \
2 Python\r\n\r\nblah blah'
```

After that, we set up a server connection to our host and then we call the **smtplib** module's **sendmail** method to send the email. Then we disconnect from the server. You will note that this code doesn't have a username or password in it. If your server requires authentication, then you'll need to add the following code:

```
1 server.login(username, password)
```

This should be added right after you create the **server** object. Normally, you would want to put this code into a function and call it with some of these parameters. You might even want to put some of this information into a config file. Let's put this code into a function.

```
1 import smtplib
2
3 def send_email(host, subject, to_addr, from_addr, body_text):
4     """
5         Send an email
6     """
7     BODY = "\r\n".join((
8         "From: %s" % from_addr,
9         "To: %s" % to_addr,
10        "Subject: %s" % subject ,
11        "",
12        body_text
13    ))
14    server = smtplib.SMTP(host)
15    server.sendmail(from_addr, [to_addr], BODY)
16    server.quit()
```

```

17
18 if __name__ == "__main__":
19     host = "mySMTP.server.com"
20     subject = "Test email from Python"
21     to_addr = "mike@someAddress.org"
22     from_addr = "python@mydomain.com"
23     body_text = "Python rules them all!"
24     send_email(host, subject, to_addr, from_addr, body_text)

```

Now you can see how small the actual code is by just looking at the function itself. That's 13 lines! And we could make it shorter if we didn't put every item in the BODY on its own line, but it wouldn't be as readable. Now we'll add a config file to hold the server information and the **from** address. Why? Well in the work I do, we might use different email servers to send email or if the email server gets upgraded and the name changes, then we only need to change the config file rather than the code. The same thing could apply to the **from** address if our company was bought and merged into another.

Let's take a look at the config file (save it as **email.ini**):

```

1 [smtp]
2 server = some.server.com
3 from_addr = python@mydomain.com

```

That is a very simple config file. In it we have a section labelled **smtp** in which we have two items: **server** and **from_addr**. We'll use **configObj** to read this file and turn it into a Python dictionary. Here's the updated version of the code (save it as **smtp_config.py**):

```

1 import os
2 import smtplib
3 import sys
4
5 from configparser import ConfigParser
6
7 def send_email(subject, to_addr, body_text):
8     """
9         Send an email
10    """
11    base_path = os.path.dirname(os.path.abspath(__file__))
12    config_path = os.path.join(base_path, "email.ini")
13
14    if os.path.exists(config_path):
15        cfg = ConfigParser()

```

```
16     cfg.read(config_path)
17 else:
18     print("Config not found! Exiting!")
19     sys.exit(1)
20
21 host = cfg.get("smtp", "server")
22 from_addr = cfg.get("smtp", "from_addr")
23
24 BODY = "\r\n".join((
25     "From: %s" % from_addr,
26     "To: %s" % to_addr,
27     "Subject: %s" % subject ,
28     "",
29     body_text
30 ))
31 server = smtplib.SMTP(host)
32 server.sendmail(from_addr, [to_addr], BODY)
33 server.quit()
34
35 if __name__ == "__main__":
36     subject = "Test email from Python"
37     to_addr = "mike@someAddress.org"
38     body_text = "Python rules them all!"
39     send_email(subject, to_addr, body_text)
```

We've added a little check to this code. We want to first grab the path that the script itself is in, which is what base_path represents. Next we combine that path with the file name to get a fully qualified path to the config file. We then check for the existence of that file. If it's there, we create a **ConfigParser** and if it's not, we print a message and exit the script. We should add an exception handler around the **ConfigParser.read()** call just to be on the safe side though as the file could exist, but be corrupt or we might not have permission to open it and that will throw an exception. That will be a little project that you can attempt on your own. Anyway, let's say that everything goes well and the **ConfigParser** object is created successfully. Now we can extract the host and from_addr information using the usual **ConfigParser** syntax.

Now we're ready to learn how to send multiple emails at the same time!

Sending Multiple Emails at Once

Let's modify our last example a little so we send multiple emails!

```
1 import os
2 import smtplib
3 import sys
4
5 from configparser import ConfigParser
6
7 def send_email(subject, body_text, emails):
8     """
9     Send an email
10    """
11    base_path = os.path.dirname(os.path.abspath(__file__))
12    config_path = os.path.join(base_path, "email.ini")
13
14    if os.path.exists(config_path):
15        cfg = ConfigParser()
16        cfg.read(config_path)
17    else:
18        print("Config not found! Exiting!")
19        sys.exit(1)
20
21    host = cfg.get("smtp", "server")
22    from_addr = cfg.get("smtp", "from_addr")
23
24    BODY = "\r\n".join((
25        "From: %s" % from_addr,
26        "To: %s" % ', '.join(emails),
27        "Subject: %s" % subject,
28        "",
29        body_text
30    ))
31    server = smtplib.SMTP(host)
32    server.sendmail(from_addr, emails, BODY)
33    server.quit()
34
35 if __name__ == "__main__":
36     emails = ["mike@someAddress.org", "someone@gmail.com"]
37     subject = "Test email from Python"
38     body_text = "Python rules them all!"
39     send_email(subject, body_text, emails)
```

You'll notice that in this example, we removed the `to_addr` parameter and added an `emails` parameter, which is a list of email addresses. To make this work, we need to create a comma-separated string

in the **To:** portion of the BODY and also pass the email list to the **sendmail** method. Thus we do the following to create a simple comma separated string: ', '.join(emails). Simple, huh?

Send email using the TO, CC and BCC lines

Now we just need to figure out how to send using the CC and BCC fields. Let's create a new version of this code that supports that functionality!

```
1 import os
2 import smtplib
3 import sys
4
5 from configparser import ConfigParser
6
7 def send_email(subject, body_text, to_emails, cc_emails, bcc_emails):
8     """
9         Send an email
10    """
11    base_path = os.path.dirname(os.path.abspath(__file__))
12    config_path = os.path.join(base_path, "email.ini")
13
14    if os.path.exists(config_path):
15        cfg = ConfigParser()
16        cfg.read(config_path)
17    else:
18        print("Config not found! Exiting!")
19        sys.exit(1)
20
21    host = cfg.get("smtp", "server")
22    from_addr = cfg.get("smtp", "from_addr")
23
24    BODY = "\r\n".join((
25        "From: %s" % from_addr,
26        "To: %s" % ', '.join(to_emails),
27        "CC: %s" % ', '.join(cc_emails),
28        "BCC: %s" % ', '.join(bcc_emails),
29        "Subject: %s" % subject,
30        "",
31        body_text
32    ))
33    emails = to_emails + cc_emails + bcc_emails
```

```
34
35     server = smtplib.SMTP(host)
36     server.sendmail(from_addr, emails, BODY)
37     server.quit()
38
39 if __name__ == "__main__":
40     emails = ["mike@somewhere.org"]
41     cc_emails = ["someone@gmail.com"]
42     bcc_emails = ["schmuck@newtel.net"]
43
44     subject = "Test email from Python"
45     body_text = "Python rules them all!"
46     send_email(subject, body_text, emails, cc_emails, bcc_emails)
```

In this code, we pass in 3 lists, each with one email address a piece. We create the CC and BCC fields exactly the same as before, but we also need to combine the 3 lists into one so we can pass the combined list to the `sendmail()` method. There is some talk on forums like StackOverflow that some email clients may handle the BCC field in odd ways that allow the recipient to see the BCC list via the email headers. I am unable to confirm this behavior, but I do know that Gmail successfully strips the BCC information from the email header.

Now we're ready to move on to using Python's email module!

Add an attachment / body using the email module

Now we'll take what we learned from the previous section and mix it together with the Python email module so that we can send attachments. The email module makes adding attachments extremely easy. Here's the code:

```
1 import os
2 import smtplib
3 import sys
4
5 from configparser import ConfigParser
6 from email import encoders
7 from email.mime.text import MIMEText
8 from email.mime.base import MIMEBase
9 from email.mime.multipart import MIMEMultipart
10 from email.utils import formatdate
11
12 #-----
13 def send_email_with_attachment(subject, body_text, to_emails,
```

```
14                               cc_emails, bcc_emails, file_to_attach):
15
16     """  
17     Send an email with an attachment  
18     """  
19  
20     base_path = os.path.dirname(os.path.abspath(__file__))  
21     config_path = os.path.join(base_path, "email.ini")  
22     header = 'Content-Disposition', 'attachment; filename=%s' % file_to_attach  
23  
24     # get the config  
25     if os.path.exists(config_path):  
26         cfg = ConfigParser()  
27         cfg.read(config_path)  
28     else:  
29         print("Config not found! Exiting!")  
30         sys.exit(1)  
31  
32     # extract server and from_addr from config  
33     host = cfg.get("smtp", "server")  
34     from_addr = cfg.get("smtp", "from_addr")  
35  
36     # create the message  
37     msg = MIMEMultipart()  
38     msg["From"] = from_addr  
39     msg["Subject"] = subject  
40     msg["Date"] = formatdate(localtime=True)  
41     if body_text:  
42         msg.attach( MIMEText(body_text) )  
43  
44     msg["To"] = ', '.join(to_emails)  
45     msg["cc"] = ', '.join(cc_emails)  
46  
47     attachment = MIMEBase('application', "octet-stream")  
48     try:  
49         with open(file_to_attach, "rb") as fh:  
50             data = fh.read()  
51             attachment.set_payload( data )  
52             encoders.encode_base64(attachment)  
53             attachment.add_header(*header)  
54             msg.attach(attachment)  
55     except IOError:  
56         msg = "Error opening attachment file %s" % file_to_attach  
57         print(msg)
```

```
56     sys.exit(1)
57
58     emails = to_emails + cc_emails
59
60     server = smtplib.SMTP(host)
61     server.sendmail(from_addr, emails, msg.as_string())
62     server.quit()
63
64 if __name__ == "__main__":
65     emails = ["mike@someAddress.org", "nedry@jp.net"]
66     cc_emails = ["someone@gmail.com"]
67     bcc_emails = ["anonymous@circe.org"]
68
69     subject = "Test email with attachment from Python"
70     body_text = "This email contains an attachment!"
71     path = "/path/to/some/file"
72     send_email_with_attachment(subject, body_text, emails,
73                                 cc_emails, bcc_emails, path)
```

Here we have renamed our function and added a new argument, `file_to_attach`. We also need to add a header and create a `MIMEMultipart` object. The header could be created any time before we add the attachment. We add elements to the `MIMEMultipart` object (`msg`) like we would keys to a dictionary. You'll note that we have to use the `email` module's `formatdate` method to insert the properly formatted date. To add the body of the message, we need to create an instance of `MIMEText`. If you're paying attention, you'll see that we didn't add the BCC information, but you could easily do so by following the conventions in the code above. Next we add the attachment. We wrap it in an exception handler and use the `with` statement to extract the file and place it in our `MIMEBase` object. Finally we add it to the `msg` variable and we send it out. Notice that we have to convert the `msg` to a string in the `sendmail` method.

Wrapping Up

Now you know how to send out emails with Python. For those of you that like mini projects, you should go back and add additional error handling around the `server.sendmail` portion of the code in case something odd happens during the process, such as an `SMTPAuthenticationError` or `SMTPConnectError`. We could also beef up the error handling during the attachment of the file to catch other errors. Finally, we may want to take those various lists of emails and create one normalized list that has removed duplicates. This is especially important if we are reading a list of email addresses from a file.

Also note that our from address is fake. We can spoof emails using Python and other programming languages, but that is very bad etiquette and possibly illegal depending on where you live. You have been warned! Use your knowledge wisely and enjoy Python for fun and profit!

Chapter 18 - The sqlite Module

SQLite is a self-contained, server-less, config-free transactional SQL database engine. Python gained the `sqlite3` module all the way back in version 2.5 which means that you can create SQLite database with any current Python without downloading any additional dependencies. Mozilla uses SQLite databases for its popular Firefox browser to store bookmarks and other various pieces of information. In this chapter you will learn the following:

- How to create a SQLite database
- How to insert data into a table
- How to edit the data
- How to delete the data
- Basic SQL queries

In other words, rather than covering bits and pieces of the `sqlite3` module, we'll go through how to actually use it.

If you want to inspect your database visually, you can use the SQLite Manager plugin for Firefox (just Google for it) or if you like the command line, you can use SQLite's command line shell.

How to Create a Database and INSERT Some Data

Creating a database in SQLite is really easy, but the process requires that you know a little SQL to do it. Here's some code that will create a database to hold music albums:

```
1 import sqlite3
2
3 conn = sqlite3.connect("mydatabase.db") # or use :memory: to put it in RAM
4
5 cursor = conn.cursor()
6
7 # create a table
8 cursor.execute("""CREATE TABLE albums
9             (title text, artist text, release_date text,
10              publisher text, media_type text)
11             """)
```

First we have to import the `sqlite3` module and create a connection to the database. You can pass it a file path, file name or just use the special string “`:memory:`” to create the database in memory. In our case, we created it on disk in a file called `mydatabase.db`. Next we create a cursor object, which allows you to interact with the database and add records, among other things. Here we use SQL syntax to create a table named `albums` with 5 text fields: title, artist, release_date, publisher and media_type. SQLite only supports five **data types**: null, integer, real, text and blob. Let’s build on this code and insert some data into our new table!

Note: If you run the `CREATE TABLE` command and the database already exists, you will receive an error message.

```
1 # insert some data
2 cursor.execute("""INSERT INTO albums
3             VALUES ('Glow', 'Andy Hunter', '7/24/2012',
4                     'Xplore Records', 'MP3')"""
5         )
6
7 # save data to database
8 conn.commit()
9
10 # insert multiple records using the more secure "?" method
11 albums = [('Exodus', 'Andy Hunter', '7/9/2002', 'Sparrow Records', 'CD'),
12           ('Until We Have Faces', 'Red', '2/1/2011', 'Essential Records', 'CD'),
13           ('The End is Where We Begin', 'Thousand Foot Krutch', '4/17/2012', 'TF\
14 Kmusic', 'CD'),
15           ('The Good Life', 'Trip Lee', '4/10/2012', 'Reach Records', 'CD')]
16 cursor.executemany("INSERT INTO albums VALUES (?,?,?,?,?)", albums)
17 conn.commit()
```

Here we use the `INSERT INTO` SQL command to insert a record into our database. Note that each item had to have single quotes around it. This can get complicated when you need to insert strings that include single quotes in them. Anyway, to save the record to the database, we have to **commit** it. The next piece of code shows how to add multiple records at once by using the cursor’s `executemany` method. Note that we’re using question marks (?) instead of string substitution (%) to insert the values. Using string substitution is NOT safe and should not be used as it can allow SQL injection attacks to occur. The question mark method is much better and using SQLAlchemy is even better because it does all the escaping for you so you won’t have to mess with the annoyances of converting embedded single quotes into something that SQLite will accept.

Updating and Deleting Records

Being able to update your database records is key to keeping your data accurate. If you can’t update, then your data will become out of date and pretty useless very quickly. Sometimes you will need to

delete rows from your data too. We'll be covering both of those topics in this section. First, let's do an update!

```
1 import sqlite3
2
3 conn = sqlite3.connect("mydatabase.db")
4 cursor = conn.cursor()
5
6 sql = """
7 UPDATE albums
8 SET artist = 'John Doe'
9 WHERE artist = 'Andy Hunter'
10 """
11 cursor.execute(sql)
12 conn.commit()
```

Here we use SQL's **UPDATE** command to update our albums table. You can use **SET** to change a field, so in this case we change the artist field to be *John Doe* in any record **WHERE** the artist field is set to *Andy Hunter*. Wasn't that easy? Note that if you don't commit the changes, then your changes won't be written out to the database. The **DELETE** command is almost as easy. Let's check that out!

```
1 import sqlite3
2
3 conn = sqlite3.connect("mydatabase.db")
4 cursor = conn.cursor()
5
6 sql = """
7 DELETE FROM albums
8 WHERE artist = 'John Doe'
9 """
10 cursor.execute(sql)
11 conn.commit()
```

Deleting is even easier than updating. The SQL is only 2 lines! In this case, all we had to do was tell SQLite which table to delete from (albums) and which records to delete using the WHERE clause. Thus it looked for any records that had "John Doe" in its artist field and deleted it.

Basic SQLite Queries

Queries in SQLite are pretty much the same as what you'd use for other databases, such as MySQL or Postgres. You just use normal SQL syntax to run the queries and then have the cursor object execute the SQL. Here are a few examples:

```
1 import sqlite3
2
3 conn = sqlite3.connect("mydatabase.db")
4 #conn.row_factory = sqlite3.Row
5 cursor = conn.cursor()
6
7 sql = "SELECT * FROM albums WHERE artist=?"
8 cursor.execute(sql, [("Red")])
9 print(cursor.fetchall()) # or use fetchone()
10
11 print("\nHere's a listing of all the records in the table:\n")
12 for row in cursor.execute("SELECT rowid, * FROM albums ORDER BY artist"):
13     print(row)
14
15 print("\nResults from a LIKE query:\n")
16 sql = """
17     SELECT * FROM albums
18     WHERE title LIKE 'The%"""
19 cursor.execute(sql)
20 print(cursor.fetchall())
```

The first query we execute is a `SELECT *` which means that we want to select all the records that match the artist name we pass in, which in this case is “Red”. Next we execute the SQL and use `fetchall()` to return all the results. You can also use `fetchone()` to grab the first result. You’ll also notice that there’s a commented out section related to a mysterious `row_factory`. If you un-comment that line, the results will be returned as Row objects that are kind of like Python dictionaries and give you access to the row’s fields just like a dictionary. However, you cannot do item assignment with a Row object.

The second query is much like the first, but it returns every record in the database and orders the results by the artist name in ascending order. This also demonstrates how we can loop over the results. The last query shows how to use SQL’s LIKE command to search for partial phrases. In this case, we do a search of the entire table for titles that start with “The”. The percent sign (%) is a wildcard operator.

Wrapping Up

Now you know how to use Python to create a SQLite database. You can also create, update and delete records as well as run queries against your database.

Chapter 19 - The subprocess Module

The **subprocess** module gives the developer the ability to start processes or programs from Python. In other words, you can start applications and pass arguments to them using the subprocess module. The subprocess module was added way back in Python 2.4 to replace the **os** modules set of `os.popen`, `os.spawn` and `os.system` calls as well as replace `popen2` and the old **commands** module. We will be looking at the following aspects of the subprocess module:

- the `call` function
- the `Popen` class
- how to communicate with a spawned process

Let's get started!

The call function

The subprocess module provides a function named `call`. This function allows you to call another program, wait for the command to complete and then return the return code. It accepts one or more arguments as well as the following keyword arguments (with their defaults): `stdin=None`, `stdout=None`, `stderr=None`, `shell=False`.

Let's look at a simple example:

```
1 >>> import subprocess
2 >>> subprocess.call("notepad.exe")
3 0
```

If you run this on a Windows machine, you should see Notepad open up. You will notice that IDLE waits for you to close Notepad and then it returns a code zero (0). This means that it completed successfully. If you receive anything except for a zero, then it usually means you have had some kind of error.

Normally when you call this function, you would want to assign the resulting return code to a variable so you can check to see if it was the result you expected. Let's do that:

```
1 >>> code = subprocess.call("notepad.exe")
2 >>> if code == 0:
3     print("Success!")
4 else:
5     print("Error!")
6 Success!
```

If you run this code, you will see that it prints out **Success!** to the screen. The `call` method also accepts arguments to be passed to the program that you're executing. Let's see how that works:

```
1 >>> code = subprocess.call(["ping", "www.yahoo.com"])
2
3 Pinging ds-any-fp3-real.wa1.b.yahoo.com [98.139.180.149] with 32 bytes of data:
4 Reply from 98.139.180.149: bytes=32 time=66ms TTL=45
5 Reply from 98.139.180.149: bytes=32 time=81ms TTL=45
6 Reply from 98.139.180.149: bytes=32 time=81ms TTL=45
7 Reply from 98.139.180.149: bytes=32 time=69ms TTL=45
8
9 Ping statistics for 98.139.180.149:
10    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
11    Approximate round trip times in milli-seconds:
12        Minimum = 66ms, Maximum = 81ms, Average = 74ms
13 >>> code
14 0
```

You will notice that in this example we are passing a list of arguments. The first item in the list is the program we want to call. Anything else in the list are arguments that we want to pass to that program. So in this example, we are executing a ping against Yahoo's website. You will notice that the return code was zero, so everything completed successfully.

You can also execute the program using the operating system's `shell`. This does add a level of abstraction to the process and it raises the possibility of security issues. Here is the Python documentation's official warning on the matter:

Executing shell commands that incorporate unsanitized input from an untrusted source makes a program vulnerable to shell injection, a serious security flaw which can result in arbitrary command execution. For this reason, the use of shell=True is strongly discouraged in cases where the command string is constructed from external input.

The usual recommendation is not to use it if an outside process or person can modify the call's arguments. If you're hard-coding something yourself, then it doesn't matter as much.

The Popen Class

The **Popen** class executes a child program in a new process. Unlike the **call** method, it does not wait for the called process to end unless you tell it to using by using the **wait** method. Let's try running Notepad through Popen and see if we can detect a difference:

```
1 >>> program = "notepad.exe"
2 >>> subprocess.Popen(program)
3 <subprocess.Popen object at 0x01EE0430>
```

Here we create a variable called **program** and assign it the value of “notepad.exe”. Then we pass it to the Popen class. When you run this, you will see that it immediately returns the **subprocess.Popen object** and the application that was called executes. Let's make Popen wait for the program to finish:

```
1 >>> program = "notepad.exe"
2 >>> process = subprocess.Popen(program)
3 >>> code = process.wait()
4 >>> print(code)
5 0
```

When you do this in IDLE, Notepad will pop up and may be in front of your IDLE session. Just move Notepad out of the way, but do not close it! You need to tell your process to **wait** for you won't be able to get the return code. Once you've typed that line, close Notepad and print the code out. Or you could just put all this code into a saved Python file and run that.

Note that using the **wait** method can cause the child process to deadlock when using the **stdout/stderr=PIPE** commands when the process generates enough output to block the pipe. You can use the **communicate** method to alleviate this situation. We'll be looking at that method in the next section.

Now let's try running Popen using multiple arguments:

```
1 >>> subprocess.Popen(["ls", "-l"])
2 <subprocess.Popen object at 0xb7451001>
```

If you run this code in Linux, you'll see it print out the Popen object message and then a listing of the permissions and contents of whatever folder you ran this in. You can use the **shell** argument with Popen too, but the same caveats apply to Popen as they did to the call method.

Learning to Communicate

There are several ways to communicate with the process you have invoked. We're just going to focus on how to use the subprocess module's **communicate** method. Let's take a look:

```
1 args = ["ping", "www.yahoo.com"]
2 process = subprocess.Popen(args,
3                             stdout=subprocess.PIPE)
4
5 data = process.communicate()
6 print(data)
```

In this code example, we create an `args` variable to hold our list of arguments. Then we redirect standard out (`stdout`) to our subprocess so we can communicate with it. The `communicate` method itself allows us to communicate with the process we just spawned. We can actually pass input to the process using this method. But in this example, we just use `communicate` to read from standard out. You will notice when you run this code that `communicate` will wait for the process to finish and then returns a two-element tuple that contains what was in `stdout` and `stderr`. Here is the result from my run:

```
1 ('Pinging ds-any-fp3-real.wa1.b.yahoo.com [98.139.180.149] with 32 bytes of data:
2 Reply from 98.139.180.149: bytes=32 time=139ms TTL=45
3 Reply from 98.139.180.149: bytes=32 time=162ms TTL=45
4 Reply from 98.139.180.149: bytes=32 time=164ms TTL=45
5 Reply from 98.139.180.149: bytes=32 time=110ms TTL=45
6 Ping statistics for 98.139.180.149:
7 Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
8 Approximate round trip times in milli-seconds:
9 Minimum = 110ms, Maximum = 164ms, Average = 143ms
10 ', None)
```

That's kind of ugly. Let's make it print out the result in a more readable format, shall we?

```
1 import subprocess
2
3 args = ["ping", "www.yahoo.com"]
4 process = subprocess.Popen(args, stdout=subprocess.PIPE)
5
6 data = process.communicate()
7 for line in data:
8     print(line)
```

If you run this code, you should see something like the following printed to your screen:

```
1 Pinging ds-any-fp3-real.wa1.b.yahoo.com [98.139.180.149] with 32 bytes of data:  
2 Reply from 98.139.180.149: bytes=32 time=67ms TTL=45  
3 Reply from 98.139.180.149: bytes=32 time=68ms TTL=45  
4 Reply from 98.139.180.149: bytes=32 time=70ms TTL=45  
5 Reply from 98.139.180.149: bytes=32 time=69ms TTL=45  
6  
7 Ping statistics for 98.139.180.149:  
8     Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),  
9 Approximate round trip times in milli-seconds:  
10        Minimum = 67ms, Maximum = 70ms, Average = 68ms  
11  
12 None
```

That last line that says “None” is the result of stderr, which means that there were no errors.

Wrapping Up

At this point, you have the knowledge to use the subprocess module effectively. You can open a process in two different ways, you know how to wait for a return code, and you know how to communicate with the child process that you created.

In the next chapter, we will be looking at the `sys` module.

Chapter 20 - The sys Module

The `sys` module provides system specific parameters and functions. We will be narrowing our study down to the following:

- `sys.argv`
- `sys.executable`
- `sys.exit`
- `sys.modules`
- `sys.path`
- `sys.platform`
- `sys.stdin/stdout/stderr`

|

`sys.argv`

The value of `sys.argv` is a Python list of command line arguments that were passed to the Python script. The first argument, `argv[0]` is the name of the Python script itself. Depending on the platform that you are running on, the first argument may contain the full path to the script or just the file name. You should study the documentation for additional details.

Let's try out a few examples to familiarize ourselves with this little tool:

```
1 >>> import sys
2 >>> sys.argv
3 [ '' ]
```

If you run this in the interpreter, you will receive a list with an empty string. Let's create a file named "sysargv.py" with the following contents:

```
1 # sysargv.py
2 import sys
3
4 print(sys.argv)
```

Now run the code in IDLE. You should see it print out a list with a single element that contains the path to your script in it. Let's try passing the script some arguments. Open up a terminal / console screen and change directories (use the "cd" command) to where your script is. Then run something like this:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mike>cd c:\py101

c:\py101>python sysargv.py -v "somefile.py"
['sysargv.py', '-v', 'somefile.py']

c:\py101>
```

image

You will notice that it outputs the following to the screen:

```
1 ['sysargv.py', '-v', 'somefile.py']
```

The first argument is the name of the script we wrote. The next two arguments in the list are the ones we passed to our script on the command line.

sys.executable

The value of **sys.executable** is the absolute path to the Python interpreter. This is useful when you are using someone else's machine and need to know where Python is installed. On some systems, this command will fail and it will return an empty string or None. Here's how to use it:

```
1 >>> import sys
2 >>> sys.executable
3 'C:\\Python27\\pythonw.exe'
```

sys.exit

The **sys.exit()** function allows the developer to exit from Python. The **exit** function takes an optional argument, typically an integer, that gives an exit status. Zero is considered a "successful termination". Be sure to check if your operating system has any special meanings for its exit statuses so that you can follow them in your own application. Note that when you call **exit**, it will raise the **SystemExit** exception, which allows cleanup functions to work in the **finally** clauses of **try / except** blocks.

Let's take a look at how to call this:

```
1 >>> import sys
2 >>> sys.exit(0)
3
4 Traceback (most recent call last):
5   File "<pyshell#5>", line 1, in <module>
6     sys.exit(0)
7 SystemExit: 0
```

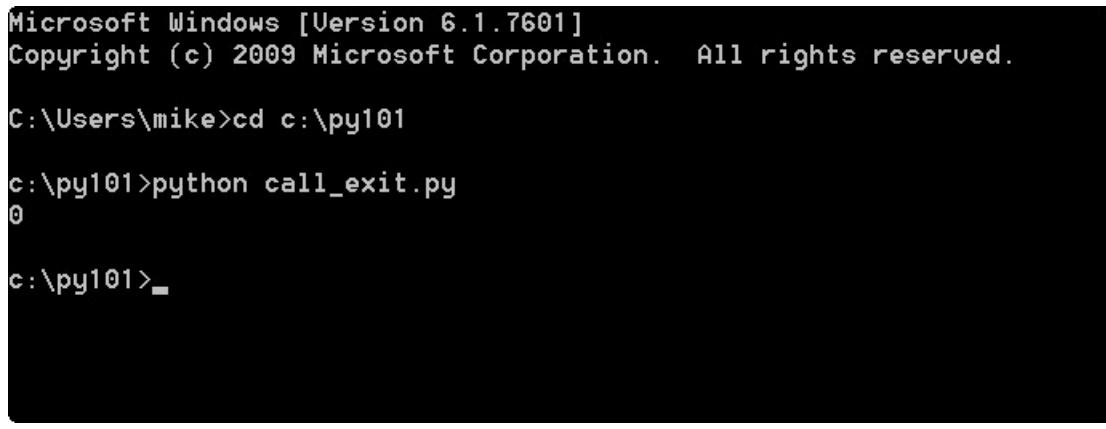
When you run this code in IDLE, you will see the `SystemExit` error raised. Let's create a couple of scripts to test this out. First you'll want to create a master script, a program that will call another Python script. Let's name it "call_exit.py". Put the following code into it:

```
1 # call_exit.py
2 import subprocess
3
4 code = subprocess.call(["python.exe", "exit.py"])
5 print(code)
```

Now create another Python script called "exit.py" and save it in the same folder. Put the following code into it:

```
1 import sys
2
3 sys.exit(0)
```

Now let's try running this code:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mike>cd c:\py101

c:\py101>python call_exit.py
0

c:\py101>
```

image

In the screenshot above, you can see that the exit script we wrote returned a zero, so it ran successfully. You have also learned how to call another Python script from within Python!

sys.path

The sys module's **path** value is a list of strings that specifies the search path for modules. Basically this tells Python what locations to look in when it tries to import a module. According to the Python documentation, **sys.path** is initialized from an environment variable called PYTHONPATH, plus an installation-dependent default. Let's give it a try:

```
1 >>> import sys
2 >>> print(sys.path)
3 [
4 'C:\\\\Python27\\\\Lib\\\\idlelib',
5 'C:\\\\Python27\\\\lib\\\\site-packages\\\\setuptools-0.9.5-py2.7.egg',
6 'C:\\\\Python27\\\\lib\\\\site-packages\\\\pip-1.3.1-py2.7.egg',
7 'C:\\\\Python27\\\\lib\\\\site-packages\\\\sphinx-1.2b3-py2.7.egg',
8 'C:\\\\Python27\\\\lib\\\\site-packages\\\\docutils-0.11-py2.7.egg',
9 'C:\\\\Python27\\\\lib\\\\site-packages\\\\pygments-1.6-py2.7.egg',
10 'C:\\\\Windows\\\\system32\\\\python27.zip',
11 C:\\\\Python27\\\\DLLs',
12 'C:\\\\Python27\\\\lib',
13 'C:\\\\Python27\\\\lib\\\\plat-win',
14 'C:\\\\Python27\\\\lib\\\\lib-tk',
15 'C:\\\\Python27',
16 'C:\\\\Python27\\\\lib\\\\site-packages',
17 'C:\\\\Python27\\\\lib\\\\site-packages\\\\PIL',
18 'C:\\\\Python27\\\\lib\\\\site-packages\\\\wx-2.9.4-msw']
```

This can be very useful for debugging why a module isn't getting imported. You can also modify the path. Because it's a list, we can add or delete paths from it. Here's how to add a path:

```
1 >>> sys.path.append("/path/to/my/module")
```

I'll leave deleting a path as an exercise for the reader.

sys.platform

The **sys.platform** value is a platform identifier. You can use this to append platform specific modules to **sys.path**, import different modules depending on platform or run different pieces of code. Let's take a look:

```
1 >>> import sys  
2 >>> sys.platform  
3 'win32'
```

This tells us that Python is running on a Windows machine. Here's an example of how we might use this information:

```
1 >>> os = sys.platform  
2 >>> if os == "win32":  
3     # use Window-related code here  
4     import _winreg  
5 elif os.startswith('linux'):  
6     # do something Linux specific  
7     import subprocess  
8     subprocess.Popen(["ls", "-l"])
```

The code above shows how we might check to see if we're using a particular operating system. If we're on Windows, we'll get some information from the Window's Registry using a Python module called `_winreg`. If we're on Linux, we might execute the `ls` command to get information about the directory we're in.

sys.stdin / stdout / stderr

The `stdin`, `stdout` and `stderr` map to file objects that correspond to the interpreter's standard input, output and error streams, respectively. `stdin` is used for all input given to the interpreter except for scripts whereas `stdout` is used for the output of `print` and `expression` statements. The primary reason I mention this is that you will sometimes need to redirect `stdout` or `stderr` or both to a file, such as a log or to some kind of display in a custom GUI you have created. You could also redirect `stdin`, but I have rarely seen this done.

Wrapping Up

There are many other values and methods in the `sys` module. Be sure to look it up in the Python documentation, section 27.1. You have learned a lot in this chapter. You now know how to exit a Python program, how to get platform information, working with arguments passed on the command line and much more. In the next chapter, we'll be learning about Python threads!

Chapter 21 - The threading module

Python has a number of different concurrency constructs such as threading, queues and multiprocessing. The threading module used to be the primary way of accomplishing concurrency. A few years ago, the multiprocessing module was added to the Python suite of standard libraries. This chapter will be focused on how to use threads and queues.

Using Threads

We will start with a simple example that just demonstrates how threads work. We will sub-class the `Thread` class and make it print out its name to stdout. Let's get coding!

```
1 import random
2 import time
3
4 from threading import Thread
5
6 class MyThread(Thread):
7     """
8     A threading example
9     """
10
11     def __init__(self, name):
12         """Initialize the thread"""
13         Thread.__init__(self)
14         self.name = name
15
16     def run(self):
17         """Run the thread"""
18         amount = random.randint(3, 15)
19         time.sleep(amount)
20         msg = "%s is running" % self.name
21         print(msg)
22
23 def create_threads():
24     """
25     Create a group of threads
26     """
```

```
27     for i in range(5):
28         name = "Thread #%" % (i+1)
29         my_thread = MyThread(name)
30         my_thread.start()
31
32 if __name__ == "__main__":
33     create_threads()
```

In the code above, we import Python's `random` module, the `time` module and we import the `Thread` class from the `threading` module. Next we sub-class `Thread` and make override its `__init__` method to accept an argument we label "name". To start a thread, you have to call its `start()` method. When you start a thread, it will automatically call the thread's `run` method. We have overridden the thread's `run` method to make it choose a random amount of time to sleep. The `random.randint` example here will cause Python to randomly choose a number from 3-15. Then we make the thread sleep the number of seconds that we just randomly chose to simulate it actually doing something. Finally we print out the name of the thread to let the user know that the thread has finished.

The `create_threads` function will create 5 threads, giving each of them a unique name. If you run this code, you should see something like this:

```
1 Thread #2 is running
2 Thread #3 is running
3 Thread #1 is running
4 Thread #4 is running
5 Thread #5 is running
```

The order of the output will be different each time. Try running the code a few times to see the order change. Now let's write something a little more practical!

Writing a Threaded Downloader

The previous example wasn't very useful other than as a tool to explain how threads work. So in this example, we will create a `Thread` class that can download files from the internet. The U.S. Internal Revenue Service has lots of PDF forms that it has its citizens use for taxes. We will use this free resource for our demo. Here's the code:

```
1 # Python 2 version
2
3 import os
4 import urllib2
5
6 from threading import Thread
7
8 class DownloadThread(Thread):
9     """
10     A threading example that can download a file
11     """
12
13     def __init__(self, url, name):
14         """Initialize the thread"""
15         Thread.__init__(self)
16         self.name = name
17         self.url = url
18
19     def run(self):
20         """Run the thread"""
21         handle = urllib2.urlopen(self.url)
22         fname = os.path.basename(self.url)
23         with open(fname, "wb") as f_handler:
24             while True:
25                 chunk = handle.read(1024)
26                 if not chunk:
27                     break
28                 f_handler.write(chunk)
29         msg = "%s has finished downloading %s!" % (self.name,
30                                                       self.url)
31         print(msg)
32
33 def main(urls):
34     """
35     Run the program
36     """
37     for item, url in enumerate(urls):
38         name = "Thread %s" % (item+1)
39         thread = DownloadThread(url, name)
40         thread.start()
41
42 if __name__ == "__main__":
```

```
43     urls = ["http://www.irs.gov/pub/irs-pdf/f1040.pdf",
44               "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
45               "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
46               "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
47               "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
48 main(urls)
```

This is basically a complete rewrite of the first script. In this one we import the `os` and `urllib2` modules as well as the `threading` module. We will be using `urllib2` to do the actual downloading inside the `thread` class. The `os` module is used to extract the name of the file we're downloading so we can use it to create a file with the same name on our machine. In the `DownloadThread` class, we set up the `__init__` to accept a url and a name for the thread. In the `run` method, we open up the url, extract the filename and then use that filename for naming / creating the file on disk. Then we use a `while` loop to download the file a kilobyte at a time and write it to disk. Once the file is finished saving, we print out the name of the thread and which url has finished downloading.

The Python 3 version of the code is slightly different. You have to import `urllib` instead of `urllib2` and use `urllib.request.urlopen` instead of `urllib2.urlopen`. Here's the code so you can see the difference:

```
1 # Python 3 version
2
3 import os
4 import urllib.request
5
6 from threading import Thread
7
8 class DownloadThread(Thread):
9     """
10     A threading example that can download a file
11     """
12
13     def __init__(self, url, name):
14         """Initialize the thread"""
15         Thread.__init__(self)
16         self.name = name
17         self.url = url
18
19     def run(self):
20         """Run the thread"""
21         handle = urllib.request.urlopen(self.url)
22         fname = os.path.basename(self.url)
23         with open(fname, "wb") as f_handler:
24             while True:
```

```
25             chunk = handle.read(1024)
26         if not chunk:
27             break
28         f_handler.write(chunk)
29     msg = "%s has finished downloading %s!" % (self.name,
30                                                 self.url)
31     print(msg)
32
33 def main(urls):
34     """
35     Run the program
36     """
37     for item, url in enumerate(urls):
38         name = "Thread %s" % (item+1)
39         thread = DownloadThread(url, name)
40         thread.start()
41
42 if __name__ == "__main__":
43     urls = [
44         "http://www.irs.gov/pub/irs-pdf/f1040.pdf",
45         "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
46         "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
47         "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
48         "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
49     main(urls)
```

Using Queues

A Queue can be used for first-in-first-out (FIFO) or last-in-last-out (LILO) stack-like implementations if you just use them directly. In this section, we're going to mix threads in and create a simple file downloader script to demonstrate how Queues work for cases where we want concurrency.

To help explain how Queues work, we will rewrite the downloading script from the previous section to use Queues. Let's get started!

```
1 import os
2 import threading
3 import urllib.request
4
5 from queue import Queue
6
7 class Downloader(threading.Thread):
8     """Threaded File Downloader"""
9
10    def __init__(self, queue):
11        """Initialize the thread"""
12        threading.Thread.__init__(self)
13        self.queue = queue
14
15    def run(self):
16        """Run the thread"""
17        while True:
18            # gets the url from the queue
19            url = self.queue.get()
20
21            # download the file
22            self.download_file(url)
23
24            # send a signal to the queue that the job is done
25            self.queue.task_done()
26
27    def download_file(self, url):
28        """Download the file"""
29        handle = urllib.request.urlopen(url)
30        fname = os.path.basename(url)
31        with open(fname, "wb") as f:
32            while True:
33                chunk = handle.read(1024)
34                if not chunk: break
35                f.write(chunk)
36
37    def main(urls):
38        """
39        Run the program
40        """
41        queue = Queue()
42
```

```

43     # create a thread pool and give them a queue
44     for i in range(5):
45         t = Downloader(queue)
46         t.setDaemon(True)
47         t.start()
48
49     # give the queue some data
50     for url in urls:
51         queue.put(url)
52
53     # wait for the queue to finish
54     queue.join()
55
56 if __name__ == "__main__":
57     urls = [
58         "http://www.irs.gov/pub/irs-pdf/f1040.pdf",
59         "http://www.irs.gov/pub/irs-pdf/f1040a.pdf",
60         "http://www.irs.gov/pub/irs-pdf/f1040ez.pdf",
61         "http://www.irs.gov/pub/irs-pdf/f1040es.pdf",
62         "http://www.irs.gov/pub/irs-pdf/f1040sb.pdf"]
63     main(urls)

```

Let's break this down a bit. First of all, we need to look at the main function definition to see how this all flows. Here we see that it accepts a list of urls. The main function then creates a queue instance that it passes to 5 daemonized threads. The main difference between daemonized and non-daemon threads is that you have to keep track of non-daemon threads and close them yourself whereas with a daemon thread you basically just set them and forget them and when your app closes, they close too. Next we load up the queue (using its put method) with the urls we passed in.

Finally we tell the queue to wait for the threads to do their processing via the join method. In the download class, we have the line `self.queue.get()` which blocks until the queue has something to return. That means the threads just sit idly waiting to pick something up. It also means that for a thread to get something from the queue, it must call the queue's get method. Thus as we add or put items in the queue, the thread pool will pick up or get items and process them. This is also known as **dequeing**. Once all the items in the queue are processed, the script ends and exits. On my machine, it downloads all 5 documents in under a second.

Wrapping Up

Now you know how to use threads and queues both in theory and in a practical way. Threads are especially useful when you are creating a user interface and you want to keep your interface usable. Without threads, the user interface would become unresponsive and would appear to hang while you did a large file download or a big query against a database. To keep that from happening, you

do the long running processes in threads and then communicate back to your interface when you are done.

Chapter 22 - Working with Dates and Time

Python gives the developer several tools for working with dates and time. In this chapter, we will be looking at the `datetime` and `time` modules. We will study how they work and some common uses for them. Let's start with the `datetime` module!

The `datetime` Module

We will be learning about the following classes from the `datetime` module:

- `datetime.date`
- `datetime.timedelta`
- `datetime.datetime`

These will cover the majority of instances where you'll need to use date and datetime object in Python. There is also a `tzinfo` class for working with time zones that we won't be covering. Feel free to take a look at the Python documentation for more information on that class.

`datetime.date`

Python can represent dates several different ways. We're going to look at the `datetime.date` format first as it happens to be one of the simpler date objects.

```
1 >>> datetime.date(2012, 13, 14)
2 Traceback (most recent call last):
3   File "<string>", line 1, in <fragment>
4     builtins.ValueError: month must be in 1..12
5 >>> datetime.date(2012, 12, 14)
6 datetime.date(2012, 12, 14)
```

This code shows how to create a simple date object. The date class accepts three arguments: the year, the month and the day. If you pass it an invalid value, you will see a `ValueError`, like the one above. Otherwise you will see a `datetime.date` object returned. Let's take a look at another example:

```
1 >>> import datetime
2 >>> d = datetime.date(2012, 12, 14)
3 >>> d.year
4 2012
5 >>> d.day
6 14
7 >>> d.month
8 12
```

Here we assign the date object to the variable `d`. Now we can access the various date components by name, such as `d.year` or `d.month`. Now let's find out what day it is:

```
1 >>> datetime.date.today()
2 datetime.date(2014, 3, 5)
```

This can be helpful whenever you need to record what day it is. Or perhaps you need to do a date-based calculation based on today. It's a handy little convenience method though.

datetime.datetime

A `datetime.datetime` object contains all the information from a `datetime.date` plus a `datetime.time` object. Let's create a couple of examples so we can better understand the difference between this object and the `datetime.date` object.

```
1 >>> datetime.datetime(2014, 3, 5)
2 datetime.datetime(2014, 3, 5, 0, 0)
3 >>> datetime.datetime(2014, 3, 5, 12, 30, 10)
4 datetime.datetime(2014, 3, 5, 12, 30, 10)
5 >>> d = datetime.datetime(2014, 3, 5, 12, 30, 10)
6 >>> d.year
7 2014
8 >>> d.second
9 10
10 >>> d.hour
11 12
```

Here we can see that `datetime.datetime` accepts several additional arguments: year, month, day, hour, minute and second. It also allows you to specify microsecond and timezone information too. When you work with databases, you will find yourself using these types of objects a lot. Most of the time, you will need to convert from the Python date or datetime format to the SQL datetime or timestamp format. You can find out what today is with `datetime.datetime` using two different methods:

```
1 >>> datetime.datetime.today()  
2 datetime.datetime(2014, 3, 5, 17, 56, 10, 737000)  
3 >>> datetime.datetime.now()  
4 datetime.datetime(2014, 3, 5, 17, 56, 15, 418000)
```

The `datetime` module has another method that you should be aware of called `strftime`. This method allows the developer to create a string that represents the time in a more human readable format. There's an entire table of formatting options that you should go read in the Python documentation, section 8.1.7. We're going to look at a couple of examples to show you the power of this method:

```
1 >>> datetime.datetime.today().strftime("%Y%m%d")  
2 '20140305'  
3 >>> today = datetime.datetime.today()  
4 >>> today.strftime("%m/%d/%Y")  
5 '03/05/2014'  
6 >>> today.strftime("%Y-%m-%d-%H.%M.%S")  
7 '2014-03-05-17.59.53'
```

The first example is kind of a hack. It shows how to convert today's `datetime` object into a string that follows the YYYYMMDD (year, month, day) format. The second example is better. Here we assign today's `datetime` object to a variable called `today` and then try out two different string formatting operations. The first one adds forward slashes between the `datetime` elements and also rearranges it so that it becomes month, day, year. The last example creates a timestamp of sorts that follows a fairly typical format: YYYY-MM-DD.HH.MM.SS. If you want to go to a two-digit year, you can swap out the `%Y` for `%y`.

datetime.timedelta

The `datetime.timedelta` object represents a time duration. In other words, it is the difference between two dates or times. Let's take a look at a simple example:

```
1 >>> now = datetime.datetime.now()  
2 >>> now  
3 datetime.datetime(2014, 3, 5, 18, 13, 51, 230000)  
4 >>> then = datetime.datetime(2014, 2, 26)  
5 >>> delta = now - then  
6 >>> type(delta)  
7 <type 'datetime.timedelta'>  
8 >>> delta.days  
9 7  
10 >>> delta.seconds  
11 65631
```

We create two datetime objects here. One for today and one for a week ago. Then we take the difference between them. This returns a timedelta object which we can then use to find out the number of days or seconds between the two dates. If you need to know the number of hours or minutes between the two, you'll have to use some math to figure it out. Here's one way to do it:

```
1 >>> seconds = delta.total_seconds()
2 >>> hours = seconds // 3600
3 >>> hours
4 186.0
5 >>> minutes = (seconds % 3600) // 60
6 >>> minutes
7 13.0
```

What this tells us is that there are 186 hours and 13 minutes in a week. Note that we are using a double-forward slash as our division operator. This is known as **floor division**.

Now we're ready to move on and learn a bit about the **time** module!

The time Module

The **time** module provides the Python developer access to various time-related functions. The **time** module is based around what it known as an **epoch**, the point when time starts. For Unix systems, the epoch was in 1970. To find out what the epoch is on your system, try running the following:

```
1 >>> import time
2 >>> time.gmtime(0)
3 time.struct_time(tm_year=1970, tm_mon=1, tm_mday=1, tm_hour=0, tm_min=0, tm_sec=\
4 0, tm_wday=3, tm_yday=1, tm_isdst=0)
```

I ran this on Windows 7 and it too seems to think that time began in 1970. Anyway, in this section, we will be studying the following time-related functions:

- `time.ctime`
- `time.sleep`
- `time.strftime`
- `time.time`

Let's get started!

time.ctime

The `time.ctime` function will convert a time in seconds since the epoch to a string representing local time. If you don't pass it anything, then the current time is returned. Let's try out a couple of examples:

```
1 >>> import time
2 >>> time.ctime()
3 'Thu Mar 06 07:28:48 2014'
4 >>> time.ctime(1384112639)
5 'Sun Nov 10 13:43:59 2013'
```

Here we show the results of calling `ctime` with nothing at all and with a fairly random number of seconds since the epoch. I have seen sort of thing used when someone saves the date as seconds since the epoch and then they want to convert it to something a human can understand. It's a bit simpler to save a big integer (or long) to a database then to mess with formatting it from a datetime object to whatever date object the database accepts. Of course, that also has the drawback that you do need to convert the integer or float value back into a string.

time.sleep

The `time.sleep` function gives the developer the ability to suspend execution of your script a given number of seconds. It's like adding a pause to your program. I have found this personally useful when I need to wait a second for a file to finish closing or a database commit to finish committing. Let's take a look at an example. Open a new window in IDLE and save the following code:

```
1 import time
2
3 for x in range(5):
4     time.sleep(2)
5     print("Slept for 2 seconds")
```

Now run the code in IDLE. You can do that by going to the **Run** menu and then choose the **Run module** menu item. When you do so, you will see it print out the phrase *Slept for 2 seconds* five times with a two second pause between each print. It's really that easy to use!

time.strftime

The `time` module has a `strftime` function that works in pretty much the same manner as the `datetime` version. The difference is mainly in what it accepts for input: a tuple or a `struct_time` object, like those that are returned when you call `time.gmtime()` or `time.localtime()`. Here's a little example:

```
1 >>> time.strftime("%Y-%m-%d-%H.%M.%S",
2                      time.localtime())
3 '2014-03-06-20.35.56'
```

This code is quite similar to the timestamp code we created in the datetime portion of this chapter. I think the datetime method is a little more intuitive in that you just create a `datetime.datetime` object and then call its `strftime` method with the format you want. With the time module, you have to pass the format plus a time tuple. It's really up to you to decide which one makes the most sense to you.

time.time

The `time.time` function will return the time in seconds since the epoch as a floating point number. Let's take a look:

```
1 >>> time.time()
2 1394199262.318
```

That was pretty simple. You could use this when you want to save the current time to a database but you didn't want to bother converting it to the database's datetime method. You might also recall that the `ctime` method accepts the time in seconds, so we could use `time.time` to get the number of seconds to pass to `ctime`, like this:

```
1 >>> time.ctime(time.time())
2 'Fri Mar 07 07:36:38 2014'
```

If you do some digging in the documentation for the time module or if you just experiment with it a bit, you will likely find a few other uses for this function.

Wrapping Up

At this point you should know how to work with dates and time using Python's standard modules. Python gives you a lot of power when it comes to working with dates. You will find these modules helpful if you ever need to create an application that keeps track of appointments or that needs to run on particular days. They are also useful when working with databases.

Chapter 23 - The xml module

Python has built-in XML parsing capabilities that you can access via its `xml` module. In this article, we will be focusing on two of the `xml` module's sub-modules:

- `minidom`
- `ElementTree`

We'll start with `minidom` simply because this used to be the de-facto method of XML parsing. Then we will look at how to use `ElementTree` instead.

Working with minidom

To start out, well need some actual XML to parse. Take a look at the following short example of XML:

```
1 <?xml version="1.0" ?>
2 <zAppointments reminder="15">
3   <appointment>
4     <begin>1181251680</begin>
5     <uid>040000008200E000</uid>
6     <alarmTime>1181572063</alarmTime>
7     <state></state>
8     <location></location>
9     <duration>1800</duration>
10    <subject>Bring pizza home</subject>
11  </appointment>
12 </zAppointments>
```

This is fairly typical XML and actually pretty intuitive to read. There is some really nasty XML out in the wild that you may have to work with. Anyway, save the XML code above with the following name: `appt.xml`

Let's spend some time getting acquainted with how to parse this file using Python's `minidom` module. This is a fairly long piece of code, so prepare yourself.

```
1 import xml.dom.minidom
2 import urllib.request
3
4 class ApptParser(object):
5
6     def __init__(self, url, flag='url'):
7         self.list = []
8         self.appt_list = []
9         self.flag = flag
10        self.rem_value = 0
11        xml = self.getXml(url)
12        self.handleXml(xml)
13
14    def getXml(self, url):
15        try:
16            print(url)
17            f = urllib.request.urlopen(url)
18        except:
19            f = url
20
21        doc = xml.dom.minidom.parse(f)
22        node = doc.documentElement
23        if node.nodeType == xml.dom.Node.ELEMENT_NODE:
24            print('Element name: %s' % node.nodeName)
25            for (name, value) in node.attributes.items():
26                print('      Attr -- Name: %s  Value: %s' % (name, value))
27
28        return node
29
30    def handleXml(self, xml):
31        rem = xml.getElementsByTagName('zAppointments')
32        appointments = xml.getElementsByTagName("appointment")
33        self.handleAppts(appointments)
34
35    def getElement(self, element):
36        return self.getText(element.childNodes)
37
38    def handleAppts(self, appts):
39        for appt in appts:
40            self.handleAppt(appt)
41            self.list = []
```

```

43     def handleAppt(self, appt):
44         begin      = self.getElement(appt.getElementsByTagName("begin")[0])
45         duration   = self.getElement(appt.getElementsByTagName("duration")[0])
46         subject    = self.getElement(appt.getElementsByTagName("subject")[0])
47         location   = self.getElement(appt.getElementsByTagName("location")[0])
48         uid        = self.getElement(appt.getElementsByTagName("uid")[0])
49
50         self.list.append(begin)
51         self.list.append(duration)
52         self.list.append(subject)
53         self.list.append(location)
54         self.list.append(uid)
55         if self.flag == 'file':
56
57             try:
58                 state      = self.getElement(appt.getElementsByTagName("state")[0\
59             ])
59             ]
60                 self.list.append(state)
61                 alarm     = self.getElement(appt.getElementsByTagName("alarmTime\"\
62             ")[0])
62             )
63                 self.list.append(alarm)
64             except Exception as e:
65                 print(e)
66
67             self.appt_list.append(self.list)
68
69     def getText(self, nodelist):
70         rc = ""
71         for node in nodelist:
72             if node.nodeType == node.TEXT_NODE:
73                 rc = rc + node.data
74         return rc
75
76 if __name__ == "__main__":
77     appt = ApptParser("appt.xml")
78     print(appt.appt_list)

```

This code is loosely based on an example from the Python documentation and I have to admit that I think my mutation of it is a bit ugly. Let's break this code down a bit. The url parameter you see in the **ApptParser** class can be either a url or a file. In the **getXml** method, we use an exception handler to try and open the url. If it happens to raise an error, than we assume that the url is actually a file path. Next we use minidom's **parse** method to parse the XML. Then we pull out a node from

the XML. We'll ignore the conditional as it isn't important to this discussion. Finally, we return the `node` object.

Technically, the node is XML and we pass it on to the `handleXml` method. To grab all the appointment instances in the XML, we do this:

```
1  xml.getElementsByTagName("appointment").
```

Then we pass that information to the `handleAppts` method. That's a lot of passing information around. It might be a good idea to refactor this code a bit to make it so that instead of passing information around, it just set class variables and then called the next method without any arguments. I'll leave this as an exercise for the reader. Anyway, all the `handleAppts` method does is loop over each appointment and call the `handleAppt` method to pull some additional information out of it, add the data to a list and add that list to another list. The idea was to end up with a list of lists that held all the pertinent data regarding my appointments.

You will notice that the `handleAppt` method calls the `getElement` method which calls the `getText` method. Technically, you could skip the call to `getElement` and just call `getText` directly. On the other hand, you may need to add some additional processing to `getElement` to convert the text to some other type before returning it back. For example, you may want to convert numbers to integers, floats or decimal.Decimal objects.

Let's try one more example with minidom before we move on. We will use an XML example from Microsoft's MSDN website: <http://msdn.microsoft.com/en-us/library/ms762271%28VS.85%29.aspx>¹³. Save the following XML as `example.xml`

```
1  <?xml version="1.0"?>
2  <catalog>
3      <book id="bk101">
4          <author>Gambardella, Matthew</author>
5          <title>XML Developer's Guide</title>
6          <genre>Computer</genre>
7          <price>44.95</price>
8          <publish_date>2000-10-01</publish_date>
9          <description>An in-depth look at creating applications
10             with XML.</description>
11     </book>
12     <book id="bk102">
13         <author>Ralls, Kim</author>
14         <title>Midnight Rain</title>
15         <genre>Fantasy</genre>
16         <price>5.95</price>
```

¹³<http://msdn.microsoft.com/en-us/library/ms762271%28VS.85%29.aspx>

```
17    <publish_date>2000-12-16</publish_date>
18    <description>A former architect battles corporate zombies,
19    an evil sorceress, and her own childhood to become queen
20    of the world.</description>
21  </book>
22  <book id="bk103">
23    <author>Corets, Eva</author>
24    <title>Maeve Ascendant</title>
25    <genre>Fantasy</genre>
26    <price>5.95</price>
27    <publish_date>2000-11-17</publish_date>
28    <description>After the collapse of a nanotechnology
29    society in England, the young survivors lay the
30    foundation for a new society.</description>
31  </book>
32 </catalog>
```

For this example, we'll just parse the XML, extract the book titles and print them to stdout. Here's the code:

```
1 import xml.dom.minidom as minidom
2
3 def getTitles(xml):
4     """
5         Print out all titles found in xml
6     """
7     doc = minidom.parse(xml)
8     node = doc.documentElement
9     books = doc.getElementsByTagName("book")
10
11    titles = []
12    for book in books:
13        titleObj = book.getElementsByTagName("title")[0]
14        titles.append(titleObj)
15
16    for title in titles:
17        nodes = title.childNodes
18        for node in nodes:
19            if node.nodeType == node.TEXT_NODE:
20                print(node.data)
21
22 if __name__ == "__main__":
```

```
23     document = 'example.xml'  
24     getTitles(document)
```

This code is just one short function that accepts one argument, the XML file. We import the minidom module and give it the same name to make it easier to reference. Then we parse the XML. The first two lines in the function are pretty much the same as the previous example. We use the `getElementsByTagName` method to grab the parts of the XML that we want, then iterate over the result and extract the book titles from them. This actually extracts title objects, so we need to iterate over that as well and pull out the plain text, which is why we use a nested `for` loop.

Now let's spend a little time trying out a different sub-module of the `xml` module named `ElementTree`.

Parsing with ElementTree

In this section, you will learn how to create an XML file, edit XML and parse the XML with `ElementTree`. For comparison's sake, we'll use the same XML we used in the previous section to illustrate the differences between using `minidom` and `ElementTree`. Here is the original XML again:

```
1 <?xml version="1.0" ?>  
2 <zAppointments reminder="15">  
3   <appointment>  
4     <begin>1181251680</begin>  
5     <uid>040000008200E000</uid>  
6     <alarmTime>1181572063</alarmTime>  
7     <state></state>  
8     <location></location>  
9     <duration>1800</duration>  
10    <subject>Bring pizza home</subject>  
11  </appointment>  
12 </zAppointments>
```

Let's begin by learning how to create this piece of XML programmatically using Python!

How to Create XML with ElementTree

Creating XML with `ElementTree` is very simple. In this section, we will attempt to create the XML above with Python. Here's the code:

```
1 import xml.etree.ElementTree as xml
2
3 def createXML(filename):
4     """
5         Create an example XML file
6     """
7     root = xml.Element("zAppointments")
8     appt = xml.Element("appointment")
9     root.append(appt)
10
11     # add appointment children
12     begin = xml.SubElement(appt, "begin")
13     begin.text = "1181251680"
14
15     uid = xml.SubElement(appt, "uid")
16     uid.text = "040000008200E000"
17
18     alarmTime = xml.SubElement(appt, "alarmTime")
19     alarmTime.text = "1181572063"
20
21     state = xml.SubElement(appt, "state")
22
23     location = xml.SubElement(appt, "location")
24
25     duration = xml.SubElement(appt, "duration")
26     duration.text = "1800"
27
28     subject = xml.SubElement(appt, "subject")
29
30     tree = xml.ElementTree(root)
31     with open(filename, "w") as fh:
32         tree.write(fh)
33
34 if __name__ == "__main__":
35     createXML("appt.xml")
```

If you run this code, you should get something like the following (probably all on one line):

```

1 <zAppointments>
2   <appointment>
3     <begin>1181251680</begin>
4     <uid>040000008200E000</uid>
5     <alarmTime>1181572063</alarmTime>
6     <state />
7     <location />
8     <duration>1800</duration>
9     <subject />
10    </appointment>
11 </zAppointments>
```

This is pretty close to the original and is certainly valid XML. While it's not quite the same, it's close enough. Let's take a moment to review the code and make sure we understand it. First we create the root element by using ElementTree's Element function. Then we create an appointment element and append it to the root. Next we create SubElements by passing the appointment Element object (appt) to SubElement along with a name, like "begin". Then for each SubElement, we set its text property to give it a value. At the end of the script, we create an ElementTree and use it to write the XML out to a file.

Now we're ready to learn how to edit the file!

How to Edit XML with ElementTree

Editing XML with ElementTree is also easy. To make things a little more interesting though, we'll add another appointment block to the XML:

```

1 <?xml version="1.0" ?>
2 <zAppointments reminder="15">
3   <appointment>
4     <begin>1181251680</begin>
5     <uid>040000008200E000</uid>
6     <alarmTime>1181572063</alarmTime>
7     <state></state>
8     <location></location>
9     <duration>1800</duration>
10    <subject>Bring pizza home</subject>
11  </appointment>
12  <appointment>
13    <begin>1181253977</begin>
14    <uid>sdlkjlkadhdakhdaf</uid>
15    <alarmTime>1181588888</alarmTime>
```

```
16      <state>TX</state>
17      <location>Dallas</location>
18      <duration>1800</duration>
19      <subject>Bring pizza home</subject>
20  </appointment>
21 </zAppointments>
```

Now let's write some code to change each of the begin tag's values from seconds since the epoch to something a little more readable. We'll use Python's `time` module to facilitate this:

```
1 import time
2 import xml.etree.ElementTree as ET
3
4 def editXML(filename):
5     """
6     Edit an example XML file
7     """
8     tree = ET.ElementTree(file=filename)
9     root = tree.getroot()
10
11    for begin_time in root.iter("begin"):
12        begin_time.text = time.ctime(int(begin_time.text))
13
14    tree = ET.ElementTree(root)
15    with open("updated.xml", "w") as f:
16        tree.write(f)
17
18 if __name__ == "__main__":
19    editXML("original_appt.xml")
```

Here we create an `ElementTree` object (`tree`) and we extract the `root` from it. Then we use `ElementTree`'s `iter()` method to find all the tags that are labeled “begin”. Note that the `iter()` method was added in Python 2.7. In our for loop, we set each item's `text` property to a more human readable time format via `time.ctime()`. You'll note that we had to convert the string to an integer when passing it to `ctime`. The output should look something like the following:

```

1 <zAppointments reminder="15">
2   <appointment>
3     <begin>Thu Jun 07 16:28:00 2007</begin>
4     <uid>040000008200E000</uid>
5     <alarmTime>1181572063</alarmTime>
6     <state />
7     <location />
8     <duration>1800</duration>
9     <subject>Bring pizza home</subject>
10    </appointment>
11    <appointment>
12      <begin>Thu Jun 07 17:06:17 2007</begin>
13      <uid>sdlkjlkadhdakhdffd</uid>
14      <alarmTime>1181588888</alarmTime>
15      <state>TX</state>
16      <location>Dallas</location>
17      <duration>1800</duration>
18      <subject>Bring pizza home</subject>
19    </appointment>
20  </zAppointments>
```

You can also use ElementTree's `find()` or `findall()` methods to search for specific tags in your XML. The `find()` method will just find the first instance whereas the `findall()` will find all the tags with the specified label. These are helpful for editing purposes or for parsing, which is our next topic!

How to Parse XML with ElementTree

Now we get to learn how to do some basic parsing with ElementTree. First we'll read through the code and then we'll go through bit by bit so we can understand it. Note that this code is based around the original example, but it should work on the second one as well.

```

1 import xml.etree.cElementTree as ET
2
3 def parseXML(xml_file):
4   """
5     Parse XML with ElementTree
6   """
7   tree = ET.ElementTree(file=xml_file)
8   print(tree.getroot())
9   root = tree.getroot()
10  print("tag=%s, attrib=%s" % (root.tag, root.attrib))
```

```

11
12     for child in root:
13         print(child.tag, child.attrib)
14         if child.tag == "appointment":
15             for step_child in child:
16                 print(step_child.tag)
17
18     # iterate over the entire tree
19     print("-" * 40)
20     print("Iterating using a tree iterator")
21     print("-" * 40)
22     iter_ = tree.getiterator()
23     for elem in iter_:
24         print(elem.tag)
25
26     # get the information via the children!
27     print("-" * 40)
28     print("Iterating using getchildren()")
29     print("-" * 40)
30     appointments = root.getchildren()
31     for appointment in appointments:
32         appt_children = appointment.getchildren()
33         for appt_child in appt_children:
34             print("%s=%s" % (appt_child.tag, appt_child.text))
35
36 if __name__ == "__main__":
37     parseXML("appt.xml")

```

You may have already noticed this, but in this example and the last one, we've been importing cElementTree instead of the normal ElementTree. The main difference between the two is that cElementTree is C-based instead of Python-based, so it's much faster. Anyway, once again we create an ElementTree object and extract the root from it. You'll note that we print out the root and the root's tag and attributes. Next we show several ways of iterating over the tags. The first loop just iterates over the XML child by child. This would only print out the top level child (appointment) though, so we added an if statement to check for that child and iterate over its children too.

Next we grab an iterator from the tree object itself and iterate over it that way. You get the same information, but without the extra steps in the first example. The third method uses the root's `getchildren()` function. Here again we need an inner loop to grab all the children inside each appointment tag. The last example uses the root's `iter()` method to just loop over any tags that match the string "begin".

As mentioned in the last section, you could also use `find()` or `findall()` to help you find specific tags or sets of tags respectively. Also note that each Element object has a `tag` and a `text` property that

you can use to acquire that exact information.

Wrapping Up

Now you know how to use minidom to parse XML. You have also learned how to use ElementTree to create, edit and parse XML. There are other libraries outside of Python that provide additional methods for working with XML too. Be sure you do some research to make sure you're using a tool that you understand as this topic can get pretty confusing if the tool you're using is obtuse.

Part III - Intermediate Odds and Ends

In Part III, you will learn about some Python internals that many would consider intermediate-level Python. You have graduated from the milk and you're ready for some meat! In this section, we will take a look at the following topics:

- Debugging
- Decorators
- The lambda statement
- Profiling your code
- Testing

The first chapter in this section will introduce you to Python's debugging module, `pdb` and how to use it to debug your code. The next chapter is all about decorators. You will learn about how to create them and about a few of the decorators that are built into Python. For the third chapter, we will be looking at the `lambda` statement, which basically creates a one-line anonymous function. It's a little weird, but fun! The fourth chapter will cover how you profile your code. This discipline gives you the ability to find possible bottlenecks in your code so that you know where to focus to optimize your code. The final chapter in this section is about testing your code. In it you will discover how to test your code using a couple of Python's own built-in modules.

I think you will find this section very helpful in continuing your Python education. Let's jump right in!



image

Chapter 24 - The Python Debugger

Python comes with its own debugger module that is named **pdb**. This module provides an interactive source code debugger for your Python programs. You can set breakpoints, step through your code, inspect stack frames and more. We will look at the following aspects of the module:

- How to start the debugger
- Stepping through your code
- Setting breakpoints

Let's start by creating a quick piece of code to attempt debugging with. Here's a silly example:

```
1 # debug_test.py
2
3 def doubler(a):
4     """
5     result = a*2
6     print(result)
7     return result
8
9 def main():
10    """
11    for i in range(1,10):
12        doubler(i)
13
14 if __name__ == "__main__":
15    main()
```

Now let's learn how to run the debugger against this piece of code.

How to Start the Debugger

You can start the debugger three different ways. The first is to just import it and insert **pdb.set_trace()** into your code to start the debugger. You can import the debugger in IDLE and have it run your module. Or you can call the debugger on the command line. We'll focus on the last two methods in this section. We will start with using it in the interpreter (IDLE). Open up a terminal (command line window) and navigate to where you save the above code example. Then start Python. Now do the following:

```
1 >>> import debug_test
2 >>> import pdb
3 >>> pdb.run('debug_test.main()')
4 > <string>(1)<module>()
5 (Pdb) continue
6 2
7 4
8 6
9 8
10 10
11 12
12 14
13 16
14 18
15 >>>
```

Here we import our module and pdb. Then we execute pdb's `run` method and tell it to call our module's `main` method. This brings up the debugger's prompt. Here we typed `continue` to tell it to go ahead and run the script. You can also type the letter `c` as a shortcut for `continue`. When you `continue`, the debugger will continue execution until it reaches a breakpoint or the script ends.

The other way to start the debugger is to execute the following command via your terminal session:

```
1 python -m pdb debug_test.py
```

If you run it this way, you will see a slightly different result:

```
1 -> def doubler(a):
2 (Pdb) c
3 2
4 4
5 6
6 8
7 10
8 12
9 14
10 16
11 18
12 The program finished and will be restarted
```

You will note that in this example we used `c` instead of `continue`. You will also note that the debugger restarts at the end. This preserves the debugger's state (such as breakpoints) and can be more useful

than having the debugger stop. Sometimes you'll need to go through the code several times to understand what's wrong with it.

Let's dig a little deeper and learn how to step through the code.

Stepping Through the Code

If you want to step through your code one line at a time, then you can use the **step** (or simply “**s**”) command. Here's a session for your viewing pleasure:

```
1 C:\Users\mike>cd c:\py101
2
3 c:\py101>python -m pdb debug_test.py
4 > c:\py101\debug_test.py(4)<module>()
5 -> def doubler(a):
6     (Pdb) step
7 > c:\py101\debug_test.py(11)<module>()
8 -> def main():
9     (Pdb) s
10    > c:\py101\debug_test.py(16)<module>()
11 -> if __name__ == "__main__":
12     (Pdb) s
13    > c:\py101\debug_test.py(17)<module>()
14 -> main()
15     (Pdb) s
16 --Call--
17    > c:\py101\debug_test.py(11)main()
18 -> def main():
19     (Pdb) next
20    > c:\py101\debug_test.py(13)main()
21 -> for i in range(1,10):
22     (Pdb) s
23    > c:\py101\debug_test.py(14)main()
24 -> doubler(i)
25     (Pdb)
```

Here we start up the debugger and tell it to step into the code. It starts at the top and goes through the first two function definitions. Then it reaches the conditional and finds that it's supposed to execute the **main** function. We step into the main function and then use the **next** command. The **next** command will execute a called function if it encounters it without stepping into it. If you want to step into the called function, then you'll only want to just use the **step** command.

When you see a line like > c:py101debug_test.py(13)main(), you will want to pay attention to the number that's in the parentheses. This number is the current line number in the code.

You can use the **args** (or **a**) to print the current argument list to the screen. Another handy command is **jump** (or **j**) followed by a space and the line number that you want to “jump” to. This gives you the ability to skip a bunch of monotonous stepping to get to the line that you want to get to. This leads us to learning about breakpoints!

Setting breakpoints

A breakpoint is a line in the code where you want to pause execution. You can set a breakpoint by calling the **break** (or **b**) command followed by a space and the line number that you want to break on. You can also prefix the line number with a filename and colon to specify a breakpoint in a different file. The **break** command also allows you to set a breakpoint with the **function** argument. There is also a **tbreak** command which will set a temporary breakpoint which is automatically removed when it gets hit.

Here's an example:

```
1 c:\py101>python -m pdb debug_test.py
2 > c:\py101\debug_test.py(4)<module>()
3 -> def doubler(a):
4     (Pdb) break 6
5 Breakpoint 1 at c:\py101\debug_test.py:6
6 (Pdb) c
7 > c:\py101\debug_test.py(6)doubler()
8 -> result = a*2
```

We start the debugger and then tell it to set a breakpoint on line 6. Then we continue and it stops on line 6 just like it's supposed to. Now would be a good time to check the argument list to see if it's what you expect. Give it a try by typing **args** now. Then do another **continue** and another **args** to see how it changed.

Wrapping Up

There are a lot of other commands that you can use in the debugger. I recommend reading the documentation to learn about the others. However, at this point you should be able to use the debugger effectively to debug your own code.

Chapter 25 - Decorators

Python decorators are really cool, but they can be a little hard to understand at first. A decorator in Python is a function that accepts another function as an argument. The decorator will usually modify or enhance the function it accepted and return the modified function. This means that when you call a decorated function, you will get a function that may be a little different than the base definition. But let's back up a bit. We should probably review the basic building block of a decorator, namely, the function.

A Simple Function

A function is a block of code that begins with the Python keyword `def` followed by the actual name of the function. A function can accept zero or more arguments, keyword arguments or a mixture of the two. A function always returns something. If you do not specify what a function should return, it will return `None`. Here is a very simple function that just returns a string:

```
1 def a_function():
2     """A pretty useless function"""
3     return "1+1"
4
5 if __name__ == "__main__":
6     value = a_function()
7     print(value)
```

All we do in the code above is call the function and print the return value. Let's create another function:

```
1 def another_function(func):
2     """
3     A function that accepts another function
4     """
5     def other_func():
6         val = "The result of %s is %s" % (func(),
7                                         eval(func()))
8         )
9         return val
10    return other_func
```

This function accepts one argument and that argument has to be a function or callable. In fact, it really should only be called using the previously defined function. You will note that this function has a nested function inside of it that we are calling **other_func**. It will take the result of the function passed to it, evaluate it and create a string that tells us about what it did, which it then returns. Let's look at the full version of the code:

```
1 def another_function(func):
2     """
3     A function that accepts another function
4     """
5
6     def other_func():
7         val = "The result of %s is %s" % (func(),
8                                         eval(func()))
9         )
10
11     return val
12
13     return other_func
14
15 def a_function():
16     """A pretty useless function"""
17     return "1+1"
18
19 if __name__ == "__main__":
20     value = a_function()
21     print(value)
22     decorator = another_function(a_function)
23     print(decorator())
```

This is how a decorator works. We create one function and then pass it into a second function. The second function is the **decorator** function. The decorator will modify or enhance the function that was passed to it and return the modification. If you run this code, you should see the following as output to stdout:

```
1 1+1
2 The result of 1+1 is 2
```

Let's change the code slightly to turn **another_function** into a decorator:

```
1 def another_function(func):
2     """
3     A function that accepts another function
4     """
5
6     def other_func():
7         val = "The result of %s is %s" % (func(),
8                                         eval(func()))
9         )
10        return val
11    return other_func
12
13 @another_function
14 def a_function():
15     """A pretty useless function"""
16     return "1+1"
17
18 if __name__ == "__main__":
19     value = a_function()
20     print(value)
```

You will note that in Python, a decorator starts with the `<@*>*` symbol followed by the name of the function that we will be using to “decorate” our regular with. To apply the decorator, you just put it on the line before the function definition. Now when we call `**a_function`, it will get decorated and we’ll get the following result:

```
1 The result of 1+1 is 2
```

Let’s create a decorator that actually does something useful.

Creating a Logging Decorator

Sometimes you will want to create a log of what a function is doing. Most of the time, you will probably be doing your logging within the function itself. Occasionally you might want to do it at the function level to get an idea of the flow of the program or perhaps to fulfill some business rules, like auditing. Here’s a little decorator that we can use to record any function’s name and what it returns:

```
1 import logging
2
3 def log(func):
4     """
5     Log what function is called
6     """
7     def wrap_log(*args, **kwargs):
8         name = func.__name__
9         logger = logging.getLogger(name)
10        logger.setLevel(logging.INFO)
11
12        # add file handler
13        fh = logging.FileHandler("%s.log" % name)
14        fmt = '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
15        formatter = logging.Formatter(fmt)
16        fh.setFormatter(formatter)
17        logger.addHandler(fh)
18
19        logger.info("Running function: %s" % name)
20        result = func(*args, **kwargs)
21        logger.info("Result: %s" % result)
22        return func
23    return wrap_log
24
25 @log
26 def double_function(a):
27     """
28     Double the input parameter
29     """
30     return a*2
31
32 if __name__ == "__main__":
33     value = double_function(2)
```

This little script has a `log` function that accepts a function as its sole argument. It will create a logger object and a log file name based on the name of the function. Then the `log` function will log what function was called and what the function returned, if anything.

Built-in Decorators

Python comes with several built-in decorators. The big three are:

- `@classmethod`
- `@staticmethod`
- `@property`

There are also decorators in various parts of Python's standard library. One example would be `functools.wraps`. We will be limiting our scope to the three above though.

@classmethod and @staticmethod

I have never actually used these myself, so I did a fair bit of research. The `<*@classmethod>` decorator can be called with with an instance of a class or directly by the class itself as its first argument. According to the Python documentation: *It can be called either on the class (such as `C.f()`) or on an instance (such as `C().f()`). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.* The primary use case of a `@classmethod` decorator that I have found in my research is as an alternate constructor or helper method for initialization.

The `<*@staticmethod>` decorator is just a function inside of a class. You can call it both with and without instantiating the class. A typical use case is when you have a function where you believe it has a connection with a class. It's a stylistic choice for the most part.

It might help to see a code example of how these two decorators work:

```
1  class DecoratorTest(object):
2      """
3          Test regular method vs @classmethod vs @staticmethod
4      """
5
6      def __init__(self):
7          """Constructor"""
8          pass
9
10     def doubler(self, x):
11         """
12             print("running doubler")
13             return x*2
14
15     @classmethod
16     def class_tripler(klass, x):
17         """
18             print("running tripler: %s" % klass)
19             return x*3
20
```

```
21     @staticmethod
22     def static_quad(x):
23         """
24             print("running quad")
25             return x*4
26
27 if __name__ == "__main__":
28     decor = DecoratorTest()
29     print(decor.doubler(5))
30     print(decor.class_tripler(3))
31     print(DecoratorTest.class_tripler(3))
32     print(DecoratorTest.static_quad(2))
33     print(decor.static_quad(3))
34
35     print(decor.doubler)
36     print(decor.class_tripler)
37     print(decor.static_quad)
```

This example demonstrates that you can call a regular method and both decorated methods in the same way. You will notice that you can call both the `@classmethod` and the `@staticmethod` decorated functions directly from the class or from an instance of the class. If you try to call a regular function with the class (i.e. `DecoratorTest.doubler(2)`) you will receive a `TypeError`. You will also note that the last print statement shows that `decor.static_quad` returns a regular function instead of a bound method.

Python Properties

Python has a neat little concept called a property that can do several useful things. We will be looking into how to do the following:

- Convert class methods into read-only attributes
- Reimplement setters and getters into an attribute

One of the simplest ways to use a property is to use it as a decorator of a method. This allows you to turn a class method into a class attribute. I find this useful when I need to do some kind of combination of values. Others have found it useful for writing conversion methods that they want to have access to as methods. Let's take a look at a simple example:

```

1 class Person(object):
2     """
3
4     def __init__(self, first_name, last_name):
5         """Constructor"""
6         self.first_name = first_name
7         self.last_name = last_name
8
9     @property
10    def full_name(self):
11        """
12            Return the full name
13        """
14        return "%s %s" % (self.first_name, self.last_name)

```

In the code above, we create two class attributes or properties: `self.first_name` and `self.last_name`. Next we create a `full_name` method that has a `@property` decorator attached to it. This allows us to do the following in an interpreter session:

```

1 >>> person = Person("Mike", "Driscoll")
2 >>> person.full_name
3 'Mike Driscoll'
4 >>> person.first_name
5 'Mike'
6 >>> person.full_name = "Jackalope"
7 Traceback (most recent call last):
8   File "<string>", line 1, in <fragment>
9 AttributeError: can't set attribute

```

As you can see, because we turned the method into a property, we can access it using normal dot notation. However, if we try to set the property to something different, we will cause an `AttributeError` to be raised. The only way to change the `full_name` property is to do so indirectly:

```

1 >>> person.first_name = "Dan"
2 >>> person.full_name
3 'Dan Driscoll'

```

This is kind of limiting, so let's look at another example where we can make a property that does allow us to set it.

Replacing Setters and Getters with a Python property

Let's pretend that we have some legacy code that someone wrote who didn't understand Python very well. If you're like me, you've already seen this kind of code before:

```
1 from decimal import Decimal
2
3 class Fees(object):
4     """
5
6     def __init__(self):
7         """Constructor"""
8         self._fee = None
9
10    def get_fee(self):
11        """
12            Return the current fee
13        """
14        return self._fee
15
16    def set_fee(self, value):
17        """
18            Set the fee
19        """
20        if isinstance(value, str):
21            self._fee = Decimal(value)
22        elif isinstance(value, Decimal):
23            self._fee = value
```

To use this class, we have to use the setters and getters that are defined:

```
1 >>> f = Fees()
2 >>> f.set_fee("1")
3 >>> f.get_fee()
4 Decimal('1')
```

If you want to add the normal dot notation access of attributes to this code without breaking all the applications that depend on this piece of code, you can change it very simply by adding a property:

```
1 from decimal import Decimal
2
3 class Fees(object):
4     """
5
6     def __init__(self):
7         """Constructor"""
8         self._fee = None
9
10    def get_fee(self):
11        """
12            Return the current fee
13        """
14        return self._fee
15
16    def set_fee(self, value):
17        """
18            Set the fee
19        """
20        if isinstance(value, str):
21            self._fee = Decimal(value)
22        elif isinstance(value, Decimal):
23            self._fee = value
24
25    fee = property(get_fee, set_fee)
```

We added one line to the end of this code. Now we can do stuff like this:

```
1 >>> f = Fees()
2 >>> f.set_fee("1")
3 >>> f.fee
4 Decimal('1')
5 >>> f.fee = "2"
6 >>> f.get_fee()
7 Decimal('2')
```

As you can see, when we use `property` in this manner, it allows the `fee` property to set and get the value itself without breaking the legacy code. Let's rewrite this code using the `property` decorator and see if we can get it to allow setting.

```
1 from decimal import Decimal
2
3 class Fees(object):
4     """
5
6     def __init__(self):
7         """Constructor"""
8         self._fee = None
9
10    @property
11    def fee(self):
12        """
13        The fee property - the getter
14        """
15        return self._fee
16
17    @fee.setter
18    def fee(self, value):
19        """
20        The setter of the fee property
21        """
22        if isinstance(value, str):
23            self._fee = Decimal(value)
24        elif isinstance(value, Decimal):
25            self._fee = value
26
27 if __name__ == "__main__":
28     f = Fees()
```

The code above demonstrates how to create a “setter” for the `fee` property. You can do this by decorating a second method that is also called `fee` with a decorator called `<@fee.setter>`. The setter is invoked when you do something like this:

```
1 >>> f = Fees()
2 >>> f.fee = "1"
```

If you look at the signature for `property`, it has `fget`, `fset`, `fdel` and `doc` as “arguments”. You can create another decorated method using the same name to correspond to a delete function using `<@fee.deleter*>*` if you want to catch the `**del` command against the attribute.

Wrapping Up

At this point you should know how to create your own decorators and how to use a few of Python's built-in decorators. We looked at `@classmethod`, `@property` and `@staticmethod`. I would be curious to know how my readers use the built-in decorators and how they use their own custom decorators.

Chapter 26 - The lambda

The Python lambda statement is an anonymous or unbound function and a pretty limited function at that. Let's take a look at a few typical examples and see if we can find a use case for it. The typical examples that one normally sees for teaching the lambda are some sort of boring doubling function. Just to be contrary, our simple example will show how to find the square root. First we'll show a normal function and then the lambda equivalent:

```
1 import math
2
3 def sqroot(x):
4     """
5     Finds the square root of the number passed in
6     """
7     return math.sqrt(x)
8
9 square_rt = lambda x: math.sqrt(x)
```

If you try each of these functions, you'll end up with a float. Here are a couple examples:

```
1 >>> sqroot(49)
2 7.0
3 >>> square_rt(64)
4 8.0
```

Pretty slick, right? But where would we actually use a lambda in real life? Maybe a calculator program? Well, that would work, but it's a pretty limited application for a builtin of Python! One of the major pieces of Python that lambda examples are applied to regularly are Tkinter callbacks. Tkinter is a toolkit for building GUIs that is included with Python.

Tkinter + lambda

We'll start with Tkinter since it's included with the standard Python package. Here's a really simple script with three buttons, two of which are bound to their event handler using a lambda:

```
1 import Tkinter as tk
2
3 class App:
4     """
5
6     def __init__(self, parent):
7         """Constructor"""
8         frame = tk.Frame(parent)
9         frame.pack()
10
11        btn22 = tk.Button(frame, text="22", command=lambda: self.printNum(22))
12        btn22.pack(side=tk.LEFT)
13        btn44 = tk.Button(frame, text="44", command=lambda: self.printNum(44))
14        btn44.pack(side=tk.LEFT)
15
16        quitBtn = tk.Button(frame, text="QUIT", fg="red", command=frame.quit)
17        quitBtn.pack(side=tk.LEFT)
18
19
20    def printNum(self, num):
21        """
22        print("You pressed the %s button" % num)
23
24 if __name__ == "__main__":
25    root = tk.Tk()
26    app = App(root)
27    root.mainloop()
```

Notice the btn22 and btn44 variables. This is where the action is. We create a tk.Button instance here and bind to our printNum method in one fell swoop. The lambda is assigned to the button's command parameter. What this means is that we're creating a one-off function for the command, much like in the quit button where we call the frame's quit method. The difference here is that this particular lambda is a method that calls another method and passes the latter an integer. In the printNum method, we print to stdout which button was pressed by using the information that was passed to it from the lambda function. Did you follow all that? If so, we can continue; if not, re-read this paragraph as many times as necessary until the information sinks in or you go crazy, whichever comes first.

Wrapping Up

The lambda statement is used in all kinds of other projects as well. If you Google a Python project name and lambda, you can find lots of live code out there. For example, if you search for "django

lambda”, you’ll find out that django has a **modelformset** factory that utilizes lambdas. The Elixir plugin for SQLAlchemy also uses lambdas. Keep your eyes open and you’ll be surprised how many times you’ll stumble across this handy little function maker.

Chapter 27 - Code Profiling

Code profiling is an attempt to find bottlenecks in your code. Profiling is supposed to find what parts of your code take the longest. Once you know that, then you can look at those pieces of your code and try to find ways to optimize it. Python comes with three profilers built in: `cProfile`, `profile` and `hotshot`. According to the Python documentation, `hotshot` is “no longer maintained and may be dropped in a future version of Python”. The `profile` module is a pure Python module, but adds a lot of overhead to profiled programs. Thus we will be focusing on `cProfile`, which has an interface that mimics the `profile` module.

Profiling Your Code with `cProfile`

Profiling code with `cProfile` is really quite easy. All you need to do is import the module and call its `run` function. Let’s look at a simple example:

```
1 >>> import hashlib
2 >>> import cProfile
3 >>> cProfile.run("hashlib.md5(b'abcdefghijklmnopqrstuvwxyz').digest()")
4     4 function calls in 0.000 CPU seconds
5
6 Ordered by: standard name
7
8   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
9       1    0.000    0.000    0.000    0.000 <string>:1(<module>)
10      1    0.000    0.000    0.000    0.000 {__builtin__.open}
11      1    0.000    0.000    0.000    0.000 {method 'digest' of '__builtin__.HASH'
12 'objects'}
13      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof'
14 'iler' objects}
```

Here we import the `hashlib` module and use `cProfile` to profile the creation of an MD5 hash. The first line shows that there were 4 function calls. The next line tells us how the results are ordered. According to the documentation, standard name refers to the far right column. There are a number of columns here.

- `ncalls` is the number of calls made.
- `tottime` is a total of the time spent in the given function.

- **percall** refers to the quotient of tottime divided by ncalls
- **cumtime** is the cumulative time spent in this and all subfunctions. It's even accurate for recursive functions!
- The second **percall** column is the quotient of cumtime divided by primitive calls
- **filename:lineno(function)** provides the respective data of each function

|

A primitive call is one that was not induced via recursion.

This isn't a very interesting example as there are no obvious bottlenecks. Let's create a piece of code with some built in bottlenecks and see if the profiler detects them.

```
1 import time
2
3 def fast():
4     #####
5     print("I run fast!")
6
7 def slow():
8     #####
9     time.sleep(3)
10    print("I run slow!")
11
12 def medium():
13    #####
14    time.sleep(0.5)
15    print("I run a little slowly...")
16
17 def main():
18    #####
19    fast()
20    slow()
21    medium()
22
23 if __name__ == '__main__':
24    main()
```

In this example, we create four functions. The first three run at different rates. The **fast** function will run at normal speed; the **medium** function will take approximately half a second to run and the **slow** function will take around 3 seconds to run. The **main** function calls the other three. Now let's run cProfile against this silly little program:

```
1 >>> import cProfile
2 >>> import ptest
3 >>> cProfile.run('ptest.main()')
4 I run fast!
5 I run slow!
6 I run a little slowly...
7     8 function calls in 3.500 seconds
8
9 Ordered by: standard name
10
11 ncalls  tottime  percall  cumtime  percall filename:lineno(function)
12      1    0.000    0.000    3.500    3.500 <string>:1(<module>)
13      1    0.000    0.000    0.500    0.500 ptest.py:15(medium)
14      1    0.000    0.000    3.500    3.500 ptest.py:21(main)
15      1    0.000    0.000    0.000    0.000 ptest.py:4(fast)
16      1    0.000    0.000    3.000    3.000 ptest.py:9(slow)
17      1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof\
18 ilder' objects}
19      2    3.499    1.750    3.499    1.750 {time.sleep}
```

This time around we see the program took 3.5 seconds to run. If you examine the results, you will see that cProfile has identified the **slow** function as taking 3 seconds to run. That's the biggest bottleneck after the **main** function. Normally when you find a bottleneck like this, you would try to find a faster way to execute your code or perhaps decide that the runtime was acceptable. In this example, we know that the best way to speed up the function is to remove the `time.sleep` call or at least reduce the sleep length.

You can also call cProfile on the command line rather than using it in the interpreter. Here's one way to do it:

```
1 python -m cProfile ptest.py
```

This will run cProfile against your script in much the same way as we did before. But what if you want to save the profiler's output? Well, that's easy with cProfile! All you need to do is pass it the **-o** command followed by the name (or path) of the output file. Here's an example:

```
1 python -m cProfile -o output.txt ptest.py
```

Unfortunately, the file it outputs isn't exactly human-readable. If you want to read the file, then you'll need to use Python's **pstats** module. You can use **pstats** to format the output in various ways. Here's some code that shows how to get some output that's similar to what we've seen so far:

```
1  >>> import pstats
2  >>> p = pstats.Stats("output.txt")
3  >>> p.strip_dirs().sort_stats(-1).print_stats()
4  Thu Mar 20 18:32:16 2014      output.txt
5
6      8 function calls in 3.501 seconds
7
8  Ordered by: standard name
9
10 ncalls  tottime  percall  cumtime  percall  filename:lineno(function)
11      1    0.000    0.000    3.501    3.501  ptest.py:1(<module>)
12      1    0.001    0.001    0.500    0.500  ptest.py:15(medium)
13      1    0.000    0.000    3.501    3.501  ptest.py:21(main)
14      1    0.001    0.001    0.001    0.001  ptest.py:4(fast)
15      1    0.001    0.001    3.000    3.000  ptest.py:9(slow)
16      1    0.000    0.000    0.000    0.000  {method 'disable' of '_lsprof.Prof\
17 iler' objects}
18      2    3.499    1.750    3.499    1.750  {time.sleep}
19
20
21 <pstats.Stats instance at 0x017C9030>
```

The `strip_dirs` call will strip out all the paths to the modules from the output while the `sort_stats` call does the sorting that we're used to seeing. There are a bunch of really interesting examples in the `cProfile` documentation showing different ways to extract information using the `pstats` module.

Wrapping Up

At this point you should be able to use the `cProfile` module to help you diagnose why your code is so slow. You might also want to take a look at Python's `timeit` module. It allows you to time small pieces of your code if you don't want to deal with the complexities involved with profiling. There are also several other 3rd party modules that are good for profiling such as the `line_profiler` and the `memory_profiler` projects.

Chapter 28 - An Intro to Testing

Python includes a couple of built-in modules for testing your code. They two methods are called **doctest** and **unittest**. We will look at how to use **doctest** first and in the second section we will introduce unit tests using Test Driven Development techniques.

Testing with doctest

The doctest module will search for pieces of text in your code that resemble interactive Python sessions. It will then execute those sessions to verify that they work exactly as written. This means that if you wrote an example in a docstring that showed the output with a trailing space or tab, then the actual output of the function has to have that trailing whitespace too. Most of the time, the docstring is where you will want to put your tests. The following aspects of doctest will be covered:

- How to run doctest from the terminal
- How to use doctest inside a module
- How to run a doctest from a separate file

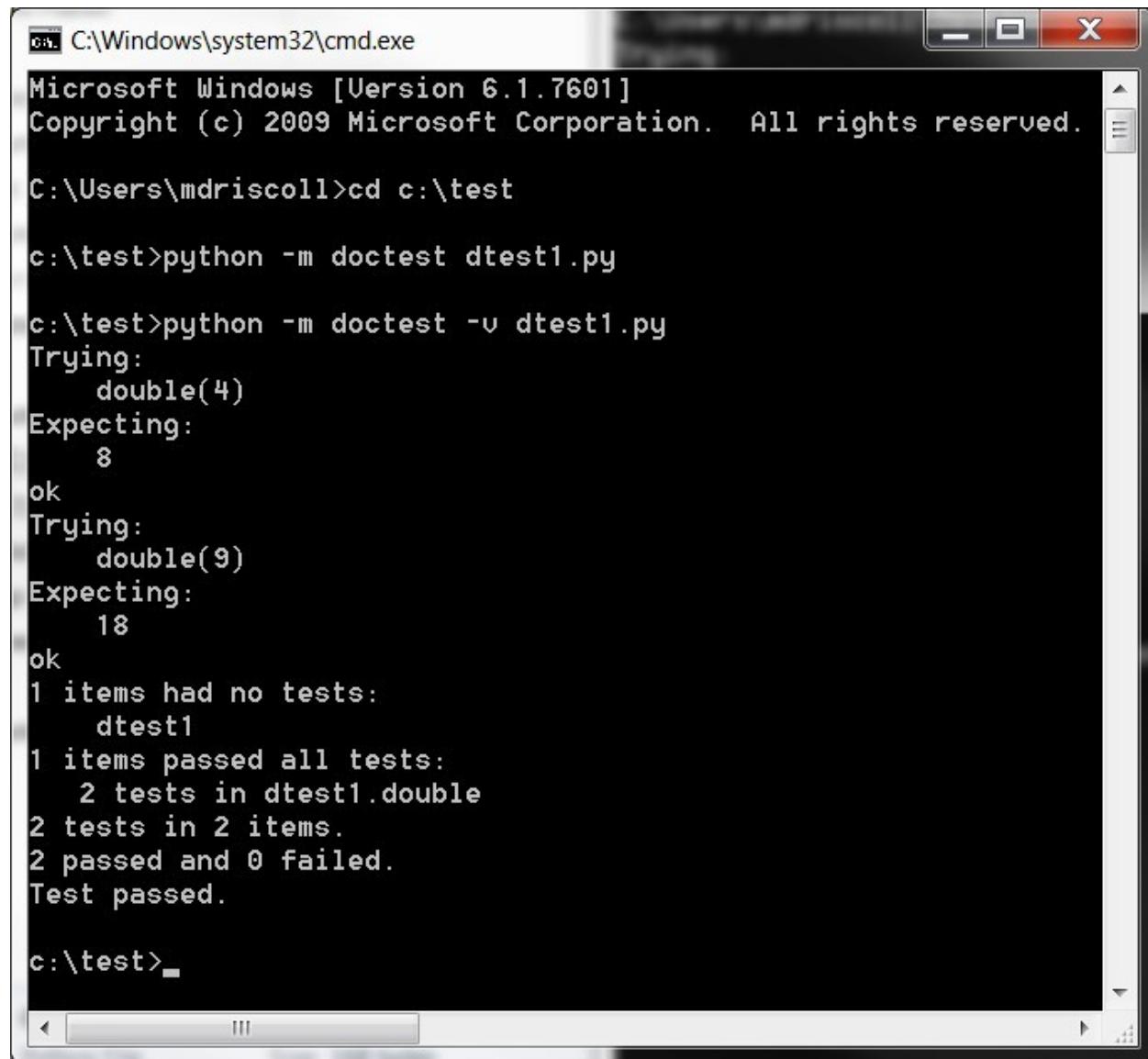
Let's get started!

Running doctest via the Terminal

We will start by creating a really simple function that will double whatever is given to it. We will include a couple of tests inside the function's **docstring**. Here's the code (be sure and save it as "dtest1.py"):

```
1 # dtest1.py
2
3 def double(a):
4     """
5     >>> double(4)
6     8
7     >>> double(9)
8     18
9     """
10    return a*2
```

Now we just need to run this code in `doctest`. Open up a terminal (or command line) and change directories to the folder that contains your script. Here's a screenshot of what I did:



The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text output:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mdriscoll>cd c:\test

c:\test>python -m doctest dtest1.py

c:\test>python -m doctest -v dtest1.py
Trying:
    double(4)
Expecting:
    8
ok
Trying:
    double(9)
Expecting:
    18
ok
1 items had no tests:
    dtest1
1 items passed all tests:
    2 tests in dtest1.double
2 tests in 2 items.
2 passed and 0 failed.
Test passed.

c:\test>
```

image

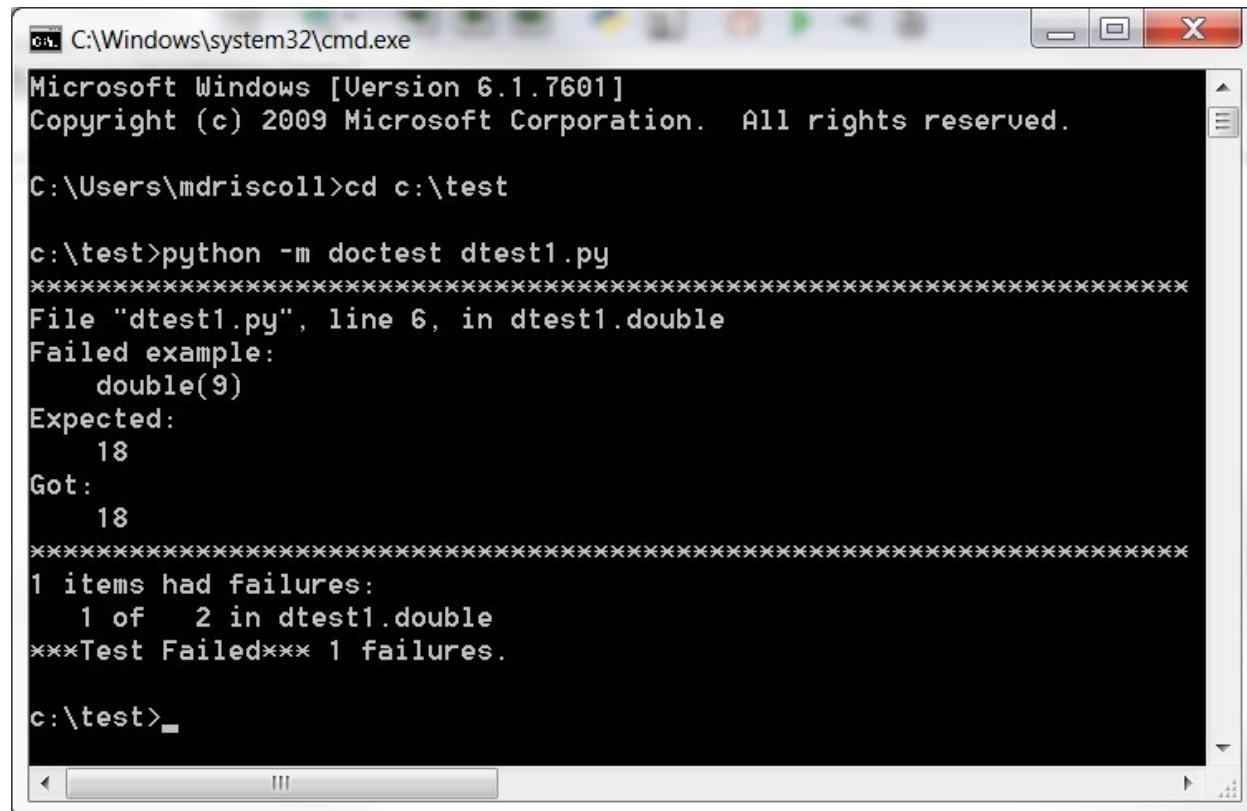
You will notice that in the first example, I executed the following:

```
1 python -m doctest dtest1.py
```

That ran the test and nothing printed out to the screen. When you don't see anything printed, that means that all the tests passed successfully. The second example shows the following command:

```
1 python -m doctest -v dtest1.py
```

The “-v” means that we want verbose output, which is exactly what we received. Open up the code again and add a space after the “18” in the docstring. Then re-run the test. Here’s the output I received:



The screenshot shows a Windows Command Prompt window titled "C:\Windows\system32\cmd.exe". The output of the command "python -m doctest dtest1.py" is displayed. The output shows a failed test example where the expected value is 18 and the got value is also 18. The error message indicates that 1 item had failures, specifically in the "dtest1.double" function.

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mdriscoll>cd c:\test

c:\test>python -m doctest dtest1.py
*****
File "dtest1.py", line 6, in dtest1.double
Failed example:
    double(9)
Expected:
    18
Got:
    18
*****
1 items had failures:
   1 of   2 in dtest1.double
***Test Failed*** 1 failures.

c:\test>
```

image

The error message says it expected “18” and it got “18”. What’s going on here? Well we added a space after “18” to our docstring, so doctest actually expected the number “18” followed by a space. Also beware of putting dictionaries as output in your docstring examples. Dictionaries can be in any order, so the likelihood of it matching your actual output isn’t very good.

Running doctest Inside a Module

Let’s modify the example slightly so that we import the `doctest` module and use its `testmod` function.

```
1 def double(a):
2     """
3     >>> double(4)
4     8
5     >>> double(9)
6     18
7     """
8     return a*2
9
10 if __name__ == "__main__":
11     import doctest
12     doctest.testmod(verbose=True)
```

Here we import `doctest` and call `doctest.testmod`. We pass it the keyword argument of `verbose=True` so that we can see some output. Otherwise, this script will run with no output whatsoever, signifying that the tests ran successfully.

If you do not want to hard-code the verbose option, you can also do it on the command line:

```
1 python dtest2.py -v
```

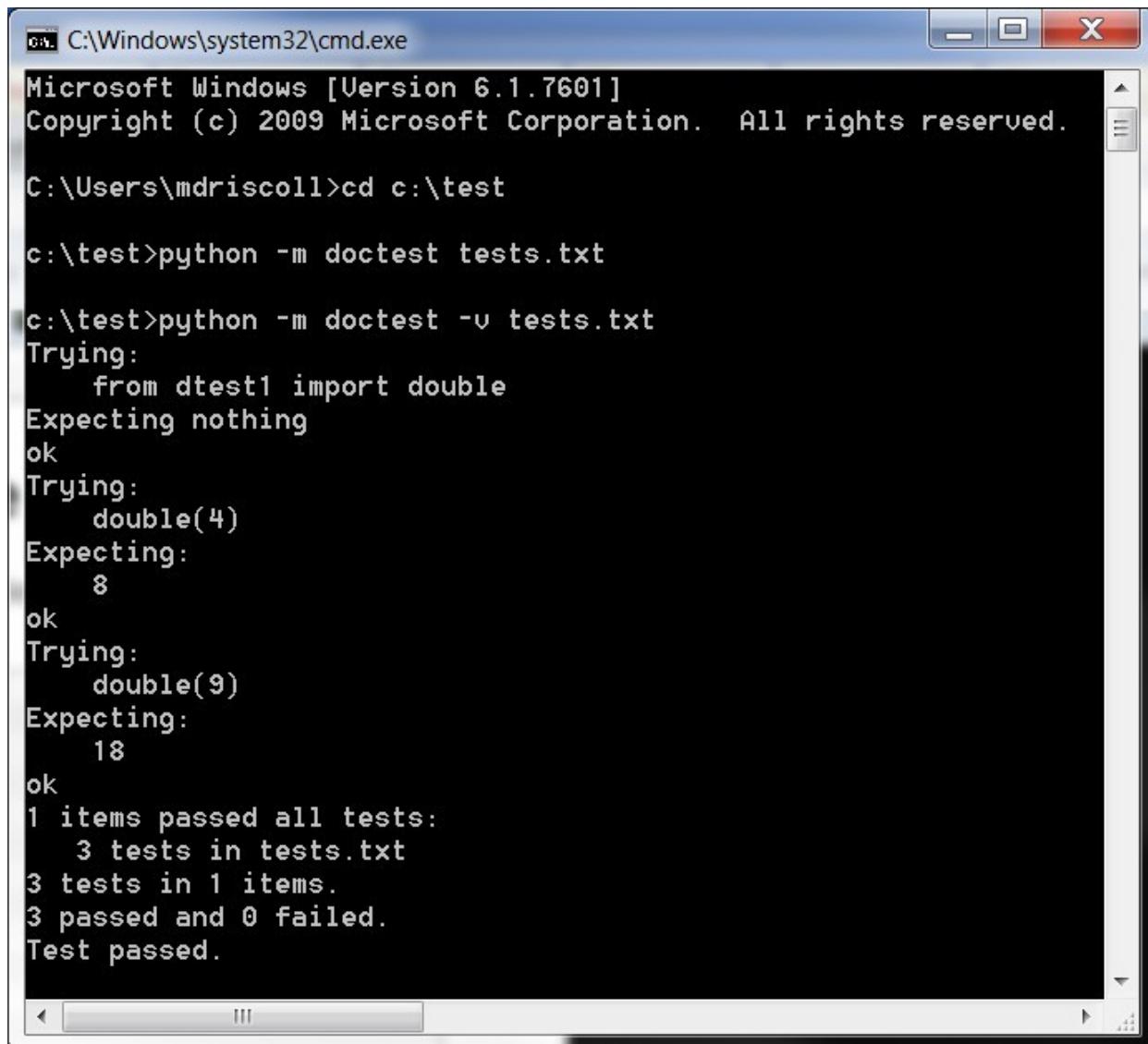
Now we're ready to learn how to put the tests into a separate file.

Running doctest From a Separate File

The `doctest` module also supports putting the testing into a separate file. This allows us to separate the tests from the code. Let's strip the tests from the previous example and put them into a text file named `tests.txt`:

```
1 The following are tests for dtest2.py
2
3 >>> from dtest2 import double
4 >>> double(4)
5 8
6 >>> double(9)
7 18
```

Let's run this test file on the command line. Here's how:



The screenshot shows a Windows command prompt window titled "C:\Windows\system32\cmd.exe". The window displays the following text output:

```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

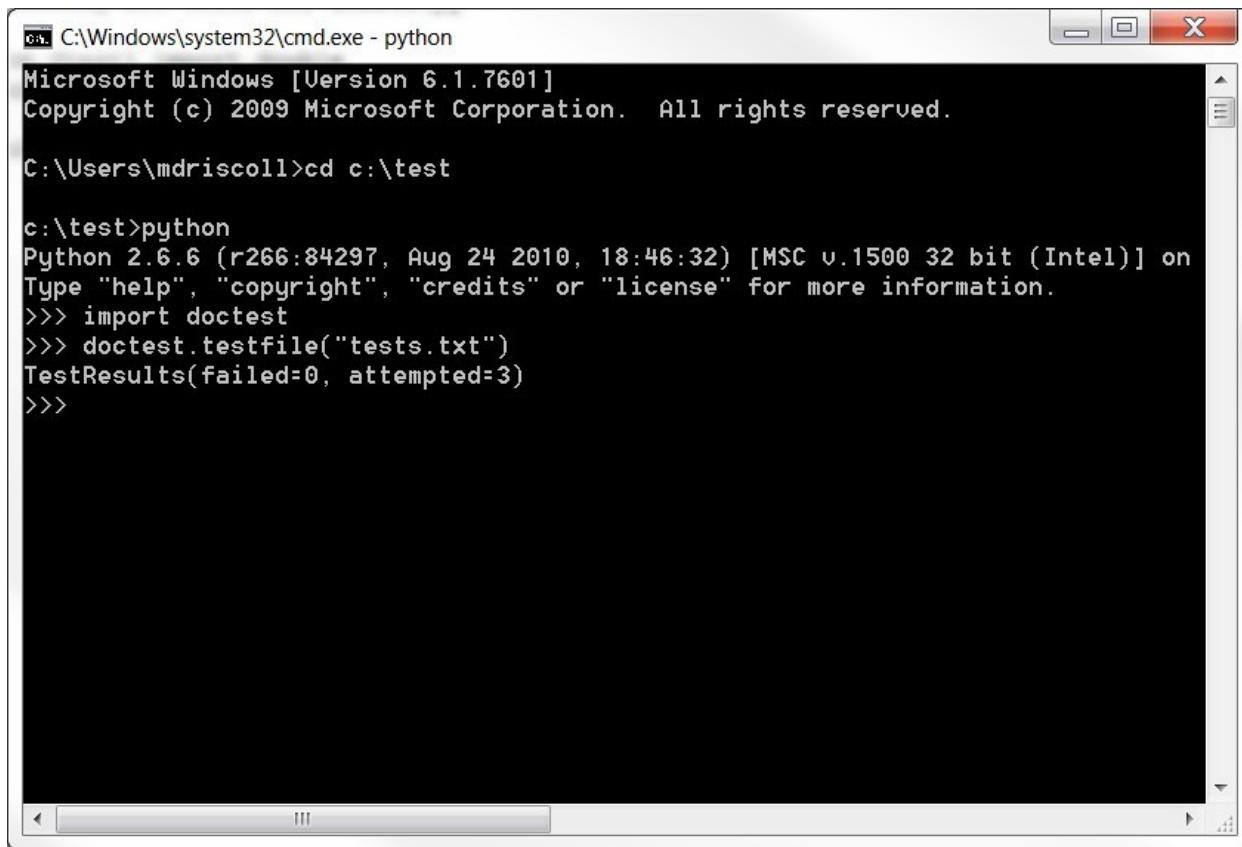
C:\Users\mdriscoll>cd c:\test

c:\test>python -m doctest tests.txt

c:\test>python -m doctest -v tests.txt
Trying:
    from dtest1 import double
Expecting nothing
ok
Trying:
    double(4)
Expecting:
    8
ok
Trying:
    double(9)
Expecting:
    18
ok
1 items passed all tests:
    3 tests in tests.txt
3 tests in 1 items.
3 passed and 0 failed.
Test passed.
```

image

You will notice that the syntax for calling doctest with a text file is the same as calling it with a Python file. The results are the same as well. In this case, there are three tests instead of two because we're also importing a module. You can also run the tests that are in a text file inside the Python interpreter. Here's one example:



```
C:\Windows\system32\cmd.exe - python
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mdriscoll>cd c:\test

c:\test>python
Python 2.6.6 (r266:84297, Aug 24 2010, 18:46:32) [MSC v.1500 32 bit (Intel)] on
Type "help", "copyright", "credits" or "license" for more information.
>>> import doctest
>>> doctest.testfile("tests.txt")
TestResults(failed=0, attempted=3)
>>>
```

image

Here we just import `doctest` and call its `testfile` method. Note that you need to also pass the filename or path to the `testfile` function. It will return a `TestResults` object that contains how many tests were attempted and how many failed.

Test Driven Development with `unittest`

In this section, you will learn about Test Driven Development (TDD) with Python using Python's built-in `unittest` module. I do want to thank Matt and Aaron for their assistance in showing me how TDD works in the real world. To demonstrate the concepts of TDD, we'll be covering how to score bowling in Python. You may want to use Google to look up the rules for bowling if you don't already know them. Once you know the rules, it's time to write some tests. In case you didn't know, the idea behind Test Driven Development is that you write the tests BEFORE you write the actual code. In this chapter, we will write a test, then some code to pass the test. We will iterate back and forth between writing tests and code until we're done. For this chapter, we'll write just three tests. Let's get started!

The First Test

Our first test will be to test our game object and see if it can calculate the correct total if we roll eleven times and only knock over one pin each time. This should give us a total of eleven.

```
1 import unittest
2
3 class TestBowling(unittest.TestCase):
4     """
5
6     def test_all_ones(self):
7         """Constructor"""
8         game = Game()
9         game.roll(11, 1)
10        self.assertEqual(game.score, 11)
```

This is a pretty simple test. We create a game object and then call its `roll` method eleven times with a score of one each time. Then we use the `assertEqual` method from the `unittest` module to test if the game object's score is correct (i.e. eleven). The next step is to write the simplest code you can think of to make the test pass. Here's one example:

```
1 class Game:
2     """
3
4     def __init__(self):
5         """Constructor"""
6         self.score = 0
7
8     def roll(self, numOfRolls, pins):
9         """
10        for roll in numOfRolls:
11            self.score += pins
```

For simplicity's sake, you can just copy and paste that into the same file with your test. We'll break them into two files for our next test. Anyway, as you can see, our `Game` class is super simple. All that was needed to pass the test was a `score` property and a `roll` method that can update it.

Let's run the test and see if it passes! The easiest way to run the tests is to add the following two lines of code to the bottom of the file:

```
1 if __name__ == '__main__':
2     unittest.main()
```

Then just run the Python file via the command line. If you do, you should get something like the following:

```
1 E
2 -----
3 ERROR: test_all_ones (__main__.TestBowling)
4 Constructor
5 -----
6 Traceback (most recent call last):
7   File "C:\Users\Mike\Documents\Scripts\Testing\bowling\test_one.py",
8   line 27, in test_all_ones
9     game.roll(11, 1)
10    File "C:\Users\Mike\Documents\Scripts\Testing\bowling\test_one.py",
11    line 15, in roll
12      for roll in numOfRolls:
13 TypeError: 'int' object is not iterable
14
15 -----
16 Ran 1 test in 0.001s
17
18 FAILED (errors=1)
```

Oops! We've got a mistake in there somewhere. It looks like we're passing an Integer and then trying to iterate over it. That doesn't work! We need to change our Game object's roll method to the following to make it work:

```
1 def roll(self, numOfRolls, pins):
2     """
3     for roll in range(numOfRolls):
4         self.score += pins
```

If you run the test now, you should get the following:

```
1 .
2 -----
3 Ran 1 test in 0.000s
4
5 OK
```

Note the “.” because it’s important. That little dot means that one test has run and that it passed. The “OK” at the end clues you into that fact as well. If you study the original output, you’ll notice it leads off with an “E” for error and there’s no dot! Let’s move on to test #2.

The Second Test

For the second test, we’ll test what happens when we get a strike. We’ll need to change the first test to use a list for the number of pins knocked down in each frame though, so we’ll look at both tests here. You’ll probably find this to be a fairly common process where you may need to edit a couple of tests due to fundamental changes in what you’re testing for. Normally this will only happen at the beginning of your coding and you will get better later on such that you shouldn’t need to do this. Since this is my first time doing this, I wasn’t thinking far enough ahead. Anyway, let’s take a look at the code:

```
1 from game import Game
2 import unittest
3
4 class TestBowling(unittest.TestCase):
5     """
6
7     def test_all_ones(self):
8         """Constructor"""
9         game = Game()
10        pins = [1 for i in range(11)]
11        game.roll(11, pins)
12        self.assertEqual(game.score, 11)
13
14    def test_strike(self):
15        """
16            A strike is 10 + the value of the next two rolls. So in this case
17            the first frame will be 10+5+4 or 19 and the second will be
18            5+4. The total score would be 19+9 or 28.
19        """
20        game = Game()
21        game.roll(11, [10, 5, 4])
```

```

22     self.assertEqual(game.score, 28)
23
24 if __name__ == '__main__':
25     unittest.main()

```

Let's take a look at our first test and how it changed. Yes, we're breaking the rules here a bit when it comes to TDD. Feel free to NOT change the first test and see what breaks. In the `test_all_ones` method, we set the `pins` variable to equal a list comprehension which created a list of eleven ones. Then we passed that to our `game` object's `roll` method along with the number of rolls.

In the second test, we roll a strike in our first roll, a five in our second and a four in our third. Note that we went a head and told it that we were passing in eleven rolls and yet we only pass in three. This means that we need to set the other eight rolls to zeros. Next, we use our trusty `assertEqual` method to check if we get the right total. Finally, note that we're now importing the `Game` class rather than keeping it with the tests. Now we need to implement the code necessary to pass these two tests. Let's take a look at one possible solution:

```

1 class Game:
2     """
3
4     def __init__(self):
5         """Constructor"""
6         self.score = 0
7         self.pins = [0 for i in range(11)]
8
9     def roll(self, numOfRolls, pins):
10        """
11        x = 0
12        for pin in pins:
13            self.pins[x] = pin
14            x += 1
15        x = 0
16        for roll in range(numOfRolls):
17            if self.pins[x] == 10:
18                self.score = self.pins[x] + self.pins[x+1] + self.pins[x+2]
19            else:
20                self.score += self.pins[x]
21            x += 1
22        print(self.score)

```

Right off the bat, you will notice that we have a new class attribute called `self.pins` that holds the default pin list, which is eleven zeroes. Then in our `roll` method, we add the correct scores to the correct position in the `self.pins` list in the first loop. Then in the second loop, we check to see if the

pins knocked down equals ten. If it does, we add it and the next two scores to score. Otherwise, we do what we did before. At the end of the method, we print out the score to check if it's what we expect. At this point, we're ready to code up our final test.

The Third (and Final) Test

Our last test will test for the correct score that would result should someone roll a spare. The test is easy, the solution is slightly more difficult. While we're at it, we're going to refactor the test code a bit. As usual, we will look at the test first.

```
1 from game_v2 import Game
2 import unittest
3
4 class TestBowling(unittest.TestCase):
5     """
6
7     def setUp(self):
8         """
9         self.game = Game()
10
11    def test_all_ones(self):
12        """
13            If you don't get a strike or a spare, then you just add up the
14            face value of the frame. In this case, each frame is worth
15            one point, so the total is eleven.
16        """
17        pins = [1 for i in range(11)]
18        self.game.roll(11, pins)
19        self.assertEqual(self.game.score, 11)
20
21    def test_spare(self):
22        """
23            A spare is worth 10, plus the value of your next roll. So in this
24            case, the first frame will be 5+5+5 or 15 and the second will be
25            5+4 or 9. The total is 15+9, which equals 24,
26        """
27        self.game.roll(11, [5, 5, 5, 4])
28        self.assertEqual(self.game.score, 24)
29
30    def test_strike(self):
31        """
```

```

32     A strike is 10 + the value of the next two rolls. So in this case
33     the first frame will be 10+5+4 or 19 and the second will be
34     5+4. The total score would be 19+9 or 28.
35     """
36     self.game.roll(11, [10, 5, 4])
37     self.assertEqual(self.game.score, 28)
38
39 if __name__ == '__main__':
40     unittest.main()

```

First off, we added a `setUp` method that will create a `self.game` object for us for each test. If we were accessing a database or something like that, we would probably have a tear down method too for closing connections or files or that sort of thing. These are run at the beginning and end of each test respectively should they exist. The `test_all_ones` and `test_strike` tests are basically the same except that they are using “`self.game`” now. The only the new test is `test_spare`. The docstring explains how spares work and the code is just two lines. Yes, you can figure this out. Let’s look at the code we’ll need to pass these tests:

```

1 # game_v2.py
2
3 class Game:
4     """
5
6     def __init__(self):
7         """Constructor"""
8         self.score = 0
9         self.pins = [0 for i in range(11)]
10
11    def roll(self, numOfRolls, pins):
12        """
13        x = 0
14        for pin in pins:
15            self.pins[x] = pin
16            x += 1
17        x = 0
18        spare_begin = 0
19        spare_end = 2
20        for roll in range(numOfRolls):
21            spare = sum(self.pins[spare_begin:spare_end])
22            if self.pins[x] == 10:
23                self.score = self.pins[x] + self.pins[x+1] + self.pins[x+2]
24            elif spare == 10:

```

```
25         self.score = spare + self.pins[x+2]
26         x += 1
27     else:
28         self.score += self.pins[x]
29         x += 1
30     if x == 11:
31         break
32     spare_begin += 2
33     spare_end += 2
34 print(self.score)
```

For this part of the puzzle, we add to our conditional statement in our loop. To calculate the spare's value, we use the `spare_begin` and `spare_end` list positions to get the right values from our list and then we sum them up. That's what the `spare` variable is for. That may be better placed in the `elif`, but I'll leave that for the reader to experiment with. Technically, that's just the first half of the spare score. The second half are the next two rolls, which is what you'll find in the calculation in the `elif` portion of the current code. The rest of the code is the same.

Other Notes

As you may have guessed, there is a whole lot more to the `unittest` module than what is covered here. There are lots of other asserts you can use to test results with. You can skip tests, run the tests from the command line, use a `TestLoader` to create a test suite and much, much more. Be sure to read the full [documentation](#)¹⁴ when you get the chance as this tutorial barely scratches the surface of this library.

Wrapping Up

At this point, you should be able to understand how to use the `doctest` and `unittest` modules effectively in your own code. You should go and read the Python documentation about these two modules as there is additional information about other options and functionality that you might find useful. You also know a little about how to use the concepts of Test Driven Development when writing your own scripts.

¹⁴<http://docs.python.org/2/library/unittest.html>

Part IV - Tips, Tricks and Tutorials

In Part IV, you will learn how to install 3rd party packages from the Python Package Index (PyPI). You will learn a bit about `easy_install`, `pip` and `setup.py` and how to use these tools to install the packages. This is just the first chapter though. Here's a listing of the packages you will learn about:

- `configobj` - working with Config files in a more “Pythonic” way.
- `lxml` - a package for working with XML
- `pylint` / `pyflakes` - Python code analyzers
- `requests` - a Python-friendly version of `urllib`
- `SQLAlchemy` - an Object Relational Mapper for Python
- `virtualenv` - learn about virtual environments in Python

The reason we will be looking at `configobj` is because I think it works better than `ConfigParser`, the module that comes with Python. The `configobj` package has an interface that is just more intuitive and powerful than `ConfigParser`. In the next chapter, we'll look at the `lxml` module and learn a couple of new ways to read, parse and create XML. In the fourth chapter, we will look at `pylint` and `pyflakes`, which are great for code analysis. They can look at your module and check for errors. `pylint` can also be used to help you get your code to conform to PEP8, the Python style guide.

The `requests` package is great replacement for the `urllib` module. Its interface is simpler and the documentation is quite good. `SQLAlchemy` is the premier Object Relational Mapper for Python. It allows you to write SQL queries, tables, etc in Python code. One of its best features is that if you need to switch database backends, you won't have to change your code very much to continue working with said database. For the last chapter, we'll look at `virtualenv`, a neat module that allows you to create mini-virtual environments that you can write your code in. These virtual environments are especially handy for testing out new modules or just new releases of modules before you apply them to your Python core installation.



image

Chapter 29 - Installing Modules

When you're first starting out as a Python programmer, you don't think about how you might need to install an external package or module. But when that need appears, you'll want to know how to in a hurry! Python packages can be found all over the internet. Most of the popular ones can be found on the Python Package Index (PyPI). You will also find a lot of Python packages on github, bitbucket, and Google code. In this article, we will be covering the following methods of installing Python packages:

- Install from source
- easy_install
- pip
- Other ways to install packages

Installing from Source

Installing from source is a great skill to have. There are easier ways, which we'll be getting to later on in the article. However, there are some packages that you have to install from source. For example, to use `easy_install`, you will need to first install `setuptools`. To do that, you will want to download the tar or zip file from the Python Package Index and extract it somewhere on your system. Then look for the `setup.py` file. Open up a terminal session and change directories to the folder that contains the setup file. Then run the following command:

```
1 python setup.py install
```

If Python isn't on your system path, you will receive an error message stating that the `python` command wasn't found or is an unknown application. You can call this command by using the full path to Python instead. Here's how you might do it if you were on Windows:

```
1 c:\python34\python.exe setup.py install
```

This method is especially handy if you have multiple versions of Python installed and you need to install the package to different ones. All you need to do is type the full path to the right Python version and install the package against it.

Some packages contain C code, such as C header files that will need to be compiled for the package to install correctly. On Linux, you will normally already have a C/C++ compiler installed and you can get the package installed with minimal headaches. On Windows, you will need to have the correct version of Visual Studio installed to compile the package correctly. Some people say you can use MingW too, but I have yet to find a way to make that work. If the package has a Windows installer already pre-made, use it. Then you don't have to mess with compiling at all.

Using easy_install

Once you have `setuptools` installed, you can use `easy_install`. You can find it installed in your Python installation's `Scripts` folder. Be sure to add the `Scripts` folder to your system path so you can call `easy_install` on the command line without specifying its full path. Try running the following command to learn about all of `easy_install`'s options:

```
1 easy_install -h
```

When you want to install a package with `easy_install`, all you have to do is this:

```
1 easy_install package_name
```

`easy_install` will attempt to download the package from PyPI, compile it (if necessary) and install it. If you go into your Python's `site-packages` directory, you will find a file named `easy-install.pth` that will contain an entry for all packages installed with `easy_install`. This file is used by Python to help in importing the module or package.

You can also tell `easy_install` to install from a URL or from a path on your computer. It can also install packages directly from a tar file. You can use `easy_install` to upgrade a package by using `-upgrade` (or `-U`). Finally, you can use `easy_install` to install Python eggs. You can find egg files on PyPI and other locations. An egg is basically a special zip file. In fact, if you change the extension to `.zip`, you can unzip the egg file.

Here are some examples:

```
1 easy_install -U SQLAlchemy
2 easy_install http://example.com/path/to/MyPackage-1.2.3.tgz
3 easy_install /path/to/downloaded/package
```

There are some issues with `easy_install`. It will try to install a package before it's finished downloading. There is no way to uninstall a package using `easy_install`. You will have to delete the package yourself and update the `easy-install.pth` file by removing the entry to the package. For these reasons and others, there was movement in the Python community to create something different, which caused `pip` to be born.

Using pip

The `pip` program actually comes with Python 3.4. If you have an older version of Python, then you will need to install `pip` manually. Installing `pip` is a little bit different than what we have previously discussed. You still go to PyPI, but instead of downloading the package and running its `setup.py` script, you will be asked to download a single script called `get-pip.py`. Then you will need to execute it by doing the following:

```
1 python get-pip.py
```

This will install **setuptools** or an alternative to setuptools called **distribute** if one of them is not already installed. It will also install pip. pip works with CPython versions 2.6, 2.7, 3.1, 3.2, 3.3, 3.4 and also pypy. You can use pip to install anything that easy_install can install, but the invocation is a bit different. To install a package, do the following:

```
1 pip install package_name
```

To upgrade a package, you would do this:

```
1 pip install -U PackageName
```

You may want to call **pip -h** to get a full listing of everything pip can do. One thing that pip can install that easy_install cannot is the Python **wheel** format. A wheel is a ZIP-format archive with a specially formatted filename and the **.whl** extension. You can also install wheels via its own command line utility. On the other hand, pip cannot install an egg. If you need to install an egg, you will want to use easy_install.

A Note on Dependencies

One of the many benefits of using **easy_install** and **pip** is that if the package has dependencies specified in their **setup.py** script, both **easy_install** and **pip** will attempt to download and install them too. This can alleviate a lot of frustration when you're trying new packages and you didn't realize that package A depended on Packages B, C and D. With **easy_install** or **pip**, you don't need to worry about that any more.

Wrapping Up

At this point, you should be able to install just about any package that you need, assuming the package supports your version of Python. There are a lot of tools available to the Python programmer. While packaging in Python is a bit confusing right now, once you know how to use the proper tools, you can usually get what you want installed or packaged up. We will be looking more at creating our own packages, eggs and wheels in Part V.

Chapter 30 - ConfigObj

Python comes with a handy module called `ConfigParser`. It's good for creating and reading configuration files (aka INI files). However, Michael Foord (author of IronPython in Action) and Nicola Larosa decided to write their own configuration module called `ConfigObj`. In many ways, it is an improvement over the standard library's module. For example, it will return a dictionary-like object when it reads a config file. `ConfigObj` can also understand some Python types. Another neat feature is that you can create a configuration spec that `ConfigObj` will use to validate the config file.

Getting Started

First of all, you need to go and get `ConfigObj`. This is a good time to use your knowledge from the last chapter on installing packages. Here is how you would get `ConfigObj` with pip:

```
1 pip install configobj
```

Once you have it installed, we can move on. To start off, open a text editor and create a file with some contents like this:

```
1 product = Sony PS3
2 accessories = controller, eye, memory stick
3 # This is a comment that will be ignored
4 retail_price = $400
```

Save it where ever you like. I'm going to call mine `config.ini`. Now let's see how `ConfigObj` can be used to extract that information:

```
1 >>> from configobj import ConfigObj
2 >>> config = ConfigObj(r"path to config.ini")
3 >>> config["product"]
4 'Sony PS3'
5 >>> config["accessories"]
6 ['controller', 'eye', 'memory stick']
7 >>> type(config["accessories"])
8 <type 'list'>
```

As you can see, ConfigObj uses Python's `dict` API to access the information it has extracted. All you had to do to get ConfigObj to parse the file was to pass the file's path to ConfigObj. Now, if the information had been under a section (i.e. [Sony]), then you would have had to do pre-pend everything with ["Sony"], like this: `config["Sony"]["product"]`. Also take note that the `accessories` section was returned as a list of strings. ConfigObj will take any valid line with a comma-separated list and return it as a Python list. You can also create multi-line strings in the config file as long as you enclose them with triple single or double quotes.

If you need to create a sub-section in the file, then use extra square brackets. For example, [Sony] is the top section, [[Playstation]] is the sub-section and [[[PS3]]] is the sub-section of the sub-section. You can create sub-sections up to any depth. For more information on the formatting of the file, I recommend reading ConfigObj's documentation.

Now we'll do the reverse and create the config file programmatically.

```
1 import configobj
2
3 def createConfig(path):
4     config = configobj.ConfigObj()
5     config.filename = path
6     config["Sony"] = {}
7     config["Sony"]["product"] = "Sony PS3"
8     config["Sony"]["accessories"] = ['controller', 'eye', 'memory stick']
9     config["Sony"]["retail price"] = "$400"
10    config.write()
11
12 if __name__ == "__main__":
13     createConfig("config.ini")
```

As you can see, all it took was 13 lines of code. In the code above, we create a function and pass it the path for our config file. Then we create a ConfigObj object and set its filename property. To create the section, we create an empty dict with the name "Sony". Then we pre-pend each line of the sections contents in the same way. Finally, we call our config object's write method to write the data to the file.

Using a configspec

ConfigObj also provides a way to validate your configuration files using a `configspec`. When I mentioned that I was going to write on this topic, Steven Sproat (creator of Whyteboard) volunteered his configspec code as an example. I took his specification and used it to create a default config file. In this example, we use Foord's validate module to do the validation. I don't think it's included in your ConfigObj download, so you may need to download it as well. Now, let's take a look at the code:

```
1 import configobj, validate
2
3 cfg = """
4 bmp_select_transparent = boolean(default=False)
5 canvas_border = integer(min=10, max=35, default=15)
6 colour1 = list(min=3, max=3, default=list('280', '0', '0'))
7 colour2 = list(min=3, max=3, default=list('255', '255', '0'))
8 colour3 = list(min=3, max=3, default=list('0', '255', '0'))
9 colour4 = list(min=3, max=3, default=list('255', '0', '0'))
10 colour5 = list(min=3, max=3, default=list('0', '0', '255'))
11 colour6 = list(min=3, max=3, default=list('160', '32', '240'))
12 colour7 = list(min=3, max=3, default=list('0', '255', '255'))
13 colour8 = list(min=3, max=3, default=list('255', '165', '0'))
14 colour9 = list(min=3, max=3, default=list('211', '211', '211'))
15 convert_quality = option('highest', 'high', 'normal', default='normal')
16 default_font = string
17 default_width = integer(min=1, max=12000, default=640)
18 default_height = integer(min=1, max=12000, default=480)
19 imagemagick_path = string
20 handle_size = integer(min=3, max=15, default=6)
21 language = option('English', 'English (United Kingdom)', 'Russian',
22                 'Hindi', default='English')
23 print_title = boolean(default=True)
24 statusbar = boolean(default=True)
25 toolbar = boolean(default=True)
26 toolbox = option('icon', 'text', default='icon')
27 undo_sheets = integer(min=5, max=50, default=10)
28 """
29
30 def createConfig(path):
31     """
32     Create a config file using a configspec
33     and validate it against a Validator object
34     """
35     spec = cfg.split("\n")
36     config = configobj.ConfigObj(path, configspec=spec)
37     validator = validate.Validator()
38     config.validate(validator, copy=True)
39     config.filename = path
40     config.write()
41
42 if __name__ == "__main__":
```

```
43     createConfig("config.ini")
```

The configspec allows the programmer the ability to specify what **types** are returned for each line in the configuration file. It also can be used to set a default value and a **min** and **max** values (among other things). If you run the code above, you will see a *config.ini* file generated in the current working directory that has just the default values. If the programmer didn't specify a default, then that line isn't even added to the configuration.

Let's take a closer look at what's going on just to make sure you understand. In the `createConfig` function, we create a `ConfigObj` instance by passing in the file path and setting the configspec. Note that the configspec can also be a normal text file or a python file rather than the string that is in this example. Next, we create a Validator object. Normal usage is to just call `config.validate(validation)`, but in this code I set the copy argument to True so that I could create a file. Otherwise, all it would do is validate that the file I passed in fit the configspec's rules. Finally I set the config's filename and write the data out.

Wrapping Up

Now you know just enough to get you started on the ins and outs of `ConfigObj`. I hope you'll find it as helpful as I have. Be sure to go to the module's documentation and read more about what it and `validate` can do.

Chapter 31 - Parsing XML with lxml

In Part I, we looked at some of Python's built-in XML parsers. In this chapter, we will look at the fun third-party package, `lxml` from codespeak. It uses the ElementTree API, among other things. The `lxml` package has XPath and XSLT support, includes an API for SAX and a C-level API for compatibility with C/Pyrex modules. Here is what we will cover:

- How to Parse XML with `lxml`
- A Refactoring example
- How to Parse XML with `lxml.objectify`
- How to Create XML with `lxml.objectify`

For this chapter, we will use the examples from the `minidom` parsing example and see how to parse those with `lxml`. Here's an XML example from a program that was written for keeping track of appointments:

```
1 <?xml version="1.0" ?>
2 <zAppointments reminder="15">
3   <appointment>
4     <begin>1181251680</begin>
5     <uid>040000008200E000</uid>
6     <alarmTime>1181572063</alarmTime>
7     <state></state>
8     <location></location>
9     <duration>1800</duration>
10    <subject>Bring pizza home</subject>
11  </appointment>
12  <appointment>
13    <begin>1234360800</begin>
14    <duration>1800</duration>
15    <subject>Check MS Office website for updates</subject>
16    <location></location>
17    <uid>604f4792-eb89-478b-a14f-dd34d3cc6c21-1234360800</uid>
18    <state>dismissed</state>
19  </appointment>
20 </zAppointments>
```

Let's learn how to parse this with `lxml`!

Parsing XML with lxml

The XML above shows two appointments. The beginning time is in seconds since the epoch; the uid is generated based on a hash of the beginning time and a key; the alarm time is the number of seconds since the epoch, but should be less than the beginning time; and the state is whether or not the appointment has been snoozed, dismissed or not. The rest of the XML is pretty self-explanatory. Now let's see how to parse it.

```

1  from lxml import etree
2
3  def parseXML(xmlFile):
4      """
5          Parse the xml
6      """
7      with open(xmlFile) as fobj:
8          xml = fobj.read()
9
10     root = etree.fromstring(xml)
11
12     for appt in root.getchildren():
13         for elem in appt.getchildren():
14             if not elem.text:
15                 text = "None"
16             else:
17                 text = elem.text
18             print(elem.tag + " => " + text)
19
20     if __name__ == "__main__":
21         parseXML("example.xml")

```

First off, we import the needed modules, namely the `etree` module from the `lxml` package and the `StringIO` function from the built-in `StringIO` module. Our `parseXML` function accepts one argument: the path to the XML file in question. We open the file, read it and close it. Now comes the fun part! We use `etree`'s `parse` function to parse the XML code that is returned from the `StringIO` module. For reasons I don't completely understand, the `parse` function requires a file-like object.

Anyway, next we iterate over the context (i.e. the `lxml.etree.iterparse` object) and extract the tag elements. We add the conditional if statement to replace the empty fields with the word "None" to make the output a little clearer. And that's it.

Parsing the Book Example

Well, the result of that example was kind of boring. Most of the time, you want to save the data you extract and do something with it, not just print it out to stdout. So for our next example, we'll create a data structure to contain the results. Our data structure for this example will be a list of dicts. We'll use the MSDN book example here from the earlier chapter again. Save the following XML as *example.xml*

```
1 <?xml version="1.0"?>
2 <catalog>
3   <book id="bk101">
4     <author>Gambardella, Matthew</author>
5     <title>XML Developer's Guide</title>
6     <genre>Computer</genre>
7     <price>44.95</price>
8     <publish_date>2000-10-01</publish_date>
9     <description>An in-depth look at creating applications
10    with XML.</description>
11  </book>
12  <book id="bk102">
13    <author>Ralls, Kim</author>
14    <title>Midnight Rain</title>
15    <genre>Fantasy</genre>
16    <price>5.95</price>
17    <publish_date>2000-12-16</publish_date>
18    <description>A former architect battles corporate zombies,
19    an evil sorceress, and her own childhood to become queen
20    of the world.</description>
21  </book>
22  <book id="bk103">
23    <author>Corets, Eva</author>
24    <title>Maeve Ascendant</title>
25    <genre>Fantasy</genre>
26    <price>5.95</price>
27    <publish_date>2000-11-17</publish_date>
28    <description>After the collapse of a nanotechnology
29    society in England, the young survivors lay the
30    foundation for a new society.</description>
31  </book>
32 </catalog>
```

Now let's parse this XML and put it in our data structure!

```

1 from lxml import etree
2
3 def parseBookXML(xmlFile):
4
5     with open(xmlFile) as fobj:
6         xml = fobj.read()
7
8     root = etree.fromstring(xml)
9
10    book_dict = {}
11    books = []
12    for book in root.getchildren():
13        for elem in book.getchildren():
14            if not elem.text:
15                text = "None"
16            else:
17                text = elem.text
18            print(elem.tag + " => " + text)
19            book_dict[elem.tag] = text
20        if book.tag == "book":
21            books.append(book_dict)
22            book_dict = {}
23    return books
24
25 if __name__ == "__main__":
26     parseBookXML("books.xml")

```

This example is pretty similar to our last one, so we'll just focus on the differences present here. Right before we start iterating over the context, we create an empty dictionary object and an empty list. Then inside the loop, we create our dictionary like this:

```
1 book_dict[elem.tag] = text
```

The text is either `elem.text` or `None`. Finally, if the tag happens to be `book`, then we're at the end of a book section and need to add the dict to our list as well as reset the dict for the next book. As you can see, that is exactly what we have done. A more realistic example would be to put the extracted data into a `Book` class. I have done the latter with json feeds before.

Now we're ready to learn how to parse XML with `lxml.objectify`!

Parsing XML with `lxml.objectify`

The `lxml` module has a module called `objectify` that can turn XML documents into Python objects. I find “objectified” XML documents very easy to work with and I hope you will too. You may need

to jump through a hoop or two to install it as `pip` doesn't work with lxml on Windows. Be sure to go to the Python Package index and look for a version that's been made for your version of Python. Also note that the latest pre-built installer for lxml only supports Python 3.2 (at the time of writing), so if you have a newer version of Python, you may have some difficulty getting lxml installed for your version.

Anyway, once you have it installed, we can start going over this wonderful piece of XML again:

```
1 <?xml version="1.0" ?>
2 <zAppointments reminder="15">
3   <appointment>
4     <begin>1181251680</begin>
5     <uid>040000008200E000</uid>
6     <alarmTime>1181572063</alarmTime>
7     <state></state>
8     <location></location>
9     <duration>1800</duration>
10    <subject>Bring pizza home</subject>
11  </appointment>
12  <appointment>
13    <begin>1234360800</begin>
14    <duration>1800</duration>
15    <subject>Check MS Office website for updates</subject>
16    <location></location>
17    <uid>604f4792-eb89-478b-a14f-dd34d3cc6c21-1234360800</uid>
18    <state>dismissed</state>
19  </appointment>
20 </zAppointments>
```

Now we need to write some code that can parse and modify the XML. Let's take a look at this little demo that shows a bunch of the neat abilities that objectify provides.

```
1 from lxml import etree, objectify
2
3 def parseXML(xmlFile):
4     """Parse the XML file"""
5     with open(xmlFile) as f:
6         xml = f.read()
7
8     root = objectify.fromstring(xml)
9
10    # returns attributes in element node as dict
11    attrib = root.attrib
```

```

12
13     # how to extract element data
14     begin = root.appointment.begin
15     uid = root.appointment.uid
16
17     # loop over elements and print their tags and text
18     for appt in root.getchildren():
19         for e in appt.getchildren():
20             print("%s => %s" % (e.tag, e.text))
21     print()
22
23     # how to change an element's text
24     root.appointment.begin = "something else"
25     print(root.appointment.begin)
26
27     # how to add a new element
28     root.appointment.new_element = "new data"
29
30     # remove the py:pytype stuff
31     objectify.deannotate(root)
32     etree.cleanup_namespaces(root)
33     obj_xml = etree.tostring(root, pretty_print=True)
34     print(obj_xml)
35
36     # save your xml
37     with open("new.xml", "w") as f:
38         f.write(obj_xml)
39
40 if __name__ == "__main__":
41     f = r'path\to\sample.xml'
42     parseXML(f)

```

The code is pretty well commented, but we'll spend a little time going over it anyway. First we pass it our sample XML file and **objectify** it. If you want to get access to a tag's attributes, use the **attrib** property. It will return a dictionary of the attributes of the tag. To get to sub-tag elements, you just use dot notation. As you can see, to get to the **begin** tag's value, we can just do something like this:

```
1 begin = root.appointment.begin
```

One thing to be aware of is if the value happens to have leading zeroes, the returned value may have them truncated. If that is important to you, then you should use the following syntax instead:

```
1 begin = root.appointment.begin.text
```

If you need to iterate over the children elements, you can use the `iterchildren` method. You may have to use a nested for loop structure to get everything. Changing an element's value is as simple as just assigning it a new value.

```
1 root.appointment.new_element = "new data"
```

Now we're ready to learn how to create XML using `lxml.objectify`.

Creating XML with `lxml.objectify`

The `lxml.objectify` sub-package is extremely handy for parsing and creating XML. In this section, we will show how to create XML using the `lxml.objectify` module. We'll start with some simple XML and then try to replicate it. Let's get started!

We will continue using the following XML for our example:

```
1 <?xml version="1.0" ?>
2 <zAppointments reminder="15">
3   <appointment>
4     <begin>1181251680</begin>
5     <uid>040000008200E000</uid>
6     <alarmTime>1181572063</alarmTime>
7     <state></state>
8     <location></location>
9     <duration>1800</duration>
10    <subject>Bring pizza home</subject>
11  </appointment>
12  <appointment>
13    <begin>1234360800</begin>
14    <duration>1800</duration>
15    <subject>Check MS Office website for updates</subject>
16    <location></location>
17    <uid>604f4792-eb89-478b-a14f-dd34d3cc6c21-1234360800</uid>
18    <state>dismissed</state>
19  </appointment>
20 </zAppointments>
```

Let's see how we can use `lxml.objectify` to recreate this XML:

```
1 from lxml import etree, objectify
2
3 def create_appt(data):
4     """
5     Create an appointment XML element
6     """
7     appt = objectify.Element("appointment")
8     appt.begin = data["begin"]
9     appt.uid = data["uid"]
10    appt.alarmTime = data["alarmTime"]
11    appt.state = data["state"]
12    appt.location = data["location"]
13    appt.duration = data["duration"]
14    appt.subject = data["subject"]
15    return appt
16
17 def create_xml():
18     """
19     Create an XML file
20     """
21
22     xml = '''<?xml version="1.0" encoding="UTF-8"?>
23     <zAppointments>
24     </zAppointments>
25     '''
26
27     root = objectify.fromstring(xml)
28     root.set("reminder", "15")
29
30     appt = create_appt({"begin":1181251680,
31                         "uid":"040000008200E000",
32                         "alarmTime":1181572063,
33                         "state":"",
34                         "location":"",
35                         "duration":1800,
36                         "subject":"Bring pizza home"})
37
38     root.append(appt)
39
40     uid = "604f4792-eb89-478b-a14f-dd34d3cc6c21-1234360800"
41     appt = create_appt({"begin":1234360800,
42                         "uid":uid,
```

```
43             "alarmTime":1181572063,
44             "state":"dismissed",
45             "location":"",
46             "duration":1800,
47             "subject":"Check MS Office website for updates"}
48         )
49     root.append(appt)
50
51     # remove lxml annotation
52     objectify.deannotate(root)
53     etree.cleanup_namespaces(root)
54
55     # create the xml string
56     obj_xml = etree.tostring(root,
57                               pretty_print=True,
58                               xml_declaration=True)
59
60     try:
61         with open("example.xml", "wb") as xml_writer:
62             xml_writer.write(obj_xml)
63     except IOError:
64         pass
65
66 if __name__ == "__main__":
67     create_xml()
```

Let's break this down a bit. We will start with the `create_xml` function. In it we create an XML root object using the `objectify` module's `fromstring` function. The root object will contain `zAppointment` as its tag. We set the root's `reminder` attribute and then we call our `create_appt` function using a dictionary for its argument. In the `create_appt` function, we create an instance of an Element (technically, it's an `ObjectifiedElement`) that we assign to our `appt` variable. Here we use `dot-notation` to create the tags for this element. Finally we return the `appt` element back and append it to our `root` object. We repeat the process for the second appointment instance.

The next section of the `create_xml` function will remove the `lxml` annotation. If you do not do this, your XML will end up looking like the following:

```

1 <?xml version="1.0" ?>
2 <zAppointments py:pytype="TREE" reminder="15">
3   <appointment py:pytype="TREE">
4     <begin py:pytype="int">1181251680</begin>
5     <uid py:pytype="str">040000008200E000</uid>
6     <alarmTime py:pytype="int">1181572063</alarmTime>
7     <state py:pytype="str"/>
8     <location py:pytype="str"/>
9     <duration py:pytype="int">1800</duration>
10    <subject py:pytype="str">Bring pizza home</subject>
11  </appointment><appointment py:pytype="TREE">
12    <begin py:pytype="int">1234360800</begin>
13    <uid py:pytype="str">604f4792-eb89-478b-a14f-dd34d3cc6c21-1234360800</ui \
14 d>
15   <alarmTime py:pytype="int">1181572063</alarmTime>
16   <state py:pytype="str">dismissed</state>
17   <location py:pytype="str"/>
18   <duration py:pytype="int">1800</duration>
19   <subject py:pytype="str">Check MS Office website for updates</subject>
20 </appointment>
21 </zAppointments>

```

To remove all that unwanted annotation, we call the following two functions:

```

1 objectify.deannotate(root)
2 etree.cleanup_namespaces(root)

```

The last piece of the puzzle is to get lxml to generate the XML itself. Here we use lxml's **etree** module to do the hard work:

```

1 obj_xml = etree.tostring(root,
2                           pretty_print=True,
3                           xml_declaration=True)

```

The `tostring` function will return a nice string of the XML and if you set `pretty_print` to True, it will usually return the XML in a nice format too. The `xml_declaration` keyword argument tells the `etree` module whether or not to include the first declaration line (i.e. `<?xml version="1.0" ?>`).

Wrapping Up

Now you know how to use lxml's `etree` and `objectify` modules to parse XML. You also know how to use `objectify` to create XML. Knowing how to use more than one module to accomplish the same task can be valuable in seeing how to approach the same problem from different angles. It will also help you choose the tool that you're most comfortable with.

Chapter 32 - Python Code Analysis

Python code analysis can be a heavy subject, but it can be very helpful in making your programs better. There are several Python code analyzers that you can use to check your code and see if they conform to standards. `pylint` is probably the most popular. It's very configurable, customizable and pluggable too. It also checks your code to see if it conforms to PEP8, the official style guide of Python Core and it looks for programming errors too.

Note that `pylint` checks your code against most, but not all of PEP8's standards. We will spend a little time learning about another code analysis package that is called `pyflakes`.

Getting Started with `pylint`

The `pylint` package isn't included with Python, so you will need to go to the Python Package Index (PyPI) or the package's website to download it. You can use the following command to do all the work for you:

```
1 pip install pylint
```

If everything goes as planned, you should now have `pylint` installed and we'll be ready to continue.

Analyzing Your Code

Once `pylint` is installed, you can run it on the command line without any arguments to see what options it accepts. If that doesn't work, you can type out the full path like this:

```
1 c:\Python34\Scripts\pylint
```

Now we need some code to analyze. Here's a piece of code that has four errors in it. Save this to a file named `crummy_code.py`:

```
1 import sys
2
3 class CarClass:
4     """
5
6     def __init__(self, color, make, model, year):
7         """Constructor"""
8         self.color = color
9         self.make = make
10        self.model = model
11        self.year = year
12
13        if "Windows" in platform.platform():
14            print("You're using Windows!")
15
16        self.weight = self.getWeight(1, 2, 3)
17
18    def getWeight(this):
19        """
20        return "2000 lbs"
```

Can you spot the errors without running the code? Let's see if pylint can find the issues!

```
1 pylint crummy_code.py
```

When you run this command, you will see a lot of output sent to your screen. Here's a partial example:

```
1 c:\py101>c:\Python34\Scripts\pylint crummy_code.py
2 No config file found, using default configuration
3 **** Module crummy_code
4 C: 2, 0: Trailing whitespace (trailing-whitespace)
5 C: 5, 0: Trailing whitespace (trailing-whitespace)
6 C: 12, 0: Trailing whitespace (trailing-whitespace)
7 C: 15, 0: Trailing whitespace (trailing-whitespace)
8 C: 17, 0: Trailing whitespace (trailing-whitespace)
9 C: 1, 0: Missing module docstring (missing-docstring)
10 C: 3, 0: Empty class docstring (empty-docstring)
11 C: 3, 0: Old-style class defined. (old-style-class)
12 E: 13,24: Undefined variable 'platform' (undefined-variable)
13 E: 16,36: Too many positional arguments for function call (too-many-function-arg\
s)
```

```
15 C: 18, 4: Invalid method name "getWeight" (invalid-name)
16 C: 18, 4: Empty method docstring (empty-docstring)
17 E: 18, 4: Method should have "self" as first argument (no-self-argument)
18 R: 18, 4: Method could be a function (no-self-use)
19 R: 3, 0: Too few public methods (1/2) (too-few-public-methods)
20 W: 1, 0: Unused import sys (unused-import)
```

Let's take a moment to break this down a bit. First we need to figure out what the letters designate: C is for convention, R is refactor, W is warning and E is error. pylint found 3 errors, 4 convention issues, 2 lines that might be worth refactoring and 1 warning. The 3 errors plus the warning were what I was looking for. We should try to make this crummy code better and reduce the number of issues. We'll fix the imports and change the getWeight function to get_weight since camelCase isn't allowed for method names. We also need to fix the call to get_weight so it passes the right number of arguments and fix it so it has "self" as the first argument. Here's the new code:

```
1 # crummy_code_fixed.py
2 import platform
3
4 class CarClass:
5     """
6
7     def __init__(self, color, make, model, year):
8         """Constructor"""
9         self.color = color
10        self.make = make
11        self.model = model
12        self.year = year
13
14        if "Windows" in platform.platform():
15            print("You're using Windows!")
16
17        self.weight = self.get_weight(3)
18
19    def get_weight(self, this):
20        """
21
22        return "2000 lbs"
```

Let's run this new code against pylint and see how much we've improved the results. For brevity, we'll just show the first section again:

```
1 c:\py101>c:\Python34\Scripts\pylint crummy_code_fixed.py
2 No config file found, using default configuration
3 **** Module crummy_code_fixed
4 C: 1,0: Missing docstring
5 C: 4,0:CarClass: Empty docstring
6 C: 21,4:CarClass.get_weight: Empty docstring
7 W: 21,25:CarClass.get_weight: Unused argument 'this'
8 R: 21,4:CarClass.get_weight: Method could be a function
9 R: 4,0:CarClass: Too few public methods (1/2)
```

That helped a lot! If we added docstrings, we could halve the number of issues. Now we're ready to take a look at pyflakes!

Getting Started with pyflakes

The pyflakes project is a part of something known as the Divmod Project. Pyflakes doesn't actually execute the code it checks much like pylint doesn't execute the code it analyzes. You can install pyflakes using pip, easy_install or from source.

We will start by running pyflakes against the original version of the same piece of code that we used with pylint. Here it is again:

```
1 import sys
2
3 class CarClass:
4     """
5
6     def __init__(self, color, make, model, year):
7         """Constructor"""
8         self.color = color
9         self.make = make
10        self.model = model
11        self.year = year
12
13        if "Windows" in platform.platform():
14            print("You're using Windows!")
15
16        self.weight = self.getWeight(1, 2, 3)
17
18    def getWeight(this):
19        """
20
21        return "2000 lbs"
```

As was noted in the previous section, this silly piece of code has 4 issues, 3 of which would stop the program from running. Let's see what pyflakes can find! Try running the following command and you'll see the following output:

```
1 c:\py101>c:\Python34\Scripts\pyflakes.exe crummy_code.py
2 crummy_code.py:1: 'sys' imported but unused
3 crummy_code.py:13: undefined name 'platform'
```

While pyflakes was super fast at returning this output, it didn't find all the errors. The `getWeight` method call is passing too many arguments and the `getWeight` method itself is defined incorrectly as it doesn't have a `self` argument. Well, you can actually call the first argument anything you want, but by convention it's usually called `self`. If you fixed your code according to what pyflakes told you, your code still wouldn't work.

Wrapping Up

The next step would be to try running pylint and pyflakes against some of your own code or against a Python package like SQLAlchemy and seeing what you get for output. You can learn a lot about your own code using these tools. pylint is integrated with many popular Python IDEs, such as Wingware, Editra, and PyDev. You may find some of the warnings from pylint to be annoying or not even really applicable. There are ways to suppress such things as deprecation warnings through command line options. Or you can use the `-generate-rcfile` to create an example config file that will help you control pylint. Note that pylint and pyflakes does not import your code, so you don't have to worry about undesirable side effects.

Chapter 33 - The requests package

The **requests** package is a more Pythonic replacement for Python's own **urllib**. You will find that requests package's API is quite a bit simpler to work with. You can install the requests library by using pip or easy_install or from source.

Using requests

Let's take a look at a few examples of how to use the requests package. We will use a series of small code snippets to help explain how to use this library.

```
1 >>> r = requests.get("http://www.google.com")
```

This example returns a **Response** object. You can use the Response object's methods to learn a lot about how you can use requests. Let's use Python's **dir** function to find out what methods we have available:

```
1 >>> dir(r)
2 ['__attrs__', '__bool__', '__class__', '__delattr__', '__dict__',
3 '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
4 '__getstate__', '__gt__', '__hash__', '__init__', '__iter__', '__le__', '__lt__',
5 '__module__', '__ne__', '__new__', '__nonzero__', '__reduce__', '__reduce_ex__',
6 '__repr__', '__setattr__', '__setstate__', '__sizeof__', '__str__', '__subclasshook__',
7 '__weakref__', '_content', '_content_consumed', 'apparent_encoding', 'close',
8 'connection', 'content', 'cookies', 'elapsed', 'encoding', 'headers', 'history',
9 'iter_content', 'iter_lines', 'json', 'links', 'ok', 'raise_for_status', 'raw',
10 'reason', 'request', 'status_code', 'text', 'url']
```

If you run the following method, you can see the web page's source code:

```
1 >>> r.content()
```

The output from this command is way too long to include in the book, so be sure to try it out yourself. If you'd like to take a look at the web pages headers, you can run the following:

```
1 >>> r.headers
```

Note that the `headers` attribute returns a dict-like object and isn't a function call. We're not showing the output as web page headers tend to be too wide to show correctly in a book. There are a bunch of other great functions and attributes in the Response object. For example, you can get the cookies, the links in the page, and the `status_code` that the page returned.

The requests package supports the following HTTP request types: POST, GET, PUT, DELETE, HEAD and OPTIONS. If the page returns json, you can access it by calling the Response object's `json` method. Let's take a look at a practical example.

How to Submit a Web Form

In this section, we will compare how to submit a web form with requests versus urllib. Let's start by learning how to submit a web form. We will be doing a web search with **duckduckgo.com** searching on the term *python* and saving the result as an HTML file. We'll start with an example that uses `urllib`:

```
1 import urllib.request
2 import urllib.parse
3 import webbrowser
4
5 data = urllib.parse.urlencode({'q': 'Python'})
6 url = 'http://duckduckgo.com/html/'
7 full_url = url + '?' + data
8 response = urllib.request.urlopen(full_url)
9 with open("results.html", "wb") as f:
10     f.write(response.read())
11
12 webbrowser.open("results.html")
```

The first thing you have to do when you want to submit a web form is figure out what the form is called and what the url is that you will be posting to. If you go to duckduckgo's website and view the source, you'll notice that its action is pointing to a relative link, "/html". So our url is "<http://duckduckgo.com/html>". The input field is named "q", so to pass duckduckgo a search term, we have to concatenate the url to the "q" field. The results are read and written to disk. Now let's find out how this process differs when using the requests package.

The requests package does form submissions a little bit more elegantly. Let's take a look:

```
1 import requests
2
3 url = 'https://duckduckgo.com/html/'
4 payload = {'q': 'python'}
5 r = requests.get(url, params=payload)
6 with open("requests_results.html", "wb") as f:
7     f.write(r.content)
```

With `requests`, you just need to create a dictionary with the field name as the key and the search term as the value. Then you use `requests.get` to do the search. Finally you use the resulting `requests` object, “`r`”, and access its `content` property which you save to disk.

Wrapping Up

Now you know the basics of the `requests` package. I would recommend reading the package’s online documentation as it has many additional examples that you may find useful. I personally think this module is more intuitive to use than the standard library’s equivalent.

Chapter 34 - SQLAlchemy

SQLAlchemy is usually referred to as an **Object Relational Mapper (ORM)**, although it is much more full featured than any of the other Python ORMs that I've used, such as SqlObject or the one that's built into Django. SQLAlchemy was created by a fellow named Michael Bayer. Since I'm a music nut, we'll be creating a simple database to store album information. A database isn't a database without some relationships, so we'll create two tables and connect them. Here are a few other things we'll be learning:

- Adding data to each table
- Modifying data
- Deleting data
- Basic queries

But first we need to actually make the database, so that's where we'll begin our journey. You will need to install SQLAlchemy to follow along in this tutorial. We'll use pip to get this job done:

```
1 pip install sqlalchemy
```

Now we're ready to get started!

How to Create a Database

Creating a database with SQLAlchemy is really easy. SQLAlchemy uses a **Declarative** method for creating databases. We will write some code to generate the database and then explain how the code works. If you want a way to view your SQLite database, I would recommend the SQLite Manager plugin for Firefox. Here's some code to create our database tables:

```
1 # table_def.py
2 from sqlalchemy import create_engine, ForeignKey
3 from sqlalchemy import Column, Date, Integer, String
4 from sqlalchemy.ext.declarative import declarative_base
5 from sqlalchemy.orm import relationship, backref
6
7 engine = create_engine('sqlite:///mymusic.db', echo=True)
8 Base = declarative_base()
9
10 class Artist(Base):
11     """
12     __tablename__ = "artists"
13
14     id = Column(Integer, primary_key=True)
15     name = Column(String)
16
17 class Album(Base):
18     """
19     __tablename__ = "albums"
20
21     id = Column(Integer, primary_key=True)
22     title = Column(String)
23     release_date = Column(Date)
24     publisher = Column(String)
25     media_type = Column(String)
26
27     artist_id = Column(Integer, ForeignKey("artists.id"))
28     artist = relationship("Artist", backref=backref("albums", order_by=id))
29
30 # create tables
31 Base.metadata.create_all(engine)
```

If you run this code, then you should see something similar to the following output:

```
1 2014-04-03 09:43:57,541 INFO sqlalchemy.engine.base.Engine SELECT CAST('test pla\
2 in returns' AS VARCHAR(60)) AS anon_1
3 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
4 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine SELECT CAST('test uni\
5 code returns' AS VARCHAR(60)) AS anon_1
6 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
7 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("ar\
8 tists")
9 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
10 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine PRAGMA table_info("al\
11 bums")
12 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
13 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine
14 CREATE TABLE artists (
15     id INTEGER NOT NULL,
16     name VARCHAR,
17     PRIMARY KEY (id)
18 )
19
20
21 2014-04-03 09:43:57,551 INFO sqlalchemy.engine.base.Engine ()
22 2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine COMMIT
23 2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine
24 CREATE TABLE albums (
25     id INTEGER NOT NULL,
26     title VARCHAR,
27     release_date DATE,
28     publisher VARCHAR,
29     media_type VARCHAR,
30     artist_id INTEGER,
31     PRIMARY KEY (id),
32     FOREIGN KEY(artist_id) REFERENCES artists (id)
33 )
34
35
36 2014-04-03 09:43:57,661 INFO sqlalchemy.engine.base.Engine ()
37 2014-04-03 09:43:57,741 INFO sqlalchemy.engine.base.Engine COMMIT
```

Why did this happen? Because when we created the engine object, we set its `echo` parameter to `True`. The engine is where the database connection information is and it has all the DBAPI stuff in it that makes communication with your database possible. You'll note that we're creating a SQLite database. Ever since Python 2.5, SQLite has been supported by the language. If you want to connect

to some other database, then you'll need to edit the connection string. Just in case you're confused about what we're talking about, here is the code in question:

```
1 engine = create_engine('sqlite:///mymusic.db', echo=True)
```

The string, `sqlite:///mymusic.db`, is our connection string. Next we create an instance of the declarative base, which is what we'll be basing our table classes on. Next we have two classes, `Artist` and `Album` that define what our database tables will look like. You'll notice that we have `Columns`, but no column names. SQLAlchemy actually used the variable names as the column names unless you specifically specify one in the `Column` definition. You'll note that we are using an `id` Integer field as our primary key in both classes. This field will auto-increment. The other columns are pretty self-explanatory until you get to the `ForeignKey`. Here you'll see that we're tying the `artist_id` to the `id` in the `Artist` table. The relationship directive tells SQLAlchemy to tie the `Album` class/table to the `Artist` table. Due to the way we set up the `ForeignKey`, the relationship directive tells SQLAlchemy that this is a **many-to-one relationship**, which is what we want. Many albums to one artist. You can read more about table relationships [here¹⁵](#).

The last line of the script will create the tables in the database. If you run this script multiple times, it won't do anything new after the first time as the tables are already created. You could add another table though and then it would create the new one.

How to Insert / Add Data to Your Tables

A database isn't very useful unless it has some data in it. In this section we'll show you how to connect to your database and add some data to the two tables. It's much easier to take a look at some code and then explain it, so let's do that!

```
1 # add_data.py
2 import datetime
3 from sqlalchemy import create_engine
4 from sqlalchemy.orm import sessionmaker
5 from table_def import Album, Artist
6
7 engine = create_engine('sqlite:///mymusic.db', echo=True)
8
9 # create a Session
10 Session = sessionmaker(bind=engine)
11 session = Session()
12
13 # Create an artist
```

¹⁵http://docs.sqlalchemy.org/en/rel_0_7/orm/relationships.html#relationship-patterns

```
14 new_artist = Artist(name="Newsboys")
15
16 new_artist.albums = [Album(title="Read All About It",
17                             release_date=datetime.date(1988,12,1),
18                             publisher="Refuge", media_type="CD")]
19
20 # add more albums
21 more_albums = [Album(title="Hell Is for Wimps",
22                       release_date=datetime.date(1990,7,31),
23                       publisher="Star Song", media_type="CD"),
24                       Album(title="Love Liberty Disco",
25                             release_date=datetime.date(1999,11,16),
26                             publisher="Sparrow", media_type="CD"),
27                       Album(title="Thrive",
28                             release_date=datetime.date(2002,3,26),
29                             publisher="Sparrow", media_type="CD")]
30 new_artist.albums.extend(more_albums)
31
32 # Add the record to the session object
33 session.add(new_artist)
34 # commit the record the database
35 session.commit()
36
37 # Add several artists
38 session.add_all([
39     Artist(name="MXPX"),
40     Artist(name="Kutless"),
41     Artist(name="Thousand Foot Krutch")
42 ])
43 session.commit()
```

First we need to import our table definitions from the previous script. Then we connect to the database with our engine and create something new, the `Session` object. The `session` is our handle to the database and lets us interact with it. We use it to create, modify, and delete records and we also use sessions to query the database. Next we create an `Artist` object and add an album. You'll note that to add an album, you just create a list of `Album` objects and set the artist object's "albums" property to that list or you can extend it, as you see in the second part of the example. At the end of the script, we add three additional Artists using the `add_all`. As you have probably noticed by now, you need to use the session object's `commit` method to write the data to the database. Now it's time to turn our attention to modifying the data.

How to Modify Records with SQLAlchemy

What happens if you saved some bad data. For example, you typed your favorite album's title incorrectly or you got the release date wrong for that fan edition you own? Well you need to learn how to modify that record! This will actually be our jumping off point into learning SQLAlchemy queries as you need to find the record that you need to change and that means you need to write a query for it. Here's some code that shows us the way:

```
1 # modify_data.py
2 from sqlalchemy import create_engine
3 from sqlalchemy.orm import sessionmaker
4 from table_def import Album, Artist
5
6 engine = create_engine('sqlite:///mymusic.db', echo=True)
7
8 # create a Session
9 Session = sessionmaker(bind=engine)
10 session = Session()
11
12 # querying for a record in the Artist table
13 res = session.query(Artist).filter(Artist.name=="Kutless").first()
14 print(res.name)
15
16 # changing the name
17 res.name = "Beach Boys"
18 session.commit()
19
20 # editing Album data
21 artist, album = session.query(Artist, Album).filter(
22     Artist.id==Album.artist_id).filter(Album.title=="Thrive").first()
23 album.title = "Step Up to the Microphone"
24 session.commit()
```

Our first query goes out and looks up an Artist by name using the `filter` method. The `.first()` tells SQLAlchemy that we only want the first result. We could have used `.all()` if we thought there would be multiple results and we wanted all of them. Anyway, this query returns an Artist object that we can manipulate. As you can see, we changed the `name` from **Kutless** to **Beach Boys** and then committed our changes.

Querying a joined table is a little bit more complicated. This time we wrote a query that queries both our tables. It filters using the Artist id AND the Album title. It returns two objects: an artist and an album. Once we have those, we can easily change the title for the album. Wasn't that easy? At this

point, we should probably note that if we add stuff to the session erroneously, we can **rollback** our changes/adds/deletes by using `session.rollback()`. Speaking of deleting, let's tackle that subject!

How to Delete Records in SQLAlchemy

Sometimes you just have to delete a record. Whether it's because you're involved in a cover-up or because you don't want people to know about your love of Britney Spears music, you just have to get rid of the evidence. In this section, we'll show you how to do just that! Fortunately for us, SQLAlchemy makes deleting records really easy. Just take a look at the following code!

```
1 # deleting_data.py
2 from sqlalchemy import create_engine
3 from sqlalchemy.orm import sessionmaker
4 from table_def import Album, Artist
5
6 engine = create_engine('sqlite:///mymusic.db', echo=True)
7
8 # create a Session
9 Session = sessionmaker(bind=engine)
10 session = Session()
11
12 res = session.query(Artist).filter(Artist.name=="MXPX").first()
13
14 session.delete(res)
15 session.commit()
```

As you can see, all you had to do was create another SQL query to find the record you want to delete and then call `session.delete(res)`. In this case, we deleted our MXPX record. Some people think punk will never die, but they must not know any DBAs! We've already seen queries in action, but let's take a closer look and see if we can learn anything new.

The Basic SQL Queries of SQLAlchemy

SQLAlchemy provides all the queries you'll probably ever need. We'll be spending a little time just looking at a few of the basic ones though, such as a couple simple SELECTs, a JOINed SELECT and using the LIKE query. You'll also learn where to go for information on other types of queries. For now, let's look at some code:

```
1 # queries.py
2 from sqlalchemy import create_engine
3 from sqlalchemy.orm import sessionmaker
4 from table_def import Album, Artist
5
6 engine = create_engine('sqlite:///mymusic.db', echo=True)
7
8 # create a Session
9 Session = sessionmaker(bind=engine)
10 session = Session()
11
12 # how to do a SELECT * (i.e. all)
13 res = session.query(Artist).all()
14 for artist in res:
15     print(artist.name)
16
17 # how to SELECT the first result
18 res = session.query(Artist).filter(Artist.name=="Newsboys").first()
19
20 # how to sort the results (ORDER_BY)
21 res = session.query(Album).order_by(Album.title).all()
22 for album in res:
23     print(album.title)
24
25 # how to do a JOINed query
26 qry = session.query(Artist, Album)
27 qry = qry.filter(Artist.id==Album.artist_id)
28 artist, album = qry.filter(Album.title=="Step Up to the Microphone").first()
29
30 # how to use LIKE in a query
31 res = session.query(Album).filter(Album.publisher.like("S%a%")).all()
32 for item in res:
33     print(item.publisher)
```

The first query we run will grab all the artists in the database (a `SELECT`) and *print out each of their name fields*. Next you'll see how to just do a query for a specific artist and return just the first result. The third query shows how do a `SELECT` on the `Album` table and order the results by album title. The fourth query is the same query (a query on a `JOIN`) we used in our editing section except that we've broken it down to better fit PEP8 standards regarding line length. Another reason to break down long queries is that they become more readable and easier to fix later on if you messed something up. The last query uses `LIKE`, which allows us to pattern match or look for something that's "like" a specified string. In this case, we wanted to find any records that had a Publisher that started with a

capital “S”, some character, an “a” and then anything else. So this will match the publishers Sparrow and Star, for example.

SQLAlchemy also supports IN, IS NULL, NOT, AND, OR and all the other filtering keywords that most DBAs use. SQLAlchemy also supports literal SQL, scalars, etc, etc.

Wrapping Up

At this point you should know SQLAlchemy well enough to get started using it confidently. The project also has excellent documentation that you should be able to use to answer just about anything you need to know. If you get stuck, the SQLAlchemy users group / mailing list is very responsive to new users and even the main developers are there to help you figure things out.

Chapter 35 - virtualenv

Virtual environments can be really handy for testing software. That's true in programming circles too. Ian Bicking created the `virtualenv` project, which is a tool for creating isolated Python environments. You can use these environments to test out new versions of your software, new versions of packages you depend on or just as a sandbox for trying out some new package in general. You can also use `virtualenv` as a workspace when you can't copy files into site-packages because it's on a shared host. When you create a virtual environment with `virtualenv`, it creates a folder and copies Python into it along with a site-packages folder and a couple others. It also installs pip. Once your virtual environment is active, it's just like using your normal Python. And when you're done, you can just delete the folder to cleanup. No muss, no fuss. Alternatively, you can keep on using it for development.

In this chapter, we'll spend some time getting to know `virtualenv` and how to use it to make our own magic.

Installation

First of all, you probably need to install `virtualenv`. You can use pip or `easy_install` to install it or you can download the `virtualenv.py` file from their website and install it from source using its `setup.py` script.

If you have Python 3.4, you will find that you actually have the `venv` module, which follows an API that is very similar to the `virtualenv` package. This chapter will focus on just the `virtualenv` package, however.

Creating a Virtual Environment

Creating a virtual sandbox with the `virtualenv` package is quite easy. All you need to do is the following:

```
1 python virtualenv.py FOLDER_NAME
```

Where `FOLDER_NAME` is the name of the folder that you want your sandbox to go. On my Windows 7 machine, I have `C:\Python34\Scripts` added to my path so I can just call `virtualenv.py` `FOLDER_NAME` without the `python` part. If you don't pass it anything, then you'll get a list of options printed out on your screen. Let's say we create a project called `sandbox`. How do we use it? Well, we need to **activate** it. Here's how:

On Posix you would do source bin/activate while on Windows, you would do .\path\to\env\Scripts\activate. on the command line. Let's actually go through these steps. We'll create the sandbox folder on our desktop so you can see an example. Here's what it looks like on my machine:

```
1 C:\Users\mdriscoll\Desktop>virtualenv sandbox
2 New python executable in sandbox\Scripts\python.exe
3 Installing setuptools.....done.
4 Installing pip.....done.
5
6 C:\Users\mdriscoll\Desktop>sandbox\Scripts\activate
(sandbox) C:\Users\mdriscoll\Desktop>
```

You'll note that once your virtual environment is activated, you'll see your prompt change to include a prefix of the folder name that you created, which is **sandbox** in this case. This lets you know that you're using your sandbox. Now you can use pip to install other packages to your virtual environment. When you're done, you just call the deactivate script to exit the environment.

There are a couple of flags you can pass to virtualenv when creating your virtual playground that you should be aware of. For example, you can use **-system-site-packages** to inherit the packages from your default Python's site packages. If you want to use distribute rather than setuptools, you can pass virtualenv the **-distribute** flag.

virtualenv also provides a way for you to just install libraries but use the system Python itself to run them. According to the documentation, you just create a special script to do it.

There's also a neat (and experimental) flag called **-relocatable** that can be used to make the folder relocatable. However, this does NOT work on Windows at the time of this writing, so I wasn't able to test it out.

Finally, there's an **-extra-search-dir** flag that you can use to keep your virtual environment offline. Basically it allows you to add a directory to the search path for distributions that pip or easy_install can install from. In this way, you don't need to have access to the internet to install packages.

Wrapping Up

At this point, you should be able to use virtualenv yourself. There are a couple of other projects worth mentioning at this point. There's Doug Hellman's **virtualenvwrapper** library that makes it even easier to create, delete and manage virtual environments and then there's **zc.buildout** which is probably the closest thing to virtualenv that could be called a competitor. I recommend checking them both out as they might help you in your programming adventures.

Part V - Packaging and Distribution

In Part V you will learn about Python packaging and the various methods to distribute your code. You will learn about the following:

- How to create a module and package
- Publishing your Packages to the Python Packaging Index (PyPI)
- Python eggs
- Python wheels
- py2exe
- bb_freeze
- cx_Freeze
- PyInstaller
- GUI2Exe
- How to Create an Installer with InnoSetup

The first chapter of this section describes how to create a module or package. Then in the following chapter, we will go over publishing our package to PyPI. Next up, we will learn how to create and install a Python egg and the Python wheel.

The next four chapters will cover how to create binaries using the following 3rd party packages: py2exe, bb_freeze, cx_Freeze and PyInstaller. The only package in this list that is actually compatible with Python 3 is cx_Freeze. Because of this fact, we will be showing Python 2 examples in these four chapters so that we can easily compare all 4 packages and their capabilities.

The next to last chapter will show you how to use GUI2Exe, a neat little user interface that was created to go on top of py2exe, bb_freeze, etc. GUI2Exe makes creating binaries even easier!

The last chapter of this section will show how to create an installer using InnoSetup. Let's get started!



image

Chapter 36 - Creating Modules and Packages

Creating Python modules is something that most Python programmers do every day without even thinking about it. Any time you save a new Python script, you have created a new module. You can import your module into other modules. A package is a collection of related modules. The things you import into your scripts from the standard library are modules or packages. In this chapter, we will learn how to create modules and packages. We'll spend more time on packages since they're more complicated than modules.

How to Create a Python Module

We will begin by creating a super simple module. This module will provide us with basic arithmetic and no error handling. Here's our first example:

```
1 def add(x, y):
2     return x + y
3
4 def division(x, y):
5     return x / y
6
7 def multiply(x, y):
8     return x * y
9
10 def subtract(x, y):
11     return x - y
```

This code has issues, of course. If you pass in two Integers to the `division` method, then you'll end up getting an Integer back if you are using Python 2.x. This may not be what you're expecting. There's also no error checking for division by zero or mixing of strings and numbers. But that's not the point. The point is that if you save this code, you have a fully qualified module. Let's call it `arithmetic.py`. Now what can you do with a module anyway? You can import it and use any of the defined functions or classes that are inside it. And we could make it `executable` with a little spit and polish. Let's do both!

First we'll write a little script that imports our module and runs the functions in it. Save the following as `math_test.py`:

```
1 import arithmetic
2
3 print(arithmetic.add(5, 8))
4 print(arithmetic.subtract(10, 5))
5 print(arithmetic.division(2, 7))
6 print(arithmetic.multiply(12, 6))
```

Now let's modify the original script so that we can run it from the command line. Here's one of the simplest ways to do it:

```
1 def add(x, y):
2     return x + y
3
4 def division(x, y):
5     return x / y
6
7 def multiply(x, y):
8     return x * y
9
10 def subtract(x, y):
11     return x - y
12
13 if __name__ == "__main__":
14     import sys
15     print(sys.argv)
16     v = sys.argv[1].lower()
17     valOne = int(sys.argv[2])
18     valTwo = int(sys.argv[3])
19     if v == "a":
20         print(add(valOne, valTwo))
21     elif v == "d":
22         print(division(valOne, valTwo))
23     elif v == "m":
24         print(multiply(valOne, valTwo))
25     elif v == "s":
26         print(subtract(valOne, valTwo))
27     else:
28         pass
```

The proper way to do this script would be to use Python's `optparse` (pre-2.7) or `argparse` (2.7+) module. You should spend some time to figure out one of these modules as a learning exercise. In the meantime, we will move on to packages!

How to Create a Python Package

The main difference between a module and a package is that a package is a collection of modules AND it has an `__init__.py` file. Depending on the complexity of the package, it may have more than one `__init__.py`. Let's take a look at a simple folder structure to make this more obvious, then we'll create some simple code to follow the structure we define.

```
1 mymath/
2     __init__.py
3     adv/
4         __init__.py
5         sqrt.py
6         add.py
7         subtract.py
8         multiply.py
9         divide.py
```

Now we just need to replicate this structure in our own package. Let's give that a whirl! Create each of these files in a folder tree like the above example. For the add, subtract, multiply and divide files, you can use the functions we created in the earlier example. For the `sqrt.py` module, we'll use the following code.

```
1 # sqrt.py
2 import math
3
4 def squareroot(n):
5     return math.sqrt(n)
```

You can leave both `__init__.py` files blank, but then you'll have to write code like `mymath.add.add(x,y)` which is pretty ugly, so we'll add the following code to the outer `__init__.py` to make using our package easier to understand and use.

```
1 # outer __init__.py
2 from . add import add
3 from . divide import division
4 from . multiply import multiply
5 from . subtract import subtract
6 from .adv.sqrt import squareroot
```

Now we should be able to use our module once we have it on our Python path. You can copy the folder into your Python's `site-packages` folder to do this. On Windows it's in the following general location: `C:\Python34\Lib\site-packages`. Alternatively, you can edit the path on the fly in your test code. Let's see how that's done:

```
1 import sys
2
3 # modify this path to match your environment
4 sys.path.append('C:\Users\mdriscoll\Documents')
5
6 import mymath
7
8 print(mymath.add(4,5))
9 print(mymath.division(4, 2))
10 print(mymath.multiply(10, 5))
11 print(mymath.squareroot(48))
```

Note that my path does NOT include the **mymath** folder. You want to append the parent folder that holds your new module, NOT the module folder itself. If you do this, then the code above should work.

You can also create a **setup.py** script and install your package in **develop** mode. Here's an example **setup.py** script:

```
1 #!/usr/bin/env python
2
3 from setuptools import setup
4
5
6 # This setup is suitable for "python setup.py develop".
7
8 setup(name='mymath',
9       version='0.1',
10      description='A silly math package',
11      author='Mike Driscoll',
12      author_email='mike@mymath.org',
13      url='http://www.mymath.org/',
14      packages=['mymath', 'mymath.adv'],
15      )
```

You would save this script one level above the **mymath** folder. To install the package in **develop** mode, you would do the following:

```
1 python setup.py develop
```

This will install a link file in the site-packages folder that points to where ever your package resides. This is great for testing without actually installing your package.

Congratulations! You've just created a Python package!

Wrapping Up

You've just learned how to create your very own, custom-made modules and packages. You will find that the more you code, the more often you'll create programs that have parts that you want to re-use. You can put those reusable pieces of code into modules. Eventually you will have enough related modules that you may want to turn them into a package. Now you have the tools to actually do that!

Chapter 37 - How to Add Your Code to PyPI

We created a package called **mymath** in the previous chapter. In this chapter, we will learn how to share it on the Python Packaging Index (PyPI). To do that, we will first need to learn how to create a **setup.py** file. Just to review, here is our current folder hierarchy:

```
1 mymath/
2     __init__.py
3     adv/
4         __init__.py
5         sqrt.py
6     add.py
7     subtract.py
8     multiply.py
9     divide.py
```

This means that you have a **mymath** folder with the following files in it: **__init__.py**, **add.py**, **subtract.py**, **multiply.py** and **divide.py**. You will also have an **adv** folder inside the **mymath** folder. In the **adv** folder, you will have two files: **__init__.py** and **sqrt.py**.

Creating a **setup.py** File

We will start out by creating a super simple **setup.py** script. Here's a bare-bones one:

```
1 from distutils.core import setup
2
3 setup(name='mymath',
4       version='0.1',
5       packages=['mymath', 'mymath.adv'],
6       )
```

This is something you might write for a package internally. To upload to PyPI, you will need to include a little more information:

```
1 from distutils.core import setup
2
3 setup(name='mymath',
4       version='0.1',
5       description='A silly math package',
6       author='Mike Driscoll',
7       author_email='mike@mymath.org',
8       url='http://www.mymath.org/',
9       packages=[ 'mymath', 'mymath.adv'],
10      )
```

Now that we're done with that, we should test our script. You can create a virtual environment using the directions from chapter 35 or you can just install your code to your Python installation by calling the following command:

```
1 python setup.py install
```

Alternatively, you can use the method at the end of the last chapter in which you created a special `setup.py` that you installed in `develop` mode. You will note that in the last chapter, we used `setuptools` whereas in this chapter we used `distutils`. The only reason we did this is that `setuptools` has the `develop` command and `distutils` does not.

Now we need to register our package with PyPI!

Registering Packages

Registering your package is very easy. Since this is your first package, you will want to register with the Test PyPI server instead of the real one. You may need to create a `.pypirc` file and enter the Test PyPI server address. See the next section for more information. Once you have done that, you just need to run the following command:

```
1 python setup.py register
```

You will receive a list of options that ask you to login, register, have the server send you a password or quit. If you have your username and password saved on your machine, you won't see that message. If you're already registered, you can login and your package's metadata will be uploaded.

Uploading Packages to PyPI

You will probably want to start out by testing with PyPI's test server, which is at <https://testpypi.python.org/pypi>. You will have to register with that site too as it uses a different database than the main site. Once you've done that, you may want to create a `.pypirc` file somewhere on your operating system's path. On Linux, you can use `$HOME` to find it and on Windows, you can use the `HOME` environ variable. This path is where you would save that file. Following is a sample of what could go in your `pypirc` file from <https://wiki.python.org/moin/TestPyPI>:

```
1 [distutils]
2 index-servers=
3     pypi
4     test
5
6 [test]
7 repository = https://testpypi.python.org/pypi
8 username = richard
9 password = <your password goes here>
10
11 [pypi]
12 repository = http://pypi.python.org/pypi
13 username = richard
14 password = <your password goes here>
```

I would highly recommend that you read the documentation in depth to understand all the options you can add to this configuration file.

To upload some files to PyPI, you will need to create some distributions.

```
1 python setup.py sdist bdist_wininst upload
```

When you run the command above, it will create a `dist` folder. The `sdist` command will create an archive file (a zip on Windows, a tarball on Linux). The `bdist_wininst` will create a simple Windows installer executable. The `upload` command will then upload these two files to PyPI.

In your `setup.py` file, you can add a `long_description` field that will be used by PyPI to generate a home page for your package on PyPI. You can use reStructuredText to format your description. Or you can skip adding the description and accept PyPI's default formatting.

If you would like a full listing of the commands you can use with `setup.py`, try running the following command:

```
1 python setup.py --help-commands
```

You should also add a **README.txt** file that explains how to install and use your package. It can also contain a “Thank You” section if you have a lot of contributors.

Wrapping Up

Now you know the basics for adding your package to the Python Packaging Index. If you want to add a Python egg to PyPI, you will need to use `easy_install` instead of `distutils`. When you release your next version, you may want to add a **CHANGES.txt** file that lists the changes to your code. There is a great website called **The Hitchhiker’s Guide to Packaging** which would be a great place for you to check out for additional information on this exciting topic. Alternatively, you may also want to check out this [tutorial¹⁶](#) by Scott Torborg to get a different take on the process.

¹⁶<http://www.scotttorborg.com/python-packaging/index.html>

Chapter 38 - The Python egg

Python **eggs** are an older distribution format for Python. The new format is called a Python **wheel**, which we will look at in the next chapter. An egg file is basically a zip file with a different extension. Python can import directly from an egg. You will need the **SetupTools** package to work with eggs. **SetupTools** is the original mainstream method of downloading and installing Python packages from PyPI and other sources via the command line, kind of like apt-get for Python. There was a fork of **SetupTools** called **distribute** that was eventually merged back into **SetupTools**. I only mention it because you may see references to that fork if you do much reading about Python eggs outside of this book.

While the egg format is being migrated away from, you do still need to know about it as there are many packages that are distributed using this technology. It will probably be years before everyone has stopped using eggs. Let's learn how to make our own!

Creating an egg

You can think of an egg as just an alternative to a source distribution or Windows executable, but it should be noted that for pure Python eggs, the egg file is completely cross-platform. We will take a look at how to create our own egg using the package we created in a previous modules and packages chapter. To get started creating an egg, you will need to create a new folder and put the **mymath** folder inside it. Then create a **setup.py** file in the parent directory to mymath with the following contents:

```
1 from setuptools import setup, find_packages
2
3 setup(
4     name = "mymath",
5     version = "0.1",
6     packages = find_packages()
7 )
```

Python has its own package for creating distributions that is called **distutils**. However instead of using Python's **distutils**' **setup** function, we're using **setuptools**' **setup**. We're also using **setuptools**' **find_packages** function which will automatically look for any packages in the current directory and add them to the egg. To create said egg, you'll need to run the following from the command line:

```
1 c:\Python34\python.exe setup.py bdist_egg
```

This will generate a lot of output, but when it's done you'll see that you have three new folders: **build**, **dist**, and **mymath.egg-info**. The only one we care about is the **dist** folder in which you will find your egg file, **mymath-0.1-py3.4.egg**. Note that on my machine, I forced it to run against Python 3.4 so that it would create the egg against that version of Python. The egg file itself is basically a zip file. If you change the extension to "zip", you can look inside it and see that it has two folders: **mymath** and **EGG-INFO**. At this point, you should be able to point **easy_install** at your egg on your file system and have it install your package.

Wrapping Up

Now it's your turn. Go onto the Python Package Index and find some pure Python modules to download. Then try creating eggs using the techniques you learned in this chapter. If you want to install an egg, you can use **easy_install**. Uninstalling an egg is a bit tougher. You will have to go to its install location and delete the folder and / or egg file it installed as well as remove the entry for the package from the **easy-install.pth** file. All of these items can be found in your Python's **site-packages** folder.

Chapter 39 - Python wheels

Python's first mainstream packaging format was the .egg file. Now there's a new format in town called the **wheel** (.whl). According to the *Python Packaging Index's description*, a wheel is designed to contain all the files for a PEP 376 compatible install in a way that is very close to the on-disk format*. In this chapter, we will learn how to create a wheel and then install our wheel in a **virtualenv**.

Getting Started

Using pip is the recommended way to work with wheels. Make sure you have installed the latest copy of pip as older versions did not support the wheel format. If you're not sure if you have the latest pip, you can run the following command:

```
1 pip install --upgrade pip
```

If you didn't have the latest, then this command will upgrade pip. Now we're ready to create a wheel!

Creating a wheel

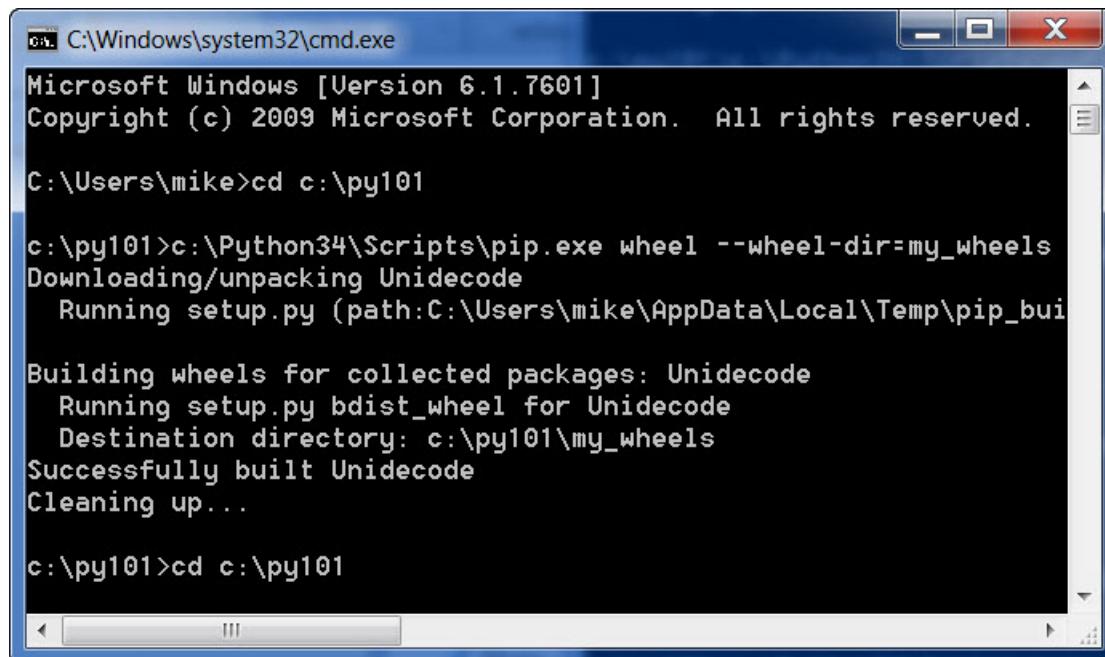
First of all, you will need to install the wheel package:

```
1 pip install wheel
```

That was easy! Next, we'll be using the **unidecode** package for creating our first wheel as it doesn't already have one made at the time of writing and I've used this package myself in several projects. The **unidecode** package will take a string of text and attempt to replace any unicode with its ASCII equivalent. This is extremely handy for when you have to scrub user provided data of weird anomalies. Here's the command you should run to create a wheel for this package:

```
1 pip wheel --wheel-dir=my_wheels Unidecode
```

Here's a screenshot of the output I received when I ran this:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\mike>cd c:\py101

c:\py101>c:\Python34\Scripts\pip.exe wheel --wheel-dir=my_wheels
Downloading/unpacking Unidecode
  Running setup.py (path:C:\Users\mike\AppData\Local\Temp\pip_bui
Building wheels for collected packages: Unidecode
  Running setup.py bdist_wheel for Unidecode
    Destination directory: c:\py101\my_wheels
Successfully built Unidecode
Cleaning up...

c:\py101>cd c:\py101
```

image

Now you should have a wheel named **Unidecode-0.04.14-py26-none-any.whl** in a folder named **my_wheels**. Let's learn how to install our new wheel!

Installing a Python wheel

Let's create a virtualenv to test with. We will use the following command to create our virtual testing environment:

```
1 virtualenv test
```

This assumes that **virtualenv** is on your system path. If you get an **unrecognized command** error, then you'll probably have to specify the full path (i.e. **c:\Python34\Scripts\virtualenv**). Running this command will create a virtual sandbox for us to play in that includes pip. Be sure to run the **activate** script from the **test** folder's Scripts folder to enable the virtuanenv before continuing. Your virtualenv does not include wheel, so you'll have to install wheel again:

```
1 pip install wheel
```

Once that is installed, we can install our wheel with the following command:

```
1 pip install --use-wheel --no-index --find-links=path/to/my_wheels Unidecode
```

To test that this worked, run Python from the Scripts folder in your virtualenv and try importing unidecode. If it imports, then you successfully installed your wheel!

The *.whl file* is similar to an.egg in that it's basically a *.zip file in disguise*. If you rename the extension from.whl to *.zip, you can open it up with your zip application of choice and examine the files and folders inside at your leisure.

Wrapping Up

Now you should be ready to create your own wheels. They are a nice way to create a local repository of dependencies for your project(s) that you can install quickly. You could create several different wheel repositories to easily switch between different version sets for testing purposes. When combined with virtualenv, you have a really easy way to see how newer versions of dependencies could affect your project without needing to download them multiple times.

Chapter 40 - py2exe

The py2exe project used to be the primary way to create Windows executables from your Python applications. The regular version of py2exe just supports Python 2.3-2.7. There is a new version listed on PyPI¹⁷ that will work with Python 3.4 as well. We will focus on the Python 2.x version, although this chapter should work with the Python 3 version too.

You have several choices for your application. You can create a program that only runs in a terminal, you can create a desktop graphical user interface (GUI) or you can create a web application. We will create a very simple desktop interface that doesn't do anything except display a form that the user can fill out. We will use the wxPython GUI toolkit to help demonstrate how py2exe can pick up packages without us telling it to.

Creating a Simple GUI

You will want to go to wxPython's website (www.wxpython.org) and download a copy that matches your Python version. If you have a 32-bit Python, make sure you download a 32-bit wxPython. You cannot use easy_install or pip to install wxPython unless you get the bleeding edge Phoenix version of wxPython, so you'll have to grab a copy that is pre-built for your system either from the wxPython website or from your system's package manager. I recommend using at least wxPython 2.9 or higher.

Let's write some code!

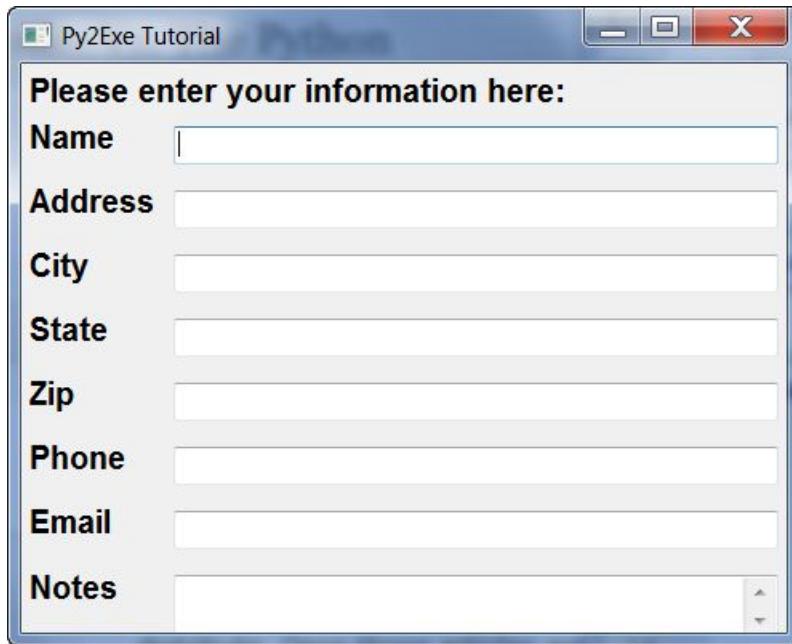
```
1 import wx
2
3 class DemoPanel(wx.Panel):
4     """
5
6     def __init__(self, parent):
7         """
8             "Constructor"
9         wx.Panel.__init__(self, parent)
10
11         labels = ["Name", "Address", "City", "State", "Zip",
12                 "Phone", "Email", "Notes"]
13
14         mainSizer = wx.BoxSizer(wx.VERTICAL)
15         lbl = wx.StaticText(self, label="Please enter your information here:")
```

¹⁷<https://pypi.python.org/pypi/py2exe/0.9.2.0/>

```
15     lbl.SetFont(wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD))
16     mainSizer.Add(lbl, 0, wx.ALL, 5)
17     for lbl in labels:
18         sizer = self.buildControls(lbl)
19         mainSizer.Add(sizer, 1, wx.EXPAND)
20     self.SetSizer(mainSizer)
21     mainSizer.Layout()
22
23 def buildControls(self, label):
24     """
25     Put the widgets together
26     """
27     sizer = wx.BoxSizer(wx.HORIZONTAL)
28     size = (80,40)
29     font = wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD)
30
31     lbl = wx.StaticText(self, label=label, size=size)
32     lbl.SetFont(font)
33     sizer.Add(lbl, 0, wx.ALL|wx.CENTER, 5)
34     if label != "Notes":
35         txt = wx.TextCtrl(self, name=label)
36     else:
37         txt = wx.TextCtrl(self, style=wx.TE_MULTILINE, name=label)
38     sizer.Add(txt, 1, wx.ALL, 5)
39     return sizer
40
41 class DemoFrame(wx.Frame):
42     """
43     Frame that holds all other widgets
44     """
45
46     def __init__(self):
47         """Constructor"""
48         wx.Frame.__init__(self, None, wx.ID_ANY,
49                           "Py2Exe Tutorial",
50                           size=(600,400)
51                           )
52         panel = DemoPanel(self)
53         self.Show()
54
55 if __name__ == "__main__":
56     app = wx.App(False)
```

```
57     frame = DemoFrame()  
58     app.MainLoop()
```

If you run the code above, you should see something like the following:



image

Let's break this down a bit. We create two classes, **DemoPanel** and **DemoFrame**. In wxPython, the **wx.Frame** object is what you use to create the actual “window” that you see in most cases. You add a **wx.Panel** to give your application the proper look and feel and to add tabbing between fields. The panel object’s parent is the frame. The frame, being the top level widget, has no parent. The panel contains all the other widgets in this example. We use sizers to help layout the widgets. Sizers allow the developer to create widgets that will resize appropriately when the window itself is resized. You can also place the widgets on the panel using absolute positioning, which is not recommended. We call the **MainLoop** method of the **wx.App** object at the end to start the event loop, which allows wxPython to respond to mouse and keyboard events (like clicking, typing, etc).

Now we're ready to learn how to package this application up into an executable!

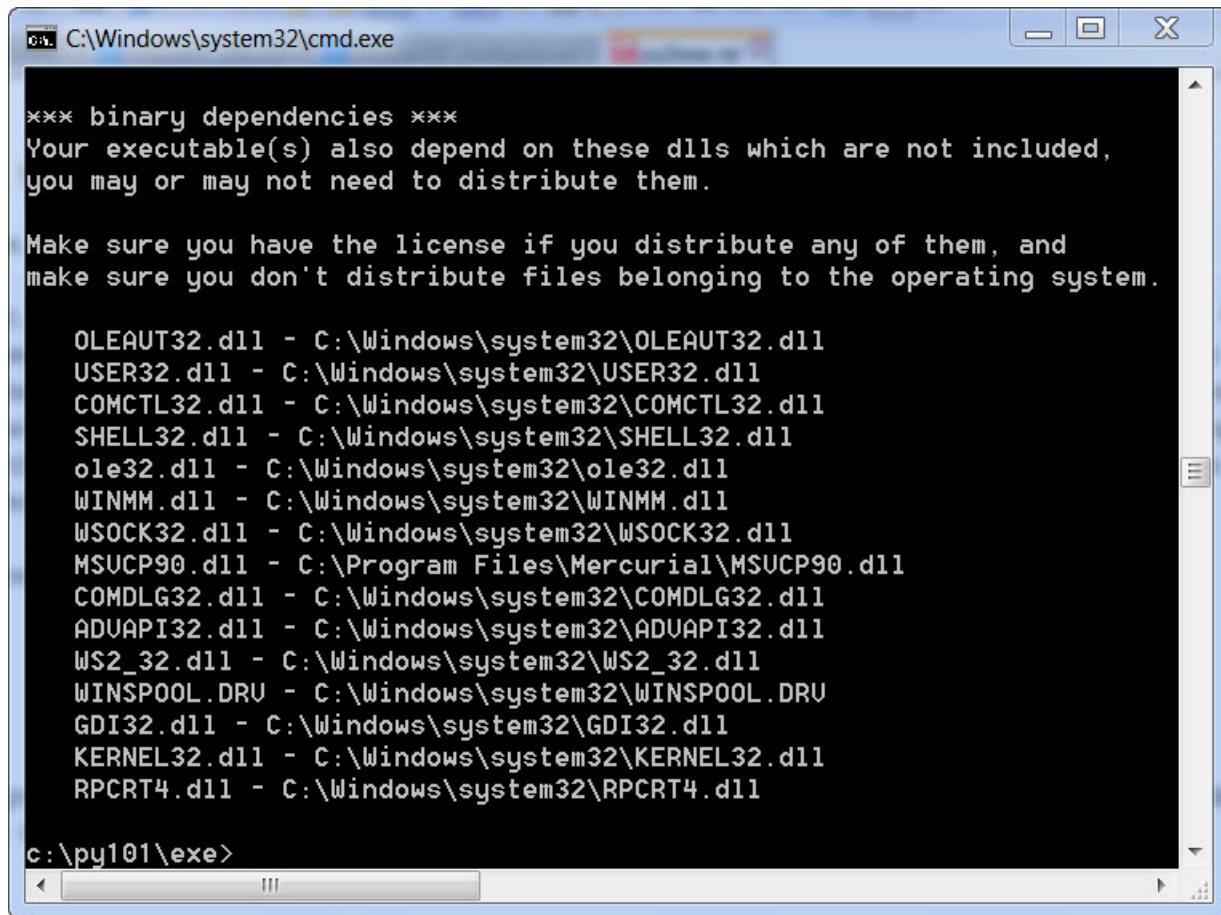
Note: I tested on Windows 7 using Python 2.7.3, wxPython 2.9.4.0 (classic) and py2exe 0.6.9.

The **py2exe setup.py** file

The key to any py2exe script is the **setup.py** file. This file controls what gets included or excluded, how much we compress and bundle, and much more! Here is the simplest setup that we can use with the wx script above:

```
1 from distutils.core import setup
2 import py2exe
3
4 setup(windows=[ 'sampleApp.py' ])
```

As you can see, we import the `setup` method from `distutils.core` and then we import `py2exe`. Next we call `setup` with a `windows` keyword parameter and pass it the name of the main file inside a python list object. If you were creating a non-GUI project, than you would use the `console` key instead of `windows`. To run this snippet, save it into the same folder as your wxPython script, open up a command prompt and navigate to the location that you saved the two files. Then type `python setup.py py2exe` to run it. If all goes well, you will see a lot of output ending with something like this:



image

If you happen to use Python 2.6, you might get an error about `MSVCP90.dll` not being found. Should you see that error, you will probably need to go find the **Microsoft Visual C++ 2008 Redistributable Package** and install it to make the DLL available on your system. Occasionally you will create the executable and then when you run it, it just won't load correctly. A log file is normally created when

this happens that you can use to try to figure out what happened. I have also found a tool called **Dependency Walker** that you can run against your executable and it can tell you about non-Python items that are missing (like DLLs, etc).

I would like to point out that the `setup.py` file doesn't explicitly include wxPython. That means that py2exe was smart enough to include the wxPython package automatically. Let's spend some time learning a bit more about including and excluding packages.

Creating an Advanced setup.py File

Let's see what other options py2exe gives us for creating binaries by creating a more complex `setup.py` file.

```
1 from distutils.core import setup
2 import py2exe
3
4 includes = []
5 excludes = ['_gtkagg', '_tkagg', 'bsddb', 'curses', 'email', 'pywin.debugger',
6             'pywin.debugger.dbgcon', 'pywin.dialogs', 'tcl',
7             'Tkconstants', 'Tkinter']
8 packages = []
9 dll_excludes = ['libgdk-win32-2.0-0.dll', 'libgobject-2.0-0.dll', 'tcl84.dll',
10                 'tk84.dll']
11
12 setup(
13     options = {"py2exe": {"compressed": 2,
14                           "optimize": 2,
15                           "includes": includes,
16                           "excludes": excludes,
17                           "packages": packages,
18                           "dll_excludes": dll_excludes,
19                           "bundle_files": 3,
20                           "dist_dir": "dist",
21                           "xref": False,
22                           "skip_archive": False,
23                           "ascii": False,
24                           "custom_boot_script": ''},
25             },
26     windows=[ 'sampleApp.py' ]
27 )
28 )
```

This is pretty self-explanatory, but let's unpack it anyway. First we set up a few lists that we pass to the options parameter of the setup function.

- The **includes** list is for special modules that you need to specifically include. Sometimes py2exe can't find certain modules, so you get to manually specify them here.
- The **excludes** list is a list of which modules to exclude from your program. In this case, we don't need Tkinter since we're using wxPython. This list of excludes is what GUI2Exe will exclude by default.
- The **packages** list is a list of specific packages to include. Again, sometimes py2exe just can't find something. I've had to include email, PyCrypto, or lxml here before. Note that if the excludes list contains something you're trying to include in the packages or includes lists, py2exe may continue to exclude them.
- **dll_excludes** - excludes dlls that we don't need in our project.

In the **options** dictionary, we have a few other options to look at. The **compressed** key tells py2exe whether or not to compress the zipfile, if it's set. The **optimize** key sets the optimization level. Zero is no optimization and 2 is the highest. By setting **optimize** to 2, we can reduce the size of folder by about one megabyte. The **bundle_files** key bundles dlls in the zipfile or the exe. Valid values for **bundle_files** are:

- 1 = bundle everything, including the Python interpreter.
- 2 = bundle everything but the Python interpreter
- 3 = don't bundle (default)

A few years ago, when I was first learning py2exe, I asked on their mailing list what the best option was because I was having issues with bundle option 1. I was told that 3 was probably the most stable. I went with that and stopped having random problems, so that's what I currently recommend. If you don't like distributing more than one file, zip them up or create an installer. The only other option I use in this list is the **dist_dir** one. I use it to experiment with different build options or to create custom builds when I don't want to overwrite my main good build. You can read about all the other options on the py2exe website.

The py2exe package does not support including Python eggs in its binaries, so if you have installed a package that your application depends on as an egg, when you go to create the executable, it won't work. You will have to make sure your dependencies are installed normally.

Wrapping Up

At this point, you should know enough to get started using py2exe yourself. You can get busy and start distributing your latest creation now. It should be noted that there are several alternatives to py2exe, such as **bbfreeze**, **cx_freeze** and **PyInstaller**. You should try at least a couple of the others to see how they compare. Creating executables can be frustrating, but have patience and persevere through it. The Python packaging community is quite willing to help. All you need to do is ask.

Chapter 41 - bbfreeze

The `bbfreeze` package also allows us to create binaries, but only on Linux and Windows. When you create a binary on Linux, the result will only run on machines that have the same hardware architecture and version of libc, which limits its usefulness on Linux. It should also be noted that `bbfreeze` only works with Python versions 2.4 - 2.7. You can use `easy_install` or `pip` to install the `bbfreeze` package onto your system. The `bbfreeze` package includes egg support, so it can include egg dependencies in your binary, unlike `py2exe`. You can also freeze multiple scripts at once, include the Python interpreter and more.

Getting Started with `bbfreeze`

You can use `easy_install` to download and install `bbfreeze` or you can just download its source or the egg file directly from the Python Package Index (PyPI). In this article, we'll try using it on a simple configuration file generator script and we'll also try it against the `wxPython` program from the `py2exe` chapter.

Note: I tested on Windows 7 using Python 2.7.3, wxPython 2.9.4.0 (classic) and bbfreeze 1.1.3

```
1 # config_1.py
2 import configobj
3
4 def createConfig(configFile):
5     """
6     Create the configuration file
7     """
8     config = configobj.ConfigObj()
9     configFile = configfile
10    config.filename = configFile
11    config['server'] = "http://www.google.com"
12    config['username'] = "mike"
13    config['password'] = "dingbat"
14    config['update interval'] = 2
15    config.write()
16
17 def getConfig(configFile):
18     """
19     Open the config file and return a configobj
```

```
20      """
21      return configobj.ConfigObj(configFile)
22
23  def createConfig2(path):
24      """
25          Create a config file
26      """
27
28      config = configobj.ConfigObj()
29      config.filename = path
30      config["Sony"] = {}
31      config["Sony"]["product"] = "Sony PS3"
32      config["Sony"]["accessories"] = ['controller', 'eye', 'memory stick']
33      config["Sony"]["retail price"] = "$400"
34
35      config.write()
36
37  if __name__ == "__main__":
38      createConfig2("sampleConfig2.ini")
```

This script has a couple of functions that are pretty pointless, but we'll leave them in for illustrative purposes. According to the bbfreeze documentation, we should be able to create a binary with the following string typed into the command line:

```
1 bb-freeze config_1.py
```

This assumes that you have C:\Python27\Scripts on your path. If you don't, you'll need to type the complete path out (i.e. C:\Python27\Scripts\bb-freeze config_1.py). If you run this, you should see a folder named **dist** get created. Here's what mine looked like after I ran **config_1.exe**:

Name	Date modified	Type	Size
hashlib.pyd	4/10/2012 11:31 PM	PYD File	279 KB
socket.pyd	4/10/2012 11:31 PM	PYD File	40 KB
ssl.pyd	4/10/2012 11:31 PM	PYD File	705 KB
bz2.pyd	4/10/2012 11:31 PM	PYD File	59 KB
config_1.exe	4/16/2014 7:11 PM	Application	8 KB
library.zip	4/18/2014 2:00 PM	Compressed (zipp...)	768 KB
py.exe	4/16/2014 7:11 PM	Application	8 KB
python27.dll	4/10/2012 11:31 PM	Application extens...	2,250 KB
sampleConfig2.ini	4/18/2014 2:00 PM	Configuration setti...	1 KB
select.pyd	4/10/2012 11:31 PM	PYD File	10 KB
unicodedata.pyd	4/10/2012 11:31 PM	PYD File	671 KB

image

You will note that when you run the executable, it creates a config file named **sampleconfig2.ini**. You may see a warning about the **pywin32** package not being installed. You can ignore the warning or download and install **pywin32**.

Now we're ready to move on and try to create an executable from code that use **wxPython**!

Using bbfreeze's Advanced Configuration

The PyPI page for **bbfreeze** (which is also its home page) has very little documentation. However, the page does say that the preferred way to use **bbfreeze** is with little scripts. We're going to try creating a binary with the **wxPython** example, mentioned earlier. Here's the **wx** code:

```
1 import wx
2
3 class DemoPanel(wx.Panel):
4     """
5
6     def __init__(self, parent):
7         """
8         wx.Panel.__init__(self, parent)
9
10        labels = ["Name", "Address", "City", "State", "Zip",
11                  "Phone", "Email", "Notes"]
```

```
13     mainSizer = wx.BoxSizer(wx.VERTICAL)
14     lbl = wx.StaticText(self, label="Please enter your information here:")
15     lbl.SetFont(wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD))
16     mainSizer.Add(lbl, 0, wx.ALL, 5)
17     for lbl in labels:
18         sizer = self.buildControls(lbl)
19         mainSizer.Add(sizer, 1, wx.EXPAND)
20     self.SetSizer(mainSizer)
21     mainSizer.Layout()
22
23 def buildControls(self, label):
24     """
25     Put the widgets together
26     """
27     sizer = wx.BoxSizer(wx.HORIZONTAL)
28     size = (80,40)
29     font = wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD)
30
31     lbl = wx.StaticText(self, label=label, size=size)
32     lbl.SetFont(font)
33     sizer.Add(lbl, 0, wx.ALL|wx.CENTER, 5)
34     if label != "Notes":
35         txt = wx.TextCtrl(self, name=label)
36     else:
37         txt = wx.TextCtrl(self, style=wx.TE_MULTILINE, name=label)
38     sizer.Add(txt, 1, wx.ALL, 5)
39     return sizer
40
41 class DemoFrame(wx.Frame):
42     """
43     Frame that holds all other widgets
44     """
45
46     def __init__(self):
47         """Constructor"""
48         wx.Frame.__init__(self, None, wx.ID_ANY,
49                           "Py2Exe Tutorial",
50                           size=(600,400)
51                           )
52         panel = DemoPanel(self)
53         self.Show()
```

```

55 if __name__ == "__main__":
56     app = wx.App(False)
57     frame = DemoFrame()
58     app.MainLoop()

```

Now let's create a simple freezing script!

```

1 # bb_setup.py
2 from bbfreeze import Freezer
3
4 f = Freezer(distdir="bb-binary")
5 f.addScript("sampleApp.py")
6 f()

```

First off, we import the `Freezer` class from the `bbfreeze` package. `Freezer` accepts three arguments: a destination folder, an `includes` iterable and an `excludes` iterable (i.e. a tuple or list). Just to see how well `bbfreeze` works with only its defaults, we leave out the `includes` and `excludes` tuples/lists. Once you have a `Freezer` object, you can add your script(s) by calling the `Freezer` object name's `addScript` method. Then you just need to call the object (i.e. `f()`).

Note: You may see a warning about `bb_freeze` not being able to find "MSVCP90.dll" or similar. If you see that message, you may need to include it explicitly or add it as a dependency when you create an installer. We will be learning about how to create an installer in a later chapter.

To run this script, you just have to do something like this:

```
1 python bb_setup.py
```

When I ran this script, it created a folder named `bb-binary` that contained 19 files that weighed in at 17.2 MB. When I ran the `sampleApp.exe` file, it ran just fine and was properly themed, however it also had a console screen. We'll have to edit our script a bit to fix that:

```

1 # bb_setup2.py
2 from bbfreeze import Freezer
3
4 includes = []
5 excludes = ['_gtkagg', '_tkagg', 'bsddb', 'curses', 'email', 'pywin.debugger',
6             'pywin.debugger.dbgcon', 'pywin.dialogs', 'tcl',
7             'Tkconstants', 'Tkinter']
8
9 bbFreeze_Class = Freezer('dist', includes=includes, excludes=excludes)
10

```

```
11 bbFreeze_Class.addScript("sampleApp.py", gui_only=True)
12
13 bbFreeze_Class.use_compression = 0
14 bbFreeze_Class.include_py = True
15 bbFreeze_Class()
```

If you run this, you should end up with a `dist` folder with about 19 files, but a slightly different size of 19.6 MB. Notice that we added a second argument to the `addScript` method: `gui_only=True`. This makes that annoying console go away. We also set compression to zero (no compression) and include the Python interpreter. Turning on compression only reduced the result back down to 17.2 MB though.

The `bbfreeze` package also handles “recipes” and includes several examples, however they are not documented well either. Feel free to study them yourself as an exercise.

Wrapping Up

Now you should know the basics of using `bbfreeze` to create binaries from your programs. I noticed that when I ran `bbfreeze` on my machine, it was considerably slower in producing the `wxPython` executable compared with `py2exe`. This is one of those things that you will have to experiment with when you are determining which tool to use to create your binaries.

Chapter 42 - cx_Freeze

In this chapter, we will be learning about **cx_Freeze**, a cross-platform set of scripts designed to **freeze** Python scripts into executables in a manner similar to py2exe, PyInstaller, etc. We will freeze one console script and one window (i.e GUI) script, using the examples from the previous chapter. The cx_Freeze tool is the **only** binary creation tool that can work with both Python 2.x and 3.x on multiple operating systems at this time. We will be using it with Python 2.7 in this chapter only because we want to compare it directly to the other binary creation tools.

You can install cx_Freeze using one of their Windows installers, via their provided Linux RPMs, via a source RPM or directly from source. You can also use **pip** to install cx_Freeze.

Note: I tested on Windows 7 using Python 2.7.3, wxPython 2.9.4.0 (classic) and cx_Freeze 4.3.2.

Getting Started with cx_Freeze

As mentioned on the cx_Freeze website, there are three ways to use this script. The first is to just use the included cxfreeze script; the second is to create a distutils setup script (think py2exe) which you can save for future use; and the third is to work with the internals of cxfreeze. We will focus on the first two ways of using cx_Freeze. We'll begin with the console script:

```
1 # config_1.py
2 import configobj
3
4 def createConfig(configFile):
5     """
6         Create the configuration file
7     """
8     config = configobj.ConfigObj()
9     configFile = configfile
10    config.filename = configFile
11    config['server'] = "http://www.google.com"
12    config['username'] = "mike"
13    config['password'] = "dingbat"
14    config['update interval'] = 2
15    config.write()
16
17 def getConfig(configFile):
18     """
```

```

19     Open the config file and return a configobj
20     """
21     return configobj.ConfigObj(configFile)
22
23 def createConfig2(path):
24     """
25     Create a config file
26     """
27     config = configobj.ConfigObj()
28     config.filename = path
29     config["Sony"] = {}
30     config["Sony"]["product"] = "Sony PS3"
31     config["Sony"]["accessories"] = ['controller', 'eye', 'memory stick']
32     config["Sony"]["retail price"] = "$400"
33     config.write()
34
35 if __name__ == "__main__":
36     createConfig2("sampleConfig2.ini")

```

All this script does is create a really simple configuration file using Michael Foord's **configobj** module. You can set it up to read the config too, but for this example, we'll skip that. Let's find out how to build a binary with cx_Freeze! According to the documentation, all it should take is the following string on the command line (assuming you are in the correct directory):

```
1 cxfreeze config_1.py --target-dir dirName
```

This assumes that you have C:\PythonXX\Scripts on your path. If not, you'll either have to fix that or type out the fully qualified path. Anyway, if the cxfreeze script runs correctly, you should have a folder with the following contents:

Name	Date modified	Type	Size
_hashlib.pyd	4/10/2012 11:31 PM	PYD File	279 KB
_socket.pyd	4/10/2012 11:31 PM	PYD File	40 KB
_ssl.pyd	4/10/2012 11:31 PM	PYD File	705 KB
bz2.pyd	4/10/2012 11:31 PM	PYD File	59 KB
config_1.exe	4/18/2014 2:32 PM	Application	1,155 KB
python27.dll	4/10/2012 11:31 PM	Application extens...	2,250 KB
sampleConfig2.ini	4/18/2014 2:32 PM	Configuration setti...	1 KB
unicodedata.pyd	4/10/2012 11:31 PM	PYD File	671 KB

image

As you can see, the total file size should around 5 megabytes. That was pretty easy. It even picked up the configobj module without our having to tell it to. There are 18 command line arguments you can pass to cx_Freeze to control how it does things. These range from what modules to include or exclude, optimization, compression, include a zip file, path manipulation and more.

Now let's try something a little more advanced.

Advanced cx_Freeze - Using a setup.py File

First off we need a script to use. We will use the wxPython form example from the previous chapters.

```
1 import wx
2
3 class DemoPanel(wx.Panel):
4     """
5
6     def __init__(self, parent):
7         """
8         wx.Panel.__init__(self, parent)
9
10        labels = ["Name", "Address", "City", "State", "Zip",
11                  "Phone", "Email", "Notes"]
12
13        mainSizer = wx.BoxSizer(wx.VERTICAL)
14        lbl = wx.StaticText(self, label="Please enter your information here:")
15        lbl.SetFont(wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD))
16        mainSizer.Add(lbl, 0, wx.ALL, 5)
17        for lbl in labels:
18            sizer = self.buildControls(lbl)
19            mainSizer.Add(sizer, 1, wx.EXPAND)
20        self.SetSizer(mainSizer)
21        mainSizer.Layout()
22
23    def buildControls(self, label):
24        """
25        Put the widgets together
26        """
27        sizer = wx.BoxSizer(wx.HORIZONTAL)
28        size = (80,40)
29        font = wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD)
30
31        lbl = wx.StaticText(self, label=label, size=size)
```

```

32         lbl.SetFont(font)
33         sizer.Add(lbl, 0, wx.ALL|wx.CENTER, 5)
34     if label != "Notes":
35         txt = wx.TextCtrl(self, name=label)
36     else:
37         txt = wx.TextCtrl(self, style=wx.TE_MULTILINE, name=label)
38     sizer.Add(txt, 1, wx.ALL, 5)
39     return sizer
40
41 class DemoFrame(wx.Frame):
42     """
43     Frame that holds all other widgets
44     """
45
46     def __init__(self):
47         """Constructor"""
48         wx.Frame.__init__(self, None, wx.ID_ANY,
49                           "Py2Exe Tutorial",
50                           size=(600,400)
51                           )
52         panel = DemoPanel(self)
53         self.Show()
54
55 if __name__ == "__main__":
56     app = wx.App(False)
57     frame = DemoFrame()
58     app.MainLoop()

```

Now let's create a `setup.py` file in the cx_Freeze style:

```

1 # setup.py
2 from cx_Freeze import setup, Executable
3
4 setup(
5     name = "wxSampleApp",
6     version = "0.1",
7     description = "An example wxPython script",
8     executables = [Executable("sampleApp.py")]
9 )

```

As you can see, this is a pretty simple one. We import a couple classes from cx_Freeze and pass some parameters into them. In this case, we give the setup class a name, version, description and

Executable class. The Executable class also gets one parameter, the script name that it will use to create the binary from.

Alternatively, you can create a simple setup.py using cx_Freeze's quickstart command (assuming it's on your system's path) in the same folder as your code:

```
cxfreeze-quickstart
```

To get the setup.py to build the binary, you need to do the following on the command line:

```
1 python setup.py build
```

After running this, you should end up with the following folders: **buildexe.win32-2.7**. Inside that last folder I ended up with 15 files that total 16.6 MB. When you run the sampleApp.exe file, you will notice that we've screwed something up. There's a console window loading in addition to our GUI! To rectify this, we'll need to change our setup file slightly. Take a look at our new one:

```
1 from cx_Freeze import setup, Executable
2
3 exe = Executable(
4     script="sampleApp.py",
5     base="Win32GUI",
6 )
7
8 setup(
9     name = "wxSampleApp",
10    version = "0.1",
11    description = "An example wxPython script",
12    executables = [exe]
13 )
```

First off, we separated the Executable class from the setup class and assigned the Executable class to a variable. We also added a second parameter to the Executable class that is key. That parameter is called **base**. By setting **base="Win32GUI"**, we are able to suppress the console window. The documentation on the cx_Freeze website shows the many other options that the Executable class takes.

Wrapping Up

Now you should know how to create binaries with cx_Freeze. It's pretty easy to do and it ran a lot faster than bbfreeze did in my testing. If you have the need to create binaries for both Python 2.x and 3.x on all major platforms, then this is the tool for you!

Chapter 43 - PyInstaller

PyInstaller is the last tool we will be looking at for creating binaries. It supports Python 2.4 - 2.7. We will continue to use our simple console and wxPython GUI scripts for our testing. PyInstaller is supposed to work on Windows, Linux, Mac, Solaris and AIX. The support for Solaris and AIX is experimental. PyInstaller supports code-signing (Windows), eggs, hidden imports, single executable, single directory, and lots more!

Note: I tested on Windows 7 using Python 2.7.3, wxPython 2.9.4.0 (classic) and PyInstaller 2.1.

Getting Started with PyInstaller

To install PyInstaller, you can download the source code in a tarball or zip archive, decompress it and run its `setup.py` file:

```
1 python setup.py install
```

You can also install PyInstaller using pip. We will start with our little piece of config creation code:

```
1 # config_1.py
2 import configobj
3
4 def createConfig(configFile):
5     """
6     Create the configuration file
7     """
8     config = configobj.ConfigObj()
9     infile = configFile
10    config.filename = infile
11    config['server'] = "http://www.google.com"
12    config['username'] = "mike"
13    config['password'] = "dingbat"
14    config['update interval'] = 2
15    config.write()
16
17 def getConfig(configFile):
18     """
19     Open the config file and return a configobj
```

```
20      """
21      return configobj.ConfigObj(configFile)
22
23  def createConfig2(path):
24      """
25      Create a config file
26      """
27      config = configobj.ConfigObj()
28      config.filename = path
29      config["Sony"] = {}
30      config["Sony"]["product"] = "Sony PS3"
31      config["Sony"]["accessories"] = ['controller', 'eye', 'memory stick']
32      config["Sony"]["retail price"] = "$400"
33      config.write()
34
35  if __name__ == "__main__":
36      createConfig2("sampleConfig2.ini")
```

Now let's try to create an executable! You should be able to just do this to get PyInstaller to work:

```
1 pyinstaller config_1.py
```

When I ran this, I got the following error:

```
1 Error: PyInstaller for Python 2.6+ on Windows needs pywin32.
2 Please install from http://sourceforge.net/projects/pywin32/
```

To use PyInstaller on Windows, you will need to install PyWin32 first! Once you have PyWin32 installed, try re-running that command. You should see a lot of output sent to the screen and you should also see these two folders appear next to your script: **build** and **dist**. If you go into the **dist** folder, and then into its **config_1** folder, you should see something like this:

Name	Date modified	Type	Size
__hashlib.pyd	4/10/2012 11:31 PM	PYD File	279 KB
__socket.pyd	4/10/2012 11:31 PM	PYD File	40 KB
__ssl.pyd	4/10/2012 11:31 PM	PYD File	705 KB
bz2.pyd	4/10/2012 11:31 PM	PYD File	59 KB
config_1.exe	4/18/2014 3:26 PM	Application	1,287 KB
config_1.exe.manifest	4/18/2014 3:26 PM	MANIFEST File	1 KB
Microsoft.VC90.CRT.manifest	10/15/2012 9:20 PM	MANIFEST File	2 KB
msvcm90.dll	10/15/2012 9:20 PM	Application exte...	220 KB
msvcp90.dll	10/15/2012 9:20 PM	Application exte...	556 KB
msvcr90.dll	10/15/2012 9:20 PM	Application exte...	641 KB
python27.dll	4/10/2012 11:31 PM	Application exte...	2,250 KB
sampleConfig2.ini	4/18/2014 3:26 PM	Configuration se...	1 KB
select.pyd	4/10/2012 11:31 PM	PYD File	10 KB
unicodedata.pyd	4/10/2012 11:31 PM	PYD File	671 KB

image

When I ran the executable, it created the config file just as it should. You will notice that PyInstaller was able to grab `configobj` without you needing to tell it to.

PyInstaller and wxPython

Now let's try creating a binary from a simple wxPython script. Here's the wxPython code that we've been using in previous chapters:

```
1 import wx
2
3 class DemoPanel(wx.Panel):
4     """
5
6     def __init__(self, parent):
7         """
8             "Constructor"
9         wx.Panel.__init__(self, parent)
10
11         labels = ["Name", "Address", "City", "State", "Zip",
12                 "Phone", "Email", "Notes"]
13
14         mainSizer = wx.BoxSizer(wx.VERTICAL)
15         lbl = wx.StaticText(self, label="Please enter your information here:")
```

```
15     lbl.SetFont(wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD))
16     mainSizer.Add(lbl, 0, wx.ALL, 5)
17     for lbl in labels:
18         sizer = self.buildControls(lbl)
19         mainSizer.Add(sizer, 1, wx.EXPAND)
20     self.SetSizer(mainSizer)
21     mainSizer.Layout()
22
23 def buildControls(self, label):
24     """
25     Put the widgets together
26     """
27     sizer = wx.BoxSizer(wx.HORIZONTAL)
28     size = (80,40)
29     font = wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD)
30
31     lbl = wx.StaticText(self, label=label, size=size)
32     lbl.SetFont(font)
33     sizer.Add(lbl, 0, wx.ALL|wx.CENTER, 5)
34     if label != "Notes":
35         txt = wx.TextCtrl(self, name=label)
36     else:
37         txt = wx.TextCtrl(self, style=wx.TE_MULTILINE, name=label)
38     sizer.Add(txt, 1, wx.ALL, 5)
39     return sizer
40
41 class DemoFrame(wx.Frame):
42     """
43     Frame that holds all other widgets
44     """
45
46     def __init__(self):
47         """Constructor"""
48         wx.Frame.__init__(self, None, wx.ID_ANY,
49                           "Py2Exe Tutorial",
50                           size=(600,400)
51                           )
52         panel = DemoPanel(self)
53         self.Show()
54
55 if __name__ == "__main__":
56     app = wx.App(False)
```

```
57     frame = DemoFrame()
58     app.MainLoop()
```

If you execute the **pyinstaller** command against this script, you will see ever more output sent to the screen. It will create 23 files that total 19.4 MB. You will also notice that when you run the **sampleApp.exe**, it shows a console window in addition to your GUI, which is not what we want. The simplest way to fix that is to call PyInstaller with the **-w** command which tells PyInstaller to suppress the console window:

```
1 pyinstaller -w sampleApp.py
```

The PyInstaller package has many command line options that you can use to change the way PyInstaller processes your program. Whenever you run PyInstaller, it will create a **spec** file that it uses to process your program. If you'd like to save a copy of the spec file to help you better understand what PyInstaller is doing, you can do so using the following command:

```
1 pyi-makespec sampleApp.py
```

You can pass the same commands to **pyi-makespec** as you do to PyInstaller, which will change the spec appropriately. Here's the contents of the spec that was created with the previous command:

```
1 # -*- mode: python -*-
2 a = Analysis(['sampleApp.py'],
3             pathex=['c:\\py101\\wxpy'],
4             hiddenimports=[],
5             hookspath=None,
6             runtime_hooks=None)
7 pyz = PYZ(a.pure)
8 exe = EXE(pyz,
9            a.scripts,
10           exclude_binaries=True,
11           name='sampleApp.exe',
12           debug=False,
13           strip=None,
14           upx=True,
15           console=False )
16 coll = COLLECT(exe,
17                  a.binaries,
18                  a.zipfiles,
19                  a.datas,
20                  strip=None,
21                  upx=True,
22                  name='sampleApp')
```

In earlier versions of PyInstaller, you would actually create the spec file and edit it directly. Now unless you need something really special, you can generate the right spec by just using flags. Be sure to read the documentation for full details as there are many flags and describing them all is outside the scope of this chapter.

Wrapping Up

This ends our quick tour of PyInstaller. I hope you found this helpful in your Python binary-making endeavors. The PyInstaller project is pretty well documented and worth your time to check out.

Chapter 44 - Creating an Installer

In this chapter, we will walk you through the process of creating an executable and then packaging it up into an installer. We will be using a really neat little user interface called GUI2Exe that was written by Andrea Gavana to create the executable. It is based on wxPython, so you will need to have that installed to use it. GUI2Exe supports py2exe, bbfreeze, cx_Freeze, PyInstaller and py2app. Then once we have the **dist** folder created, we will use **Inno Setup** to create our installer.

We will be using the following code once more:

```
1 # sampleApp.py
2
3 import wx
4
5 class DemoPanel(wx.Panel):
6     """
7
8     def __init__(self, parent):
9         """
10        "Constructor"
11        wx.Panel.__init__(self, parent)
12
13        labels = ["Name", "Address", "City", "State", "Zip",
14                  "Phone", "Email", "Notes"]
15
16        mainSizer = wx.BoxSizer(wx.VERTICAL)
17        lbl = wx.StaticText(self, label="Please enter your information here:")
18        lbl.SetFont(wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD))
19        mainSizer.Add(lbl, 0, wx.ALL, 5)
20        for lbl in labels:
21            sizer = self.buildControls(lbl)
22            mainSizer.Add(sizer, 1, wx.EXPAND)
23        self.SetSizer(mainSizer)
24        mainSizer.Layout()
25
26    def buildControls(self, label):
27        """
28        Put the widgets together
29        """
30
31        sizer = wx.BoxSizer(wx.HORIZONTAL)
```

```
30         size = (80,40)
31         font = wx.Font(12, wx.SWISS, wx.NORMAL, wx.BOLD)
32
33         lbl = wx.StaticText(self, label=label, size=size)
34         lbl.SetFont(font)
35         sizer.Add(lbl, 0, wx.ALL|wx.CENTER, 5)
36         if label != "Notes":
37             txt = wx.TextCtrl(self, name=label)
38         else:
39             txt = wx.TextCtrl(self, style=wx.TE_MULTILINE, name=label)
40         sizer.Add(txt, 1, wx.ALL, 5)
41         return sizer
42
43 class DemoFrame(wx.Frame):
44     """
45     Frame that holds all other widgets
46     """
47
48     def __init__(self):
49         """Constructor"""
50         wx.Frame.__init__(self, None, wx.ID_ANY,
51                           "Py2Exe Tutorial",
52                           size=(600,400)
53                           )
54         panel = DemoPanel(self)
55         self.Show()
56
57 if __name__ == "__main__":
58     app = wx.App(False)
59     frame = DemoFrame()
60     app.MainLoop()
```

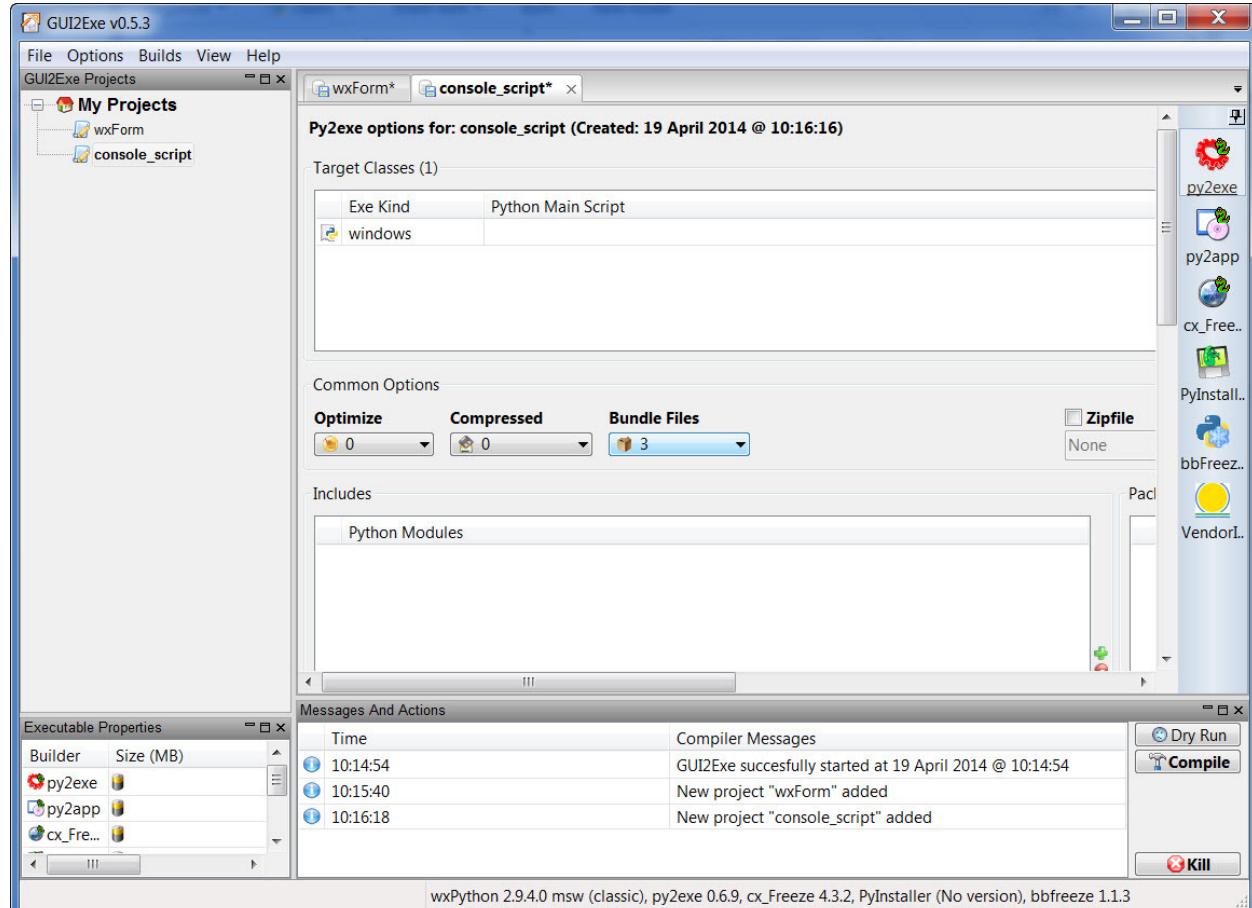
Let's get started!

Getting Started with GUI2Exe

To use GUI2Exe, you just have to go to its website (<http://code.google.com/p/gui2exe/>) and download a release. Then you unzip it and run the script that's called `GUI2Exe.py`. The GUI2Exe project is based on wxPython, so make sure you have that installed as well. I ran mine successfully with wxPython 2.9. Here's how you would call it:

```
1 python GUI2Exe.py
```

If that executed successfully, you should see a screen similar to this one:



image

Now go to File -> New Project and give your project a name. In this case, I called the project **wxForm**. If you want to, you can add a fake Company Name, Copyright and give it a Program Name. Be sure to also browse for your main Python script (i.e. `sampleApp.py`). According to Andrea's website, you should set **Optimize** to 2, **Compressed** to 2 and **Bundled Files** to 1. This seems to work most of the time, but I've had some screwy errors that seem to stem from setting the last one to 1. In fact, according to one of my contacts on the py2exe mailing list, the **bundle** option should be set to 3 to minimize errors. The nice thing about setting bundle to "1" is that you end up with just one file, but since I'm going to roll it up with Inno I'm going to go with option 3 to make sure my program works well.

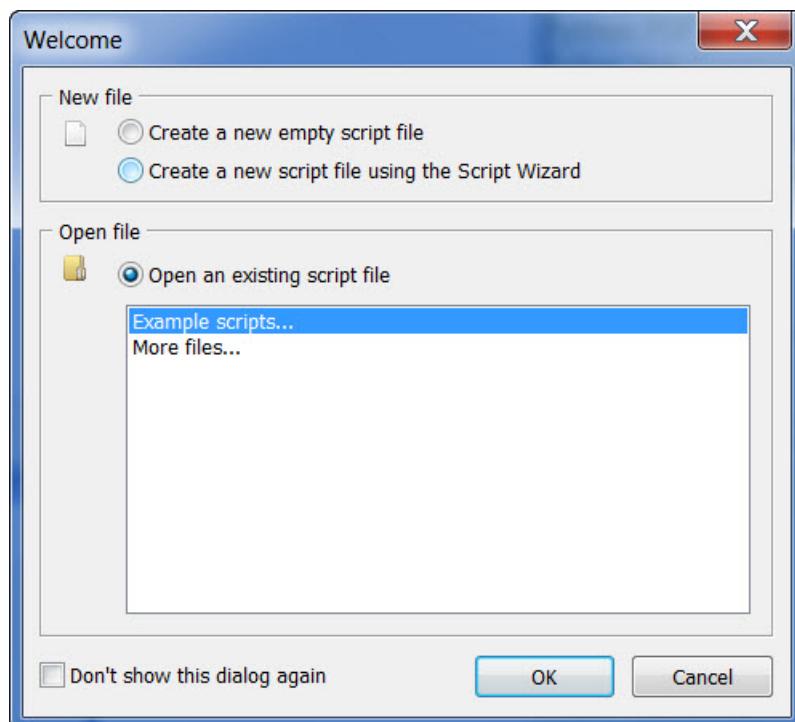
Once you have everything the way you want it, click the **Compile** button in the lower right-hand corner. This will create all the files you want to distribute in the **dist** folder unless you have changed the name by checking the **dist checkbox** and editing the subsequent textbox. When it's done compiling, GUI2Exe will ask you if you want to test your executable. Go ahead and hit **Yes**. If

you receive any errors about missing modules, you can add them in the **Python Modules** or **Python Packages** section as appropriate. For this example, you shouldn't have that issue though.

Now we're ready to learn about creating the installer!

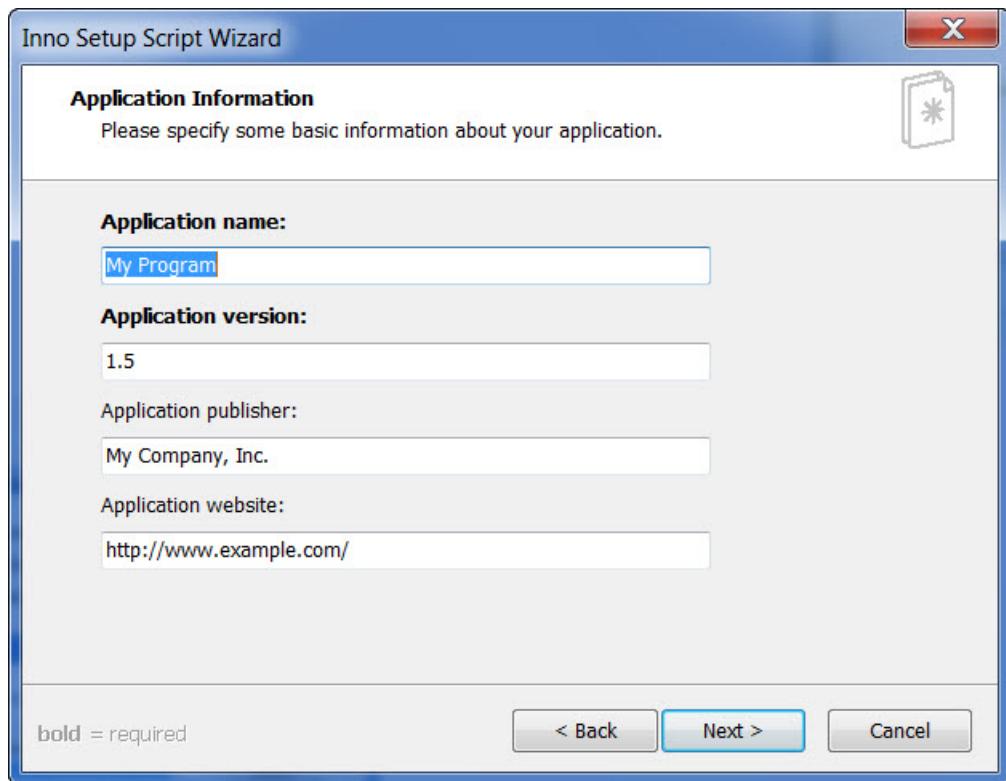
Let's Make an Installer!

Now that we have an executable and a bunch of dependencies, how do we make an installer? For this chapter we'll be using Inno Setup, but you could also use NSIS or a Microsoft branded installer. You will need to go to their website (<http://www.jrsoftware.org/isdl.php>), download the program and install it. Then run the program. You should see the main program along with the following dialog on top of it:



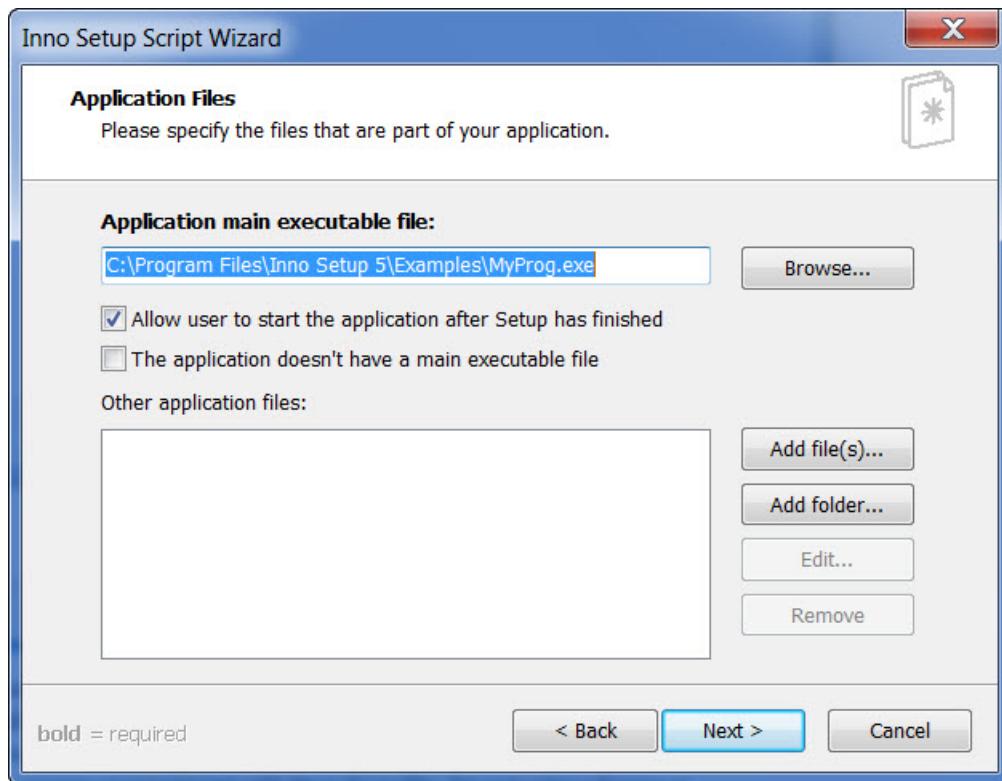
image

Choose the **Create a new script using the Script Wizard** option and then press the **OK** button. Click **Next** and you should see something like this:



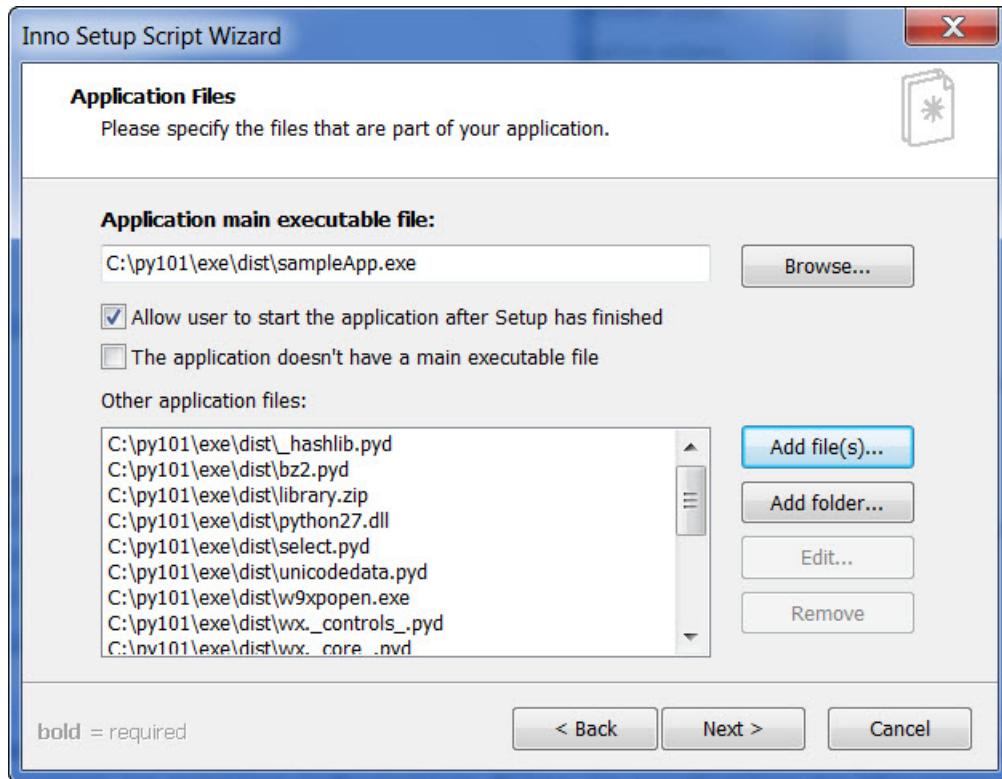
image

Fill this out however you like and click **Next** (I called mine **wxForm**). This next screen allows you to choose where you want the application to be installed by default. It defaults to **Program Files** which is fine. Click **Next**. Now you should see the following screen:



image

Browse to the executable you created to add it. Then click the **Add file(s)...** button to add the rest. You can actually select all of the files except for the exe and hit OK. This is how mine turned out:



image

Now you're ready to click **Next**. Make sure the Start Menu folder has the right name (in this case, **wxForm**) and continue. You can ignore the next two screens or experiment with them if you like. I'm not using a license or putting information files in to display to the user though. The last screen before finishing allows you to choose a directory to put the output into. I just left that empty since it defaults to where the executable is located, which is fine with this example. Click **Next**, **Next** and **Finish**. This will generate a full-fledged .iss file, which is what Inno Setup uses to turn your application into an installer. It will ask you if you'd like to go ahead and compile the script now. Go ahead and do that. Then it will ask if you'd like to save your .iss script. That's a good idea, so go ahead and do that too. Hopefully you didn't receive any errors and you can try out your new installer.

If you're interested in learning about Inno's scripting language, feel free to read Inno's Documentation. You can do quite a bit with it. If you happen to make changes to your build script, you can rebuild your installer by going to the **build** menu and choosing the **compile** menu item.

Wrapping Up

At this point, you now know how to create a real, live installer that you can use to install your application and any files it needs to run. This is especially handy when you have a lot of custom icons for your toolbars or a default database, config file, etc that you need to distribute with your

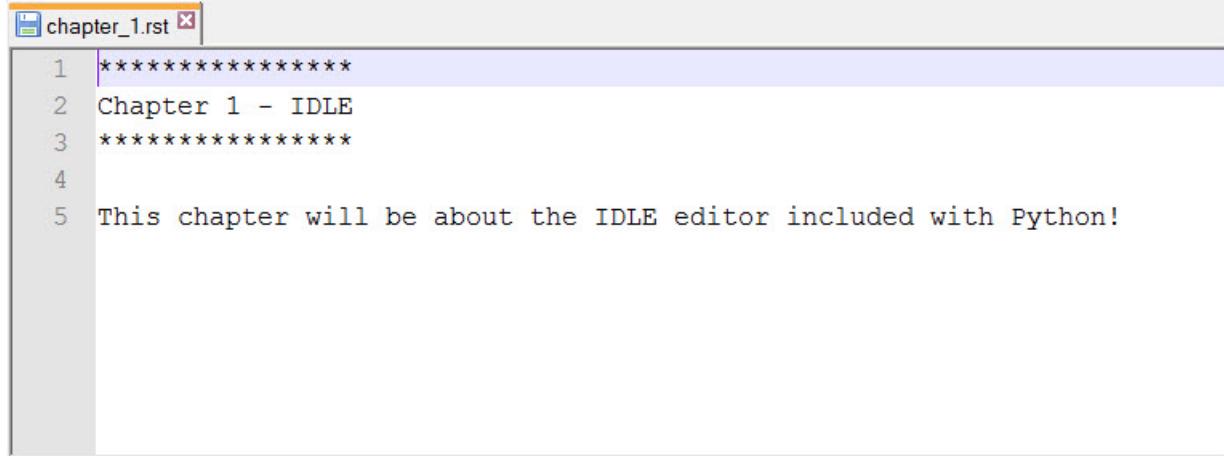
application. Go back and try creating the installer again, but choose different options to see what else you can do. Experimenting is a great way to learn. Just make sure you always have a backup in case something goes wrong!

Appendix A: Putting It All Together

Now that you've finished reading the book, what do you do? Most beginner programming books I've read would leave you hanging at this point, but not this one. In this chapter, we will take what we've learned and apply it to create a real program! First we need some background and a set of requirements. In the business world, we might call this a **specification**. The more specific the specification is, the more likely your code will match the customer's expectations. Unfortunately, you will find that in the real world, good specifications are hard to come by and you have to spend a lot of time nailing down exactly what the customer wants. But I digress. Let's get started on our program!

The Background

This book was written using a markup language called **RestructuredText**. Each chapter of the book was saved in its own file (Ex. chapter1.rst) as I have found that my proofers would rather proof individual chapters than get overwhelmed by large chunks of the book. Here is a really short example that demonstrates how to create a chapter heading:



A screenshot of a text editor window titled "chapter_1.rst". The code inside the window is as follows:

```
1 *****
2 Chapter 1 - IDLE
3 *****
4
5 This chapter will be about the IDLE editor included with Python!
```

image

You may want to take a look at this [Quick Reference¹⁸](#) to learn more about RestructuredText.

The Specification

What I need to be able to do is find a way to produce a PDF out of these individual chapters, add the customer's name and email address to each page and email the PDF to each customer. This was

¹⁸<http://docutils.sourceforge.net/docs/user/rst/quickref.html>

my original idea for distributing the book to my Kickstarter backers. Note that Kickstarter provides the customer information in a CSV file.

Breaking the Specification Down

Most programs that you will write can be broken down into smaller programs. In fact, you will want to do this to make your code more **modular**. This means that if you break the code down into small enough pieces that the pieces can be reused by other programs. We won't go that far in this example, but you should have an idea of how to do that by the end of this chapter.

Let's break the specification down a bit. Here are the tasks that occurred to me:

- Write a program to put all the chapters into one book file
- Figure out how to add a footer
- Find some way to turn the book file into a PDF
- Write a program that can email PDFs

Let's start with the first two items since they're related.

Turning Chapters Into a Book

I went the easy route and just created a Python list of the chapters in the order that I wanted them to appear in the book. I could have used Python's **glob** module to grab all the files in a folder that had the **.rst** extension, but then they might not have been in the right order.

Let's take a look at a simple example of what I did:

```
1 import os
2 import subprocess
3
4 chapters = ['chapter_1.rst',
5             'chapter_2.rst',
6             'chapter_3.rst']
7
8 def read_chapter(chapter):
9     """
10     Reads a chapter and returns the stream
11     """
12     path = os.path.join("data", chapter)
13     try:
14         with open(path) as chp_handler:
```

```
15         data = chp_handler.read()
16     except (IOError, OSError):
17         raise Exception("Unable to open chapter: %s" % chapter)
18
19     return data
20
21 def make_book(name="Mike", email_address="test@py.com"):
22     """
23     Creates Python 101 book
24     """
25     book_path = "output/python101.rst"
26     pdf_path = "output/python101.pdf"
27     page_break = """
28 .. raw:: pdf
29
30     PageBreak
31     """
32     footer = """
33 .. footer::
34
35     Copyright |copy| 2014 by Michael Driscoll, all rights reserved.
36
37     Licensed to %s <%s>
38
39 .. |copy| unicode:: 0xA9 .. copyright sign
40     """ % (name, email_address)
41     try:
42         with open(book_path, "w") as book:
43             book.write(footer + "\n")
44             for chapter in chapters:
45                 data = read_chapter(chapter)
46                 book.write(data)
47                 book.write("\n")
48                 book.write(page_break + "\n")
49     except:
50         print("Error writing book!")
51         raise
52
53     cmd = [r"C:\Python27\Scripts\rst2pdf.exe",
54            book_path, "-o", pdf_path]
55     subprocess.call(cmd)
56
```

```
57 if __name__ == "__main__":
58     make_book()
```

Let's break this down a bit. At the beginning of the script, we import two modules: `os` and `subprocess`. Then we create a really simple function that will open a chapter file that's located in the `data` folder, read the chapter and return the file's contents. The meat of the program is in the `make_book` function. Here we set up the names of the `RestructuredText` file and the PDF. Then we create a `page_break` variable and a `footer` variable. These two variables contain `RestructuredText` that we insert into the document. Right now, the program does not insert a customer's name or email, but will just use a couple of defaults instead.

Next we open the `RestructuredText` book document in `write` mode and write the footer text to it. Then we loop over the chapters, writing them to disk with a page break between each of them. Then at the very end of the script, we call a program called `rst2pdf` that will turn our new book file into a PDF. At this point, we have about a third of the program written. Now we just need to figure out how to read CSV file and email the PDF.

Reading the Customer CSV File

You may recall that we already covered a module that can help us in this quest back in Chapter 13. It's the `csv` module, of course! I won't reveal exactly what information Kickstarter has in their files, but we can create our own pretend version. Let's say it's something like this:

```
1 Mike Driscoll,test@py.com
2 Andy Hunter,andwa@hunter.org
3 Toby Mac,macdaddy@gotee.com
```

Let's create a quick function that can extract the name and email address from this file.

```
1 import csv
2
3 def read_csv(path):
4     """
5     try:
6         with open(path) as csv_file:
7             reader = csv.reader(csv_file)
8             for line in reader:
9                 name, email = line
10                print(name)
11                print(email)
12
13 except IOError:
```

```
13     print("Error reading file: %s" % path)
14     raise
15
16 if __name__ == "__main__":
17     path = "backers.csv"
18     read_csv(path)
```

All this script does is read the CSV file and print out the name and email address. Let's rename the function to **main** and have it call our **make_book** function:

```
1 def main(path):
2     """
3
4     try:
5         with open(path) as csv_file:
6             reader = csv.reader(csv_file)
7             for line in reader:
8                 name, email = line
9                 make_book(name, email)
10    except IOError:
11        print("Error reading file: %s" % path)
12        raise
13
14 if __name__ == "__main__":
15     main("backers.csv")
```

This will call the **make_book** function 3 times, so we will create the PDF 3 times and overwrite each time. You can check the file at the end of the run to see which name and email address is in the footer. This gets us two-thirds of the way there. All that's left is to figure out how to send the PDF out in an email. Fortunately, we covered that back in Chapter 17 when you learned about the **email** and **smtplib** modules!

Emailing the PDF

Let's spend a few minutes putting together a simple program that can email PDFs. Here's a fairly simple example:

```
1 import os
2 import smtplib
3
4 from email import encoders
5 from email.mime.text import MIMEText
6 from email.mime.base import MIMEBase
7 from email.mime.multipart import MIMEMultipart
8 from email.utils import formatdate
9
10 def send_email(email, pdf):
11     """
12     Send an email out
13     """
14     header0 = 'Content-Disposition'
15     header1 = 'attachment; filename="%s"' % os.path.basename(pdf)
16     header = header0, header1
17
18     host = "mail.server.com"
19     server = smtplib.SMTP(host)
20     subject = "Test email from Python"
21     to = email
22     from_addr = "test@pylib.com"
23     body_text = "Here is the Alpha copy of Python 101, Part I"
24
25     # create the message
26     msg = MIMEMultipart()
27     msg["From"] = from_addr
28     msg["Subject"] = subject
29     msg["Date"] = formatdate(localtime=True)
30     msg["To"] = email
31
32     msg.attach( MIMEText(body_text) )
33
34     attachment = MIMEBase('application', "octet-stream")
35     try:
36         with open(pdf, "rb") as fh:
37             data = fh.read()
38             attachment.set_payload( data )
39             encoders.encode_base64(attachment)
40             attachment.add_header(*header)
41             msg.attach(attachment)
42     except IOError:
```

```
43     msg = "Error opening attachment file %s" % file_to_attach
44     print(msg)
45
46     server.sendmail(from_addr, to, msg.as_string())
47
48 if __name__ == "__main__":
49     send_email("mike@example.org", "output/python101.pdf")
```

This is basically a modified example of the email attachment code from earlier in the book. This function is made to accept two arguments: an email address and a path to a PDF file. Technically, you could pass it any type of file and it would work. You will need to modify the `host` variable to point to your own SMTP server. I tested this on my machine with the proper host name and it worked great. Now we just need to integrate this code into the main code.

Putting it all Together

Now we get to put all the code together and see it as a whole. Here it is:

```
1 import csv
2 import os
3 import smtplib
4 import subprocess
5
6 from email import encoders
7 from email.mime.text import MIMEText
8 from email.mime.base import MIMEBase
9 from email.mime.multipart import MIMEMultipart
10 from email.utils import formatdate
11
12 chapters = ['chapter_1.rst',
13             'chapter_2.rst',
14             'chapter_3.rst']
15
16 def make_book(name="Mike", email_address="test@py.com"):
17     """
18     Creates Python 101 book
19     """
20     book_path = "output/python101.rst"
21     pdf_path = "output/python101.pdf"
22     page_break = """
23 .. raw:: pdf
```

```
24
25     PageBreak
26     """
27     footer = """
28 .. footer::
29
30     Copyright |copy| 2014 by Michael Driscoll, all rights reserved.
31
32     Licensed to %s <%s>
33
34 .. |copy| unicode:: 0xA9 .. copyright sign
35     """ % (name, email_address)
36     try:
37         with open(book_path, "w") as book:
38             book.write(footer + "\n")
39             for chapter in chapters:
40                 data = read_chapter(chapter)
41                 book.write(data)
42                 book.write("\n")
43                 book.write(page_break + "\n")
44     except:
45         print("Error writing book!")
46         raise
47
48     cmd = [r"C:\Python27\Scripts\rst2pdf.exe",
49            book_path, "-o", pdf_path]
50     subprocess.call(cmd)
51     return pdf_path
52
53 def read_chapter(chapter):
54     """
55     Reads a chapter and returns the stream
56     """
57     path = os.path.join("data", chapter)
58     try:
59         with open(path) as chp_handler:
60             data = chp_handler.read()
61     except (IOError, OSError):
62         raise Exception("Unable to open chapter: %s" % chapter)
63
64     return data
65
```

```
66 def send_email(email, pdf):
67     """
68     Send an email out
69     """
70     header0 = 'Content-Disposition'
71     header1 ='attachment; filename="%s"' % os.path.basename(pdf)
72     header = header0, header1
73
74     host = "mail.server.com"
75     server = smtplib.SMTP(host)
76     subject = "Test email from Python"
77     to = email
78     from_addr = "test@pylib.com"
79     body_text = "Here is the Alpha copy of Python 101, Part I"
80
81     # create the message
82     msg = MIMEMultipart()
83     msg["From"] = from_addr
84     msg["Subject"] = subject
85     msg["Date"] = formatdate(localtime=True)
86     msg["To"] = email
87
88     msg.attach( MIMEText(body_text) )
89
90     attachment = MIMEBase('application', "octet-stream")
91     try:
92         with open(pdf, "rb") as fh:
93             data = fh.read()
94             attachment.set_payload( data )
95             encoders.encode_base64(attachment)
96             attachment.add_header(*header)
97             msg.attach(attachment)
98     except IOError:
99         msg = "Error opening attachment file %s" % file_to_attach
100        print(msg)
101
102    server.sendmail(from_addr, to, msg.as_string())
103
104 def main(path):
105     """
106     try:
107         with open(path) as csv_file:
```

```
108     reader = csv.reader(csv_file)
109     for line in reader:
110         name, email = line
111         pdf = make_book(name, email)
112         send_email(email, pdf)
113     except IOError:
114         print("Error reading file: %s" % path)
115         raise
116
117 if __name__ == "__main__":
118     main("backers.csv")
```

Let's go over the minor changes. First off, we had to change the `make_book` function so that it returned the PDF path. Then we changed the `main` function so that it took that path and passed it to the `send_email` function along with the email address. Two changes and we're done!

Wrapping Up

Now you have the knowledge to take apart a new program and turn it into a set of smaller tasks. Then you can join the smaller programs into one large program. There are lots of different ways we could have done this. For example, we could have left each piece in its own file or module. Then we could import each piece into a different script and run them that way. That would be a good exercise for to try on your own.

When I did this, I ran into issues with my email host. Most email hosts will not let you send out lots of emails per hour or even make many connections at once. You can find a few services on the internet that you can use instead of your own internet provider that may work better for you.

I hope you found this chapter helpful in your own endeavours. Thanks for reading!

Tkinter 8.5 reference: a GUI for Python



John W. Shipman

2013-12-31 17:59

Abstract

Describes the *Tkinter* widget set for constructing graphical user interfaces (GUIs) in the Python programming language. Includes coverage of the *ttk* themed widgets.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

1. A cross-platform graphical user interface builder for Python	3
2. A minimal application	4
3. Definitions	5
4. Layout management	5
4.1. The <code>.grid()</code> method	6
4.2. Other grid management methods	7
4.3. Configuring column and row sizes	7
4.4. Making the root window resizeable	8
5. Standard attributes	9
5.1. Dimensions	9
5.2. The coordinate system	10
5.3. Colors	10
5.4. Type fonts	10
5.5. Anchors	12
5.6. Relief styles	12
5.7. Bitmaps	12
5.8. Cursors	13
5.9. Images	14
5.10. Geometry strings	15
5.11. Window names	16
5.12. Cap and join styles	16
5.13. Dash patterns	17
5.14. Matching stipple patterns	17
6. Exception handling	18
7. The <code>Button</code> widget	18
8. The <code>Canvas</code> widget	20
8.1. <code>Canvas</code> coordinates	22
8.2. The <code>Canvas</code> display list	22
8.3. <code>Canvas</code> object IDs	22

¹ <http://www.nmt.edu/tcc/help/pubs/tkinter/>

² <http://www.nmt.edu/tcc/help/pubs/tkinter/tkinter.pdf>

8.4. Canvas tags	22
8.5. Canvas <i>tagOrId</i> arguments	22
8.6. Methods on Canvas widgets	22
8.7. Canvas arc objects	28
8.8. Canvas bitmap objects	29
8.9. Canvas image objects	30
8.10. Canvas line objects	30
8.11. Canvas oval objects	32
8.12. Canvas polygon objects	33
8.13. Canvas rectangle objects	35
8.14. Canvas text objects	37
8.15. Canvas window objects	38
9. The Checkbutton widget	38
10. The Entry widget	41
10.1. Scrolling an Entry widget	45
10.2. Adding validation to an Entry widget	45
11. The Frame widget	47
12. The Label widget	48
13. The LabelFrame widget	50
14. The Listbox widget	52
14.1. Scrolling a Listbox widget	56
15. The Menu widget	56
15.1. Menu item creation (<i>option</i>) options	59
15.2. Top-level menus	60
16. The Menubutton widget	61
17. The Message widget	63
18. The OptionMenu widget	64
19. The PanedWindow widget	65
19.1. PanedWindow child configuration options	67
20. The Radiobutton widget	68
21. The Scale widget	71
22. The Scrollbar widget	74
22.1. The Scrollbar <i>command</i> callback	77
22.2. Connecting a Scrollbar to another widget	77
23. The Spinbox widget	78
24. The Text widget	82
24.1. Text widget indices	84
24.2. Text widget marks	86
24.3. Text widget images	86
24.4. Text widget windows	87
24.5. Text widget tags	87
24.6. Setting tabs in a Text widget	87
24.7. The Text widget undo/redo stack	88
24.8. Methods on Text widgets	88
25. Toplevel: Top-level window methods	95
26. Universal widget methods	97
27. Standardizing appearance	105
27.1. How to name a widget class	106
27.2. How to name a widget instance	107
27.3. Resource specification lines	107
27.4. Rules for resource matching	108
28. ttk: Themed widgets	108

28.1. Importing <code>ttk</code>	109
28.2. The <code>ttk</code> widget set	110
29. <code>ttk.Button</code>	110
30. <code>ttk.Checkbutton</code>	112
31. <code>ttk.Combobox</code>	115
32. <code>ttk.Entry</code>	116
33. <code>ttk.Frame</code>	118
34. <code>ttk.Label</code>	119
35. <code>ttk.LabelFrame</code>	122
36. <code>ttk.Menubutton</code>	124
37. <code>ttk.Notebook</code>	126
37.1. Virtual events for the <code>ttk.Notebook</code> widget	128
38. <code>ttk.PanedWindow</code>	129
39. <code>ttk.Progressbar</code>	130
40. <code>ttk.Radiobutton</code>	131
41. <code>ttk.Scale</code>	133
42. <code>ttk.Scrollbar</code>	135
43. <code>ttk.Separator</code>	137
44. <code>ttk.Sizegrip</code>	137
45. <code>ttk.Treeview</code>	137
45.1. Virtual events for the <code>ttk.Treeview</code> widget	145
46. Methods common to all <code>ttk</code> widgets	145
46.1. Specifying widget states in <code>ttk</code>	146
47. Customizing and creating <code>ttk</code> themes and styles	146
48. Finding and using <code>ttk</code> themes	147
49. Using and customizing <code>ttk</code> styles	147
50. The <code>ttk</code> element layer	149
50.1. <code>ttk</code> layouts: Structuring a style	149
50.2. <code>ttk</code> style maps: dynamic appearance changes	151
51. Connecting your application logic to the widgets	153
52. Control variables: the values behind the widgets	153
53. Focus: routing keyboard input	155
53.1. Focus in <code>ttk</code> widgets	156
54. Events	157
54.1. Levels of binding	157
54.2. Event sequences	158
54.3. Event types	158
54.4. Event modifiers	160
54.5. Key names	160
54.6. Writing your handler: The <code>Event</code> class	162
54.7. The extra arguments trick	164
54.8. Virtual events	165
55. Pop-up dialogs	165
55.1. The <code>tkMessageBox</code> dialogs module	165
55.2. The <code>tkFileDialog</code> module	167
55.3. The <code>tkColorChooser</code> module	168

1. A cross-platform graphical user interface builder for Python

Tkinter is a GUI (graphical user interface) widget set for Python. This document was written for Python 2.7 and *Tkinter* 8.5 running in the X Window system under Linux. Your version may vary.

Pertinent references:

- Fredrik Lundh³, who wrote *Tkinter*⁴, has two versions of his *An Introduction to Tkinter*: a more complete 1999 version⁵ and a 2005 version⁶ that presents a few newer features.
- *Python 2.7 quick reference*⁷: general information about the Python language.
- For an example of a sizeable working application (around 1000 lines of code), see *huey: A color and font selection tool*⁸. The design of this application demonstrates how to build your own compound widgets.

We'll start by looking at the visible part of *Tkinter*: creating the widgets and arranging them on the screen. Later we will talk about how to connect the face—the “front panel”—of the application to the logic behind it.

2. A minimal application

Here is a trivial *Tkinter* program containing only a Quit button:

```
#!/usr/bin/env python      1
import Tkinter as tk       2

class Application(tk.Frame):
    def __init__(self, master=None):
        tk.Frame.__init__(self, master)  3
        self.grid()                   4
        self.createWidgets()

    def createWidgets(self):
        self.quitButton = tk.Button(self, text='Quit', 5
                                   command=self.quit)
        self.quitButton.grid()         6

app = Application()          7
app.master.title('Sample application') 8
app.mainloop()                9
                                10
```

- 1 This line makes the script self-executing, assuming that your system has Python correctly installed.
- 2 This line imports the *Tkinter* module into your program's namespace, but renames it as `tk`.
- 3 Your application class must inherit from *Tkinter*'s `Frame` class.
- 4 Calls the constructor for the parent class, `Frame`.
- 5 Necessary to make the application actually appear on the screen.
- 6 Creates a button labeled “Quit”.
- 7 Places the button on the application.
- 8 The main program starts here by instantiating the `Application` class.
- 9 This method call sets the title of the window to “Sample application”.
- 10 Starts the application's main loop, waiting for mouse and keyboard events.

³ <http://www.pythonware.com/library/tkinter/introduction/>
⁴ <http://effbot.org/tkinterbook/>

⁵ <http://www.nmt.edu/tcc/help/pubs/python/>
⁶ <http://www.nmt.edu/tcc/help/lang/python/examples/huey/>

3. Definitions

Before we proceed, let's define some of the common terms.

window

This term has different meanings in different contexts, but in general it refers to a rectangular area somewhere on your display screen.

top-level window

A window that exists independently on your screen. It will be decorated with the standard frame and controls for your system's desktop manager. You can move it around on your desktop. You can generally resize it, although your application can prevent this

widget

The generic term for any of the building blocks that make up an application in a graphical user interface. Examples of widgets: buttons, radiobuttons, text fields, frames, and text labels.

frame

In *Tkinter*, the `Frame` widget is the basic unit of organization for complex layouts. A frame is a rectangular area that can contain other widgets.

child, parent

When any widget is created, a *parent-child* relationship is created. For example, if you place a text label inside a frame, the frame is the parent of the label.

4. Layout management

Later we will discuss the widgets, the building blocks of your GUI application. How do widgets get arranged in a window?

Although there are three different “geometry managers” in *Tkinter*, the author strongly prefers the `.grid()` geometry manager for pretty much everything. This manager treats every window or frame as a table—a gridwork of rows and columns.

- A *cell* is the area at the intersection of one row and one column.
- The width of each column is the width of the widest cell in that column.
- The height of each row is the height of the largest cell in that row.
- For widgets that do not fill the entire cell, you can specify what happens to the extra space. You can either leave the extra space outside the widget, or stretch the widget to fit it, in either the horizontal or vertical dimension.
- You can combine multiple cells into one larger area, a process called *spanning*.

When you create a widget, it does not appear until you register it with a geometry manager. Hence, construction and placing of a widget is a two-step process that goes something like this:

```
self.thing = tk.Constructor(parent, ...)
self.thing.grid(...)
```

where *Constructor* is one of the widget classes like `Button`, `Frame`, and so on, and *parent* is the parent widget in which this child widget is being constructed. All widgets have a `.grid()` method that you can use to tell the geometry manager where to put it.

4.1. The `.grid()` method

To display a widget `w` on your application screen:

```
w.grid(option=value, ...)
```

This method registers a widget `w` with the grid geometry manager—if you don't do this, the widget will exist internally, but it will not be visible on the screen. For the options, see Table 1, “Arguments of the `.grid()` geometry manager” (p. 6).

Table 1. Arguments of the `.grid()` geometry manager

<code>column</code>	The column number where you want the widget gridded, counting from zero. The default value is zero.
<code>columnspan</code>	Normally a widget occupies only one cell in the grid. However, you can grab multiple cells of a row and merge them into one larger cell by setting the <code>columnspan</code> option to the number of cells. For example, <code>w.grid(row=0, column=2, columnspan=3)</code> would place widget <code>w</code> in a cell that spans columns 2, 3, and 4 of row 0.
<code>in_</code>	To register <code>w</code> as a child of some widget <code>w₂</code> , use <code>in_=w₂</code> . The new parent <code>w₂</code> must be a descendant of the <code>parent</code> widget used when <code>w</code> was created.
<code>ipadx</code>	Internal x padding. This dimension is added inside the widget inside its left and right sides.
<code>ipady</code>	Internal y padding. This dimension is added inside the widget inside its top and bottom borders.
<code>padx</code>	External x padding. This dimension is added to the left and right outside the widget.
<code>pady</code>	External y padding. This dimension is added above and below the widget.
<code>row</code>	The row number into which you want to insert the widget, counting from 0. The default is the next higher-numbered unoccupied row.
<code>rowspan</code>	Normally a widget occupies only one cell in the grid. You can grab multiple adjacent cells of a column, however, by setting the <code>rowspan</code> option to the number of cells to grab. This option can be used in combination with the <code>columnspan</code> option to grab a block of cells. For example, <code>w.grid(row=3, column=2, rowspan=4, columnspan=5)</code> would place widget <code>w</code> in an area formed by merging 20 cells, with row numbers 3–6 and column numbers 2–6.
<code>sticky</code>	This option determines how to distribute any extra space within the cell that is not taken up by the widget at its natural size. See below.

- If you do not provide a `sticky` attribute, the default behavior is to center the widget in the cell.
- You can position the widget in a corner of the cell by using `sticky=tk.NE` (top right), `tk.SE` (bottom right), `tk.SW` (bottom left), or `tk.NW` (top left).
- You can position the widget centered against one side of the cell by using `sticky=tk.N` (top center), `tk.E` (right center), `tk.S` (bottom center), or `tk.W` (left center).
- Use `sticky=tk.N+tk.S` to stretch the widget vertically but leave it centered horizontally.
- Use `sticky=tk.E+tk.W` to stretch it horizontally but leave it centered vertically.
- Use `sticky=tk.N+tk.E+tk.S+tk.W` to stretch the widget both horizontally and vertically to fill the cell.

- The other combinations will also work. For example, `sticky=tk.N+tk.S+tk.W` will stretch the widget vertically and place it against the west (left) wall.

4.2. Other grid management methods

These grid-related methods are defined on all widgets:

`w.grid_bbox(column=None, row=None, col2=None, row2=None)`

Returns a 4-tuple describing the bounding box of some or all of the grid system in widget `w`. The first two numbers returned are the `x` and `y` coordinates of the upper left corner of the area, and the second two numbers are the width and height.

If you pass in `column` and `row` arguments, the returned bounding box describes the area of the cell at that column and row. If you also pass in `col2` and `row2` arguments, the returned bounding box describes the area of the grid from columns `column` to `col2` inclusive, and from rows `row` to `row2` inclusive.

For example, `w.grid_bbox(0, 0, 1, 1)` returns the bounding box of four cells, not one.

`w.grid_forget()`

This method makes widget `w` disappear from the screen. It still exists, it just isn't visible. You can use `.grid()` it to make it appear again, but it won't remember its grid options.

`w.grid_info()`

Returns a dictionary whose keys are `w`'s option names, with the corresponding values of those options.

`w.grid_location(x, y)`

Given a coordinates `(x, y)` relative to the containing widget, this method returns a tuple `(col, row)` describing what cell of `w`'s grid system contains that screen coordinate.

`w.grid_propagate()`

Normally, all widgets *propagate* their dimensions, meaning that they adjust to fit the contents. However, sometimes you want to force a widget to be a certain size, regardless of the size of its contents. To do this, call `w.grid_propagate(0)` where `w` is the widget whose size you want to force.

`w.grid_remove()`

This method is like `.grid_forget()`, but its grid options are remembered, so if you `.grid()` it again, it will use the same grid configuration options.

`w.grid_size()`

Returns a 2-tuple containing the number of columns and the number of rows, respectively, in `w`'s grid system.

`w.grid_slaves(row=None, column=None)`

Returns a list of the widgets managed by widget `w`. If no arguments are provided, you will get a list of all the managed widgets. Use the `row=` argument to select only the widgets in one row, or the `column=` argument to select only the widgets in one column.

4.3. Configuring column and row sizes

Unless you take certain measures, the width of a grid column inside a given widget will be equal to the width of its widest cell, and the height of a grid row will be the height of its tallest cell. The `sticky` attribute on a widget controls only where it will be placed if it doesn't completely fill the cell.

If you want to override this automatic sizing of columns and rows, use these methods on the *parent* widget `w` that contains the grid layout:

`w.columnconfigure(N, option=value, ...)`

In the grid layout inside widget *w*, configure column *N* so that the given *option* has the given *value*. For options, see the table below.

`w.rowconfigure(N, option=value, ...)`

In the grid layout inside widget *w*, configure row *N* so that the given *option* has the given *value*. For options, see the table below.

Here are the options used for configuring column and row sizes.

Table 2. Column and row configuration options for the `.grid()` geometry manager

<code>minsize</code>	The column or row's minimum size in pixels. If there is nothing in the given column or row, it will not appear, even if you use this option.
<code>pad</code>	A number of pixels that will be added to the given column or row, over and above the largest cell in the column or row.
<code>weight</code>	To make a column or row stretchable, use this option and supply a value that gives the relative weight of this column or row when distributing the extra space. For example, if a widget <i>w</i> contains a grid layout, these lines will distribute three-fourths of the extra space to the first column and one-fourth to the second column: <pre>w.columnconfigure(0, weight=3) w.columnconfigure(1, weight=1)</pre> If this option is not used, the column or row will not stretch.

4.4. Making the root window resizable

Do you want to let the user resize your entire application window, and distribute the extra space among its internal widgets? This requires some operations that are not obvious.

It's necessary to use the techniques for row and column size management, described in Section 4.3, "Configuring column and row sizes" (p. 7), to make your `Application` widget's grid stretchable. However, that alone is not sufficient.

Consider the trivial application discussed in Section 2, "A minimal application" (p. 4), which contains only a `Quit` button. If you run this application, and resize the window, the button stays the same size, centered within the window.

Here is a replacement version of the `__createWidgets()` method in the minimal application. In this version, the `Quit` button always fills all the available space.

```
def createWidgets(self):
    top=self.winfo_toplevel()                      1
    top.rowconfigure(0, weight=1)                    2
    top.columnconfigure(0, weight=1)                  3
    self.rowconfigure(0, weight=1)                   4
    self.columnconfigure(0, weight=1)                 5
    self.quit = Button(self, text='Quit', command=self.quit)
    self.quit.grid(row=0, column=0,                  6
                   sticky=tk.N+tk.S+tk.E+tk.W)
```

1 The "top level window" is the outermost window on the screen. However, this window is not your `Application` window—it is the *parent* of the `Application` instance. To get the top-level window,

call the `.winfo_toplevel()` method on any widget in your application; see Section 26, “Universal widget methods” (p. 97).

- 2 This line makes row 0 of the top level window's grid stretchable.
- 3 This line makes column 0 of the top level window's grid stretchable.
- 4 Makes row 0 of the `Application` widget's grid stretchable.
- 5 Makes column 0 of the `Application` widget's grid stretchable.
- 6 The argument `sticky=tk.N+tk.S+tk.E+tk.W` makes the button expand to fill its cell of the grid.

There is one more change that must be made. In the constructor, change the second line as shown:

```
def __init__(self, master=None):  
    tk.Frame.__init__(self, master)  
    self.grid(sticky=tk.N+tk.S+tk.E+tk.W)  
    self.createWidgets()
```

The argument `sticky=tk.N+tk.S+tk.E+tk.aW` to `self.grid()` is necessary so that the `Application` widget will expand to fill its cell of the top-level window's grid.

5. Standard attributes

Before we look at the widgets, let's take a look at how some of their common attributes—such as sizes, colors and fonts—are specified.

- Each widget has a set of *options* that affect its appearance and behavior—attributes such as fonts, colors, sizes, text labels, and such.
- You can specify options when calling the widget's constructor using keyword arguments such as `text='PANIC!'` or `height=20`.
- After you have created a widget, you can later change any option by using the widget's `.config()` method. You can retrieve the current setting of any option by using the widget's `.cget()` method. See Section 26, “Universal widget methods” (p. 97) for more on these methods.

5.1. Dimensions

Various lengths, widths, and other dimensions of widgets can be described in many different units.

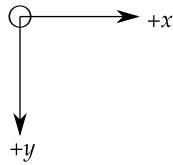
- If you set a dimension to an integer, it is assumed to be in pixels.
- You can specify units by setting a dimension to a string containing a number followed by:

Table 3. Dimensional units

c	Centimeters
i	Inches
m	Millimeters
p	Printer's points (about 1/72")

5.2. The coordinate system

As in most contemporary display systems, the origin of each coordinate system is at its upper left corner, with the x coordinate increasing toward the right, and the y coordinate increasing toward the bottom:



The base unit is the pixel, with the top left pixel having coordinates (0,0). Coordinates that you specify as integers are always expressed in pixels, but any coordinate may be specified as a dimensioned quantity; see Section 5.1, “Dimensions” (p. 9).

5.3. Colors

There are two general ways to specify colors in *Tkinter*.

- You can use a string specifying the proportion of red, green, and blue in hexadecimal digits:

#rgb	Four bits per color
#rrggb	Eight bits per color
#rrrrgggbbb	Twelve bits per color

For example, '#fff' is white, '#000000' is black, '#000fffff' is pure green, and '#00ffff' is pure cyan (green plus blue).

- You can also use any locally defined standard color name. The colors 'white', 'black', 'red', 'green', 'blue', 'cyan', 'yellow', and 'magenta' will always be available. Other names may work, depending on your local installation.

5.4. Type fonts

Depending on your platform, there may be up to three ways to specify type style.

- As a tuple whose first element is the font family, followed by a size (in points if positive, in pixels if negative), optionally followed by a string containing one or more of the style modifiers **bold**, **italic**, **underline**, and **overstrike**.

Examples: ('Helvetica', '16') for a 16-point Helvetica regular; ('Times', '24', 'bold italic') for a 24-point Times bold italic. For a 20-pixel Times bold font, use ('Times', -20, 'bold').

- You can create a “font object” by importing the **tkFont** module and using its **Font** class constructor:

```
import tkFont  
  
font = tkFont.Font(option, ...)
```

where the options include:

family	The font family name as a string.
--------	-----------------------------------

size	The font height as an integer in points. To get a font <i>n</i> pixels high, use <code>-n</code> .
weight	'bold' for boldface, 'normal' for regular weight.
slant	'italic' for italic, 'roman' for unslanted.
underline	1 for underlined text, 0 for normal.
overstrike	1 for overstruck text, 0 for normal.

For example, to get a 36-point bold Helvetica italic face:

```
helv36 = tkFont.Font(family='Helvetica',
                      size=36, weight='bold')
```

- If you are running under the X Window System, you can use any of the X font names. For example, the font named '`lucidatypewriter-medium-r-*-*-140-*-*-*-*`' is a good fixed-width font for onscreen use. Use the `xfontsel` program to help you select pleasing fonts.

To get a list of all the families of fonts available on your platform, call this function:

```
tkFont.families()
```

The return value is a list of strings. *Note:* You must create your root window before calling this function.

These methods are defined on all `Font` objects:

.actual(*option=None*)

If you pass no arguments, you get back a dictionary of the font's actual attributes, which may differ from the ones you requested. To get back the value of an attribute, pass its name as an argument.

.cget(*option*)

Returns the value of the given *option*.

.configure(*option, ...*)

Use this method to change one or more options on a font. For example, if you have a `Font` object called `titleFont`, if you call `titleFont.configure(family='times', size=18)`, that font will change to 18pt Times and any widgets that use that font will change too.

.copy()

Returns a copy of a `Font` object.

.measure(*text*)

Pass this method a string, and it will return the number of pixels of width that string will take in the font. Warning: some slanted characters may extend outside this area.

.metrics(*option*)

If you call this method with no arguments, it returns a dictionary of all the *font metrics*. You can retrieve the value of just one metric by passing its name as an argument. Metrics include:

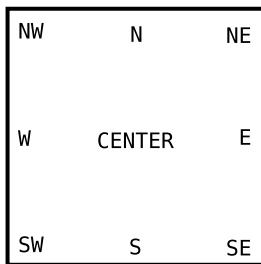
ascent	Number of pixels of height between the baseline and the top of the highest ascender.
descent	Number of pixels of height between the baseline and the bottom of the lowest ascender.
fixed	This value is 0 for a variable-width font and 1 for a monospaced font.
linespace	Number of pixels of height total. This is the leading of type set solid in the given font.

5.5. Anchors

The *Tkinter* module defines a number of *anchor* constants that you can use to control where items are positioned relative to their context. For example, anchors can specify where a widget is located inside a frame when the frame is bigger than the widget.

These constants are given as compass points, where north is up and west is to the left. We apologize to our Southern Hemisphere readers for this Northern Hemisphere chauvinism⁷.

The anchor constants are shown in this diagram:



For example, if you create a small widget inside a large frame and use the `anchor=tk.SE` option, the widget will be placed in the bottom right corner of the frame. If you used `anchor=tk.N` instead, the widget would be centered along the top edge.

Anchors are also used to define where text is positioned relative to a reference point. For example, if you use `tk.CENTER` as a text anchor, the text will be centered horizontally and vertically around the reference point. Anchor `tk.NW` will position the text so that the reference point coincides with the northwest (top left) corner of the box containing the text. Anchor `tk.W` will center the text vertically around the reference point, with the left edge of the text box passing through that point, and so on.

5.6. Relief styles

The *relief style* of a widget refers to certain simulated 3-D effects around the outside of the widget. Here is a screen shot of a row of buttons exhibiting all the possible relief styles:



The width of these borders depends on the `borderwidth` option of the widget. The above graphic shows what they look like with a 5-pixel border; the default border width is 2.

5.7. Bitmaps

For `bitmap` options in widgets, these bitmaps are guaranteed to be available:



⁷ <http://flourish.org/upsidedownmap/>

The graphic above shows Button widgets bearing the standard bitmaps. From left to right, they are 'error', 'gray75', 'gray50', 'gray25', 'gray12', 'hourglass', 'info', 'questhead', 'question', and 'warning'.

You can use your own bitmaps. Any file in .xbm (X bit map) format will work. In place of a standard bitmap name, use the string '@' followed by the pathname of the .xbm file.

5.8. Cursors

There are quite a number of different mouse cursors available. Their names and graphics are shown here. The exact graphic may vary according to your operating system.

Table 4. Values of the cursor option

	arrow		man
	based_arrow_down		middlebutton
	based_arrow_up		mouse
	boat		pencil
	bogosity		pirate
	bottom_left_corner		plus
	bottom_right_corner		question_arrow
	bottom_side		right_ptr
	bottom_tee		right_side
	box_spiral		right_tee
	center_ptr		rightbutton
	circle		rtl_logo
	clock		sailboat
	coffee_mug		sb_down_arrow
	cross		sb_h_double_arrow
	cross_reverse		sb_left_arrow
	crosshair		sb_right_arrow
	diamond_cross		sb_up_arrow
	dot		sb_v_double_arrow

 dotbox	 shuttle
 double_arrow	 sizing
 draft_large	 spider
 draft_small	 spraycan
 draped_box	 star
 exchange	 target
 fleur	 tcross
 gobblor	 top_left_arrow
 gumby	 top_left_corner
 hand1	 top_right_corner
 hand2	 top_side
 heart	 top_tee
 icon	 trek
 iron_cross	 ul_angle
 left_ptr	 umbrella
 left_side	 ur_angle
 left_tee	 watch
 leftbutton	 xterm
 ll_angle	 X_cursor
 lr_angle	

5.9. Images

There are three general methods for displaying graphic images in your *Tkinter* application.

- To display bitmap (two-color) images in the `.xbm` format, refer to Section 5.9.1, “The `BitmapImage` class” (p. 15).
- To display full-color images in the `.gif`, `.pgm`, or `.ppm` format, see Section 5.9.2, “The `PhotoImage` class” (p. 15).

- The Python Imaging Library (PIL) supports images in a much wider variety of formats. Its `ImageTk` class is specifically designed for displaying images within `Tkinter` applications. See the author's companion document for PIL documentation: *Python Imaging Library (PIL) quick reference*⁸.

5.9.1. The `BitmapImage` class

To display a two-color image in the `.xbm` format, you will need this constructor:

```
tk.BitmapImage(file=f[, background=b][, foreground=c])
```

where `f` is the name of the `.xbm` image file.

Normally, foreground (1) bits in the image will be displayed as black pixels, and background (0) bits in the image will be transparent. To change this behavior, use the optional `background=b` option to set the background to color `b`, and the optional `foreground=c` option to set the foreground to color `c`. For color specification, see Section 5.3, “Colors” (p. 10).

This constructor returns a value that can be used anywhere `Tkinter` expects an image. For example, to display an image as a label, use a `Label` widget (see Section 12, “The `Label` widget” (p. 48)) and supply the `BitmapImage` object as the value of the `image` option:

```
logo = tk.BitmapImage('logo.xbm', foreground='red')
Label(image=logo).grid()
```

5.9.2. The `PhotoImage` class

To display a color image in `.gif`, `.pgm`, or `.ppm` format, you will need this constructor:

```
tk.PhotoImage(file=f)
```

where `f` is the name of the image file. The constructor returns a value that can be used anywhere `Tkinter` expects an image.

5.10. Geometry strings

A *geometry string* is a standard way of describing the size and location of a top-level window on a desktop.

A geometry string has this general form:

```
'wxh±x±y'
```

where:

- The `w` and `h` parts give the window width and height in pixels. They are separated by the character '`x`'.
- If the next part has the form `+x`, it specifies that the left side of the window should be `x` pixels from the left side of the desktop. If it has the form `-x`, the right side of the window is `x` pixels from the right side of the desktop.
- If the next part has the form `+y`, it specifies that the top of the window should be `y` pixels below the top of the desktop. If it has the form `-y`, the bottom of the window will be `y` pixels above the bottom edge of the desktop.

⁸ <http://www.nmt.edu/tcc/help/pubs/pil/>

For example, a window created with `geometry='120x50-0+20'` would be 120 pixels wide by 50 pixels high, and its top right corner will be along the right edge of the desktop and 20 pixels below the top edge.

5.11. Window names

The term *window* describes a rectangular area on the desktop.

- A *top-level* or *root* window is a window that has an independent existence under the window manager. It is decorated with the window manager's decorations, and can be moved and resized independently. Your application can use any number of top-level windows.
- The term “window” also applies to any widget that is part of a top-level window.

Tkinter names all these windows using a hierarchical *window path name*.

- The root window's name is '`.'`'.
- Child windows have names of the form '`.n`', where *n* is some integer in string form. For example, a window named '`.135932060`' is a child of the root window ('`.'`).
- Child windows within child windows have names of the form '`p.n`' where *p* is the name of the parent window and *n* is some integer. For example, a window named '`.135932060.137304468`' has parent window '`.135932060`', so it is a grandchild of the root window.
- The *relative name* of a window is the part past the last '`.'`' in the path name. To continue the previous example, the grandchild window has a relative name '`137304468`'.

To get the path name for a widget *w*, use `str(w)`.

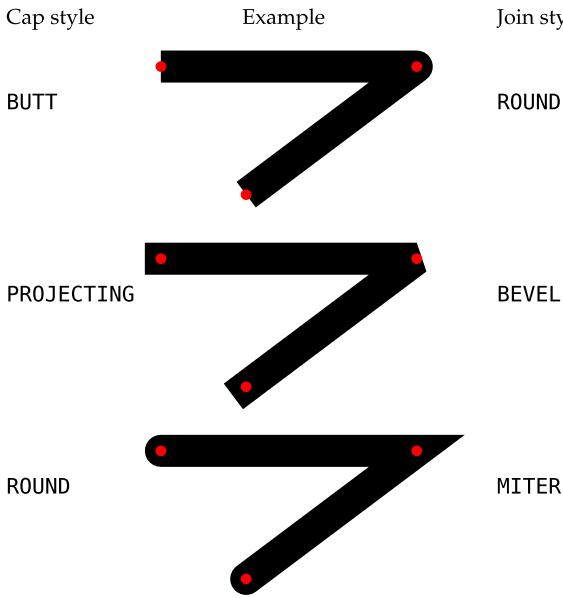
See also Section 26, “Universal widget methods” (p. 97) for methods you can use to operate on window names, especially the `.winfo_name`, `.winfo_parent`, and `.winfo_pathname` methods.

5.12. Cap and join styles

For pleasant and effective rendering of diagrams, sometimes it is a good idea to pay attention to cap and join styles.

- The *cap style* of a line is the shape of the end of the line. Styles are:
 - `tk.BUTT`: The end of the line is cut off square at a line that passes through the endpoint.
 - `tk.PROJECTING`: The end of the line is cut off square, but the cut line projects past the endpoint a distance equal to half the line's width.
 - `tk.ROUND`: The end describes a semicircle centered on the endpoint.
- The *join style* describes the shape where two line segments meet at an angle.
 - `tk.ROUND`: The join is a circle centered on the point where the adjacent line segments meet.
 - `tk.BEVEL`: A flat facet is drawn at an angle intermediate between the angles of the adjacent lines.
 - `tk.MITER`: The edges of the adjacent line segments are continued to meet at a sharp point.

This illustration shows how *Tkinter*'s cap and join options work with a line made of two connected line segments. Small red circles show the location of the points that define this line.



5.13. Dash patterns

A number of widgets allow you to specify a dashed outline. The `dash` and `dashoffset` options give you fine control over the exact pattern of the dashes.

`dash`

This option is specified as a tuple of integers. The first integer specifies how many pixels should be drawn. The second integer specifies how many pixels should be skipped before starting to draw again, and so on. When all the integers in the tuple are exhausted, they are reused in the same order until the border is complete.

For example, `dash=(3, 5)` produces alternating 3-pixel dashes separated by 5-pixel gaps. A value of `dash=(7, 1, 1, 1)` produces a dash-and-dot pattern, with the dash seven times as long as the dot or the gaps around the dot. A value of `dash=(5,)` produces alternating five-pixel dashes and five-pixel gaps.

`dashoff`

To start the dash pattern in a different point of cycle instead of at the beginning, use an option of `dashoff=n`, where *n* is the number of pixels to skip at the beginning of the pattern.

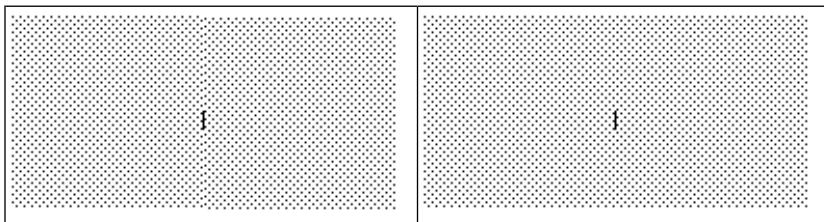
For example, for options `dash=(5, 1, 2, 1)` and `dashoff=3`, the first pattern produced will be: 2 on, 1 off, 2 on, and 1 off. Subsequent patterns will be 5 on, 1 off, 2 on, and 1 off. Here is a screen shot of a line drawn with this combination of options:



5.14. Matching stipple patterns

This may seem like an incredibly picky style point, but if you draw a graphic that has two objects with stippled patterns, a real professional will make sure that the patterns align along their boundary.

Here is an example. The left-hand screen shot shows two adjacent 100×100 squares stippled with the “gray12” pattern, but the right-hand square is offset vertically by one pixel. The short black line in the center of the figure is drawn along the boundary of the two figures.



The second screen shot is the same, except that the two 100×100 squares have their stipple patterns lined up.

In practice, this arises in two situations. The alignment of large stippled areas is controlled by an option named `offset`. For figures with stippled outlines, the `outlineoffset` option controls their alignment. Both options have values of one of these forms:

- ' x, y ': Offset the stipple patterns by this x and y value relative to the top-level window or to the canvas's origin.
- '# x, y ': For objects on a canvas, use offset x and y relative to the top-level window.
- `tk.NE, tk.SE, tk.SW, tk.NW`: Align a corner of the stipple pattern with the corresponding corner of the containing object. For example, `tk.NE` means that the top left corner of the stipple pattern coincides with the top left corner of the area to be stippled.
- `tk.N, tk.E, tk.S, tk.W`: Align the stipple pattern with the center of one side of the containing object. For example, `tk.E` means the center of the stipple pattern will coincide with the center of the right side of the area to be stippled.
- `tk.CENTER`: Align the center of the stipple pattern with the center of the containing object.

6. Exception handling

The exception raised by most programming errors is `tk.TclError`.

7. The Button widget

To create a pushbutton in a top-level window or frame named *parent*:

```
w = tk.Button(parent, option=value, ...)
```

The constructor returns the new **Button** widget. Its options include:

Table 5. Button widget options

<code>activebackground</code>	Background color when the button is under the cursor.
<code>activeforeground</code>	Foreground color when the button is under the cursor.
<code>anchor</code>	Where the text is positioned on the button. See Section 5.5, “Anchors” (p. 12). For example, <code>anchor=tk.NE</code> would position the text at the top right corner of the button.

<code>bd</code> or <code>borderwidth</code>	Width of the border around the outside of the button; see Section 5.1, “Dimensions” (p. 9). The default is two pixels.
<code>bg</code> or <code>background</code>	Normal background color.
<code>bitmap</code>	Name of one of the standard bitmaps to display on the button (instead of text).
<code>command</code>	Function or method to be called when the button is clicked.
<code>cursor</code>	Selects the cursor to be shown when the mouse is over the button.
<code>default</code>	<code>tk.NORMAL</code> is the default; use <code>tk.DISABLED</code> if the button is to be initially disabled (grayed out, unresponsive to mouse clicks).
<code>disabledforeground</code>	Foreground color used when the button is disabled.
<code>fg</code> or <code>foreground</code>	Normal foreground (text) color.
<code>font</code>	Text font to be used for the button's label.
<code>height</code>	Height of the button in text lines (for textual buttons) or pixels (for images).
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus.
<code>highlightcolor</code>	The color of the focus highlight when the widget has focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>image</code>	Image to be displayed on the button (instead of text).
<code>justify</code>	How to show multiple text lines: <code>tk.LEFT</code> to left-justify each line; <code>tk.CENTER</code> to center them; or <code>tk.RIGHT</code> to right-justify.
<code>overrelief</code>	The relief style to be used while the mouse is on the button; default relief is <code>tk.RAISED</code> . See Section 5.6, “Relief styles” (p. 12).
<code>padx</code>	Additional padding left and right of the text. See Section 5.1, “Dimensions” (p. 9) for the possible values for padding.
<code>pady</code>	Additional padding above and below the text.
<code>relief</code>	Specifies the relief type for the button (see Section 5.6, “Relief styles” (p. 12)). The default relief is <code>tk.RAISED</code> .
<code>repeatdelay</code>	See <code>repeatinterval</code> , below.
<code>repeatinterval</code>	Normally, a button fires only once when the user releases the mouse button. If you want the button to fire at regular intervals as long as the mouse button is held down, set this option to a number of milliseconds to be used between repeats, and set the <code>repeatdelay</code> to the number of milliseconds to wait before starting to repeat. For example, if you specify “ <code>repeatdelay=500, repeatinterval=100</code> ” the button will fire after half a second, and every tenth of a second thereafter, until the user releases the mouse button. If the user does not hold the mouse button down at least <code>repeatdelay</code> milliseconds, the button will fire normally.
<code>state</code>	Set this option to <code>tk.DISABLED</code> to gray out the button and make it unresponsive. Has the value <code>tk.ACTIVE</code> when the mouse is over it. Default is <code>tk.NORMAL</code> .
<code>takefocus</code>	Normally, keyboard focus does visit buttons (see Section 53, “Focus: routing keyboard input” (p. 155)), and a <code>space</code> character acts as the same as a mouse click, “pushing” the button. You can set the <code>takefocus</code> option to zero to prevent focus from visiting the button.

text	Text displayed on the button. Use internal newlines to display multiple text lines.
textvariable	An instance of <code>StringVar()</code> that is associated with the text on this button. If the variable is changed, the new value will be displayed on the button. See Section 52, “Control variables: the values behind the widgets” (p. 153).
underline	Default is -1, meaning that no character of the text on the button will be underlined. If nonnegative, the corresponding text character will be underlined. For example, <code>underline=1</code> would underline the second character of the button’s text.
width	Width of the button in letters (if displaying text) or pixels (if displaying an image).
wraplength	If this value is set to a positive number, the text lines will be wrapped to fit within this length. For possible values, see Section 5.1, “Dimensions” (p. 9).

Methods on `Button` objects:

.flash()

Causes the button to flash several times between active and normal colors. Leaves the button in the state it was in originally. Ignored if the button is disabled.

.invoke()

Calls the button’s `command` callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

8. The Canvas widget

A canvas is a rectangular area intended for drawing pictures or other complex layouts. On it you can place graphics, text, widgets, or frames. See the following sections for methods that create objects on canvases:

- `.create_arc()`: A slice out of an ellipse. See Section 8.7, “Canvas arc objects” (p. 28).
- `.create_bitmap()`: An image as a bitmap. See Section 8.8, “Canvas bitmap objects” (p. 29).
- `.create_image()`: A graphic image. See Section 8.9, “Canvas image objects” (p. 30).
- `.create_line()`: One or more line segments. See Section 8.10, “Canvas line objects” (p. 30).
- `.create_oval()`: An ellipse; use this also for drawing circles, which are a special case of an ellipse. See Section 8.11, “Canvas oval objects” (p. 32).
- `.create_polygon()`: A polygon. See Section 8.12, “Canvas polygon objects” (p. 33).
- `.create_rectangle()`: A rectangle. See Section 8.13, “Canvas rectangle objects” (p. 35).
- `.create_text()`: Text annotation. See Section 8.14, “Canvas text objects” (p. 37).
- `.create_window()`: A rectangular window. See Section 8.15, “Canvas window objects” (p. 38).

To create a `Canvas` object:

```
w = tk.Canvas(parent, option=value, ...)
```

The constructor returns the new `Canvas` widget. Supported options include:

Table 6. Canvas widget options

<code>bd</code> or <code>borderwidth</code>	Width of the border around the outside of the canvas; see Section 5.1, “Dimensions” (p. 9). The default is two pixels.
<code>bg</code> or <code>background</code>	Background color of the canvas. Default is a light gray, about '#E4E4E4'.
<code>closeenough</code>	A <code>float</code> that specifies how close the mouse must be to an item to be considered inside it. Default is 1.0.
<code>confine</code>	If true (the default), the canvas cannot be scrolled outside of the <code>scrollregion</code> (see below).
<code>cursor</code>	Cursor used in the canvas. See Section 5.8, “Cursors” (p. 13).
<code>height</code>	Size of the canvas in the Y dimension. See Section 5.1, “Dimensions” (p. 9).
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight.
<code>highlightthickness</code>	Thickness of the focus highlight. The default value is 1.
<code>relief</code>	The relief style of the canvas. Default is <code>tk.FLAT</code> . See Section 5.6, “Relief styles” (p. 12).
<code>scrollregion</code>	A tuple (<code>w</code> , <code>n</code> , <code>e</code> , <code>s</code>) that defines over how large an area the canvas can be scrolled, where <code>w</code> is the left side, <code>n</code> the top, <code>e</code> the right side, and <code>s</code> the bottom.
<code>selectbackground</code>	The background color to use displaying selected items.
<code>selectborderwidth</code>	The width of the border to use around selected items.
<code>selectforeground</code>	The foreground color to use displaying selected items.
<code>takefocus</code>	Normally, focus (see Section 53, “Focus: routing keyboard input” (p. 155)) will cycle through this widget with the tab key only if there are keyboard bindings set for it (see Section 54, “Events” (p. 157) for an overview of keyboard bindings). If you set this option to 1, focus will always visit this widget. Set it to '' to get the default behavior.
<code>width</code>	Size of the canvas in the X dimension. See Section 5.1, “Dimensions” (p. 9).
<code>xscrollincrement</code>	Normally, canvases can be scrolled horizontally to any position. You can get this behavior by setting <code>xscrollincrement</code> to zero. If you set this option to some positive dimension, the canvas can be positioned only on multiples of that distance, and the value will be used for scrolling by <i>scrolling units</i> , such as when the user clicks on the arrows at the ends of a scrollbar. For more information on scrolling units, see Section 22, “The Scrollbar widget” (p. 74).
<code>xscrollcommand</code>	If the canvas is scrollable, set this option to the <code>.set()</code> method of the horizontal scrollbar.
<code>yscrollincrement</code>	Works like <code>xscrollincrement</code> , but governs vertical movement.
<code>yscrollcommand</code>	If the canvas is scrollable, this option should be the <code>.set()</code> method of the vertical scrollbar.

8.1. Canvas coordinates

Because the canvas may be larger than the window, and equipped with scrollbars to move the overall canvas around in the window, there are two coordinate systems for each canvas:

- The *window coordinates* of a point are relative to the top left corner of the area on the display where the canvas appears.
- The *canvas coordinates* of a point are relative to the top left corner of the total canvas.

8.2. The Canvas display list

The *display list* refers to the sequence of all the objects on the canvas, from background (the “bottom” of the display list) to foreground (the “top”).

If two objects overlap, the one *above* the other in the display list means the one closer to the foreground, which will appear in the area of overlap and obscure the one *below*. By default, new objects are always created at the top of the display list (and hence in front of all other objects), but you can re-order the display list.

8.3. Canvas object IDs

The *object ID* of an object on the canvas is the value returned by the constructor for that object. All object ID values are simple integers, and the object ID of an object is unique within that canvas.

8.4. Canvas tags

A *tag* is a string that you can associate with objects on the canvas.

- A tag can be associated with any number of objects on the canvas, including zero.
- An object can have any number of tags associated with it, including zero.

Tags have many uses. For example, if you are drawing a map on a canvas, and there are text objects for the labels on rivers, you could attach the tag '`'riverLabel'`' to all those text objects. This would allow you to perform operations on all the objects with that tag, such as changing their color or deleting them.

8.5. Canvas `tagOrId` arguments

A `tagOrId` argument specifies one or more objects on the canvas.

- If a `tagOrId` argument is an integer, it is treated as an object ID, and it applies only to the unique object with that ID. See Section 8.3, “Canvas object IDs” (p. 22).
- If such an argument is a string, it is interpreted as a tag, and selects all the objects that have that tag (if there are any). See Section 8.4, “Canvas tags” (p. 22).

8.6. Methods on Canvas widgets

All `Canvas` objects support these methods:

`.addtag_above(newTag, tagOrId)`

Attaches a new tag to the object just above the one specified by `tagOrId` in the display list. The `newTag` argument is the tag you want to attach, as a string.

.addtag_all(*newTag*)

Attaches the given tag *newTag* to all the objects on the canvas.

.addtag_below(*newTag*, *tagOrID*)

Attaches a new tag to the object just below the one specified by *tagOrID* in the display list. The *newTag* argument is a tag string.

.addtag_closest(*newTag*, *x*, *y*, *halo=None*, *start=None*)

Adds a tag to the object closest to screen coordinate (x,y). If there are two or more objects at the same distance, the one higher in the display list is selected.

Use the *halo* argument to increase the effective size of the point. For example, a value of 5 would treat any object within 5 pixels of (x,y) as overlapping.

If an object ID is passed in the *start* argument, this method tags the highest qualifying object that is below *start* in the display list.

.addtag_enclosed(*newTag*, *x1*, *y1*, *x2*, *y2*)

Add tag *newTag* to all objects that occur completely within the rectangle whose top left corner is (*x1*, *y1*) and whose bottom right corner is (*x2*, *y2*).

.addtag_overlapping(*newTag*, *x1*, *y1*, *x2*, *y2*)

Like the previous method, but affects all objects that share at least one point with the given rectangle.

.addtag_withtag(*newTag*, *tagOrID*)

Adds tag *newTag* to the object or objects specified by *tagOrID*.

.bbox(*tagOrID=None*)

Returns a tuple (*x₁*, *y₁*, *x₂*, *y₂*) describing a rectangle that encloses all the objects specified by *tagOrID*. If the argument is omitted, returns a rectangle enclosing all objects on the canvas. The top left corner of the rectangle is (*x₁*, *y₁*) and the bottom right corner is (*x₂*, *y₂*).

.canvasx(*screenx*, *gridspacing=None*)

Translates a window x coordinate *screenx* to a canvas coordinate. If *gridspacing* is supplied, the canvas coordinate is rounded to the nearest multiple of that value.

.canvasy(*screeny*, *gridspacing=None*)

Translates a window y coordinate *screeny* to a canvas coordinate. If *gridspacing* is supplied, the canvas coordinate is rounded to the nearest multiple of that value.

.coords(*tagOrID*, *x0*, *y0*, *x1*, *y1*, ..., *xn*, *yn*)

If you pass only the *tagOrID* argument, returns a tuple of the coordinates of the lowest or only object specified by that argument. The number of coordinates depends on the type of object. In most cases it will be a 4-tuple (*x₁*, *y₁*, *x₂*, *y₂*) describing the bounding box of the object.

You can move an object by passing in new coordinates.

.dchars(*tagOrID*, *first=0*, *last=first*)

Deletes characters from a text item or items. Characters between *first* and *last inclusive* are deleted, where those values can be integer indices or the string 'end' to mean the end of the text. For example, for a canvas *C* and an item *I*, *C.dchars(I, 1, 1)* will remove the second character.

.delete(*tagOrID*)

Deletes the object or objects selected by *tagOrID*. It is not considered an error if no items match *tagOrID*.

.dtag(*tagOrID*, *tagToDelete*)

Removes the tag specified by *tagToDelete* from the object or objects specified by *tagOrID*.

.find_above(tagOrId)

Returns the ID number of the object just above the object specified by *tagOrId*. If multiple objects match, you get the highest one. Returns an empty tuple if you pass it the object ID of the highest object.

.find_all()

Returns a list of the object ID numbers for all objects on the canvas, from lowest to highest.

.find_below(tagOrId)

Returns the object ID of the object just below the one specified by *tagOrId*. If multiple objects match, you get the lowest one. Returns an empty tuple if you pass it the object ID of the lowest object.

.find_closest(x, y, halo=None, start=None)

Returns a singleton tuple containing the object ID of the object closest to point (*x*, *y*). If there are no qualifying objects, returns an empty tuple.

Use the *halo* argument to increase the effective size of the point. For example, *halo*=5 would treat any object within 5 pixels of (*x*, *y*) as overlapping.

If an object ID is passed as the *start* argument, this method returns the highest qualifying object that is below *start* in the display list.

.find_enclosed(x1, y1, x2, y2)

Returns a list of the object IDs of all objects that occur completely within the rectangle whose top left corner is (*x1*, *y1*) and bottom right corner is (*x2*, *y2*).

.find_overlapping(x1, y1, x2, y2)

Like the previous method, but returns a list of the object IDs of all the objects that share at least one point with the given rectangle.

.find_withtag(tagOrId)

Returns a list of the object IDs of the object or objects specified by *tagOrId*.

.focus(tagOrId=None)

Moves the focus to the object specified by *tagOrId*. If there are multiple such objects, moves the focus to the first one in the display list that allows an insertion cursor. If there are no qualifying items, or the canvas does not have focus, focus does not move.

If the argument is omitted, returns the ID of the object that has focus, or '' if none of them do.

.gettags(tagOrId)

If *tagOrId* is an object ID, returns a list of all the tags associated with that object. If the argument is a tag, returns all the tags for the lowest object that has that tag.

.icursor(tagOrId, index)

Assuming that the selected item allows text insertion and has the focus, sets the insertion cursor to *index*, which may be either an integer index or the string 'end'. Has no effect otherwise.

.index(tagOrId, specifier)

Returns the integer index of the given *specifier* in the text item specified by *tagOrId* (the lowest one that, if *tagOrId* specifies multiple objects). The return value is the corresponding position as an integer, with the usual Python convention, where 0 is the position before the first character.

The *specifier* argument may be any of:

- `tk.INSERT`, to return the current position of the insertion cursor.
- `tk.END`, to return the position after the last character of the item.
- `tk.SEL_FIRST`, to return the position of the start of the current text selection. *Tkinter* will raise a `tk.TclError` exception if the text item does not currently contain the text selection.

- `tk.SEL_LAST`, to return the position after the end of the current text selection, or raise `tk.TclError` if the item does not currently contain the selection.
- A string of the form “`@x,y`”, to return the character of the character containing canvas coordinates (x, y) . If those coordinates are above or to the left of the text item, the method returns 0; if the coordinates are to the right of or below the item, the method returns the index of the end of the item.

.insert(*tagOrId, specifier, text*)

Inserts the given *string* into the object or objects specified by *tagOrId*, at the position given by the *specifier* argument.

The *specifier* values may be:

- Any of the keywords `tk.INSERT`, `tk.END`, `tk.SEL_FIRST`, or `tk.SEL_LAST`. Refer to the description of the `index` method above for the interpretation of these codes.
- The position of the desired insertion, using the normal Python convention for positions in strings.

.itemcget(*tagOrId, option*)

Returns the value of the given configuration *option* in the selected object (or the lowest object if *tagOrId* specifies more than one). This is similar to the `.cget()` method for *Tkinter* objects.

.itemconfigure(*tagOrId, option, ...*)

If no *option* arguments are supplied, returns a dictionary whose keys are the options of the object specified by *tagOrId* (the lowest one, if *tagOrId* specifies multiple objects).

To change the configuration option of the specified item, supply one or more keyword arguments of the form *option=value*.

.move(*tagOrId, xAmount, yAmount*)

Moves the items specified by *tagOrId* by adding *xAmount* to their x coordinates and *yAmount* to their y coordinates.

.postscript(*option, ...*)

Generates an Encapsulated PostScript representation of the canvas's current contents. The options include:

<code>colormode</code>	Use 'color' for color output, 'gray' for grayscale, or 'mono' for black and white.
<code>file</code>	If supplied, names a file where the PostScript will be written. If this option is not given, the PostScript is returned as a string.
<code>height</code>	How much of the Y size of the canvas to print. Default is the entire visible height of the canvas.
<code>rotate</code>	If false, the page will be rendered in portrait orientation; if true, in landscape.
<code>x</code>	Leftmost canvas coordinate of the area to print.
<code>y</code>	Topmost canvas coordinate of the area to print.
<code>width</code>	How much of the X size of the canvas to print. Default is the visible width of the canvas.

.scale(*tagOrId, xOffset, yOffset, xScale, yScale*)

Scale all objects according to their distance from a point P=(*xOffset, yOffset*). The scale factors *xScale* and *yScale* are based on a value of 1.0, which means no scaling. Every point in the objects selected by *tagOrId* is moved so that its x distance from P is multiplied by *xScale* and its y distance is multiplied by *yScale*.

This method will not change the size of a text item, but may move it.

.scan_dragto(*x*, *y*, *gain*=10.0)

See the `.scan_mark()` method below.

.scan_mark(*x*, *y*)

This method is used to implement fast scrolling of a canvas. The intent is that the user will press and hold a mouse button, then move the mouse up to scan (scroll) the canvas horizontally and vertically in that direction at a rate that depends on how far the mouse has moved since the mouse button was depressed.

To implement this feature, bind the mouse's button-down event to a handler that calls `scan_mark(x, y)` where *x* and *y* are the current mouse coordinates. Bind the `<Motion>` event to a handler that, assuming the mouse button is still down, calls `scan_dragto(x, y, gain)` where *x* and *y* are the current mouse coordinates.

The *gain* argument controls the rate of scanning. This argument has a default value of 10.0. Use larger numbers for faster scanning.

.select_adjust(*oid*, *specifier*)

Adjusts the boundaries of the current text selection to include the position given by the *specifier* argument, in the text item with the object ID *oid*.

The current selection anchor is also set to the specified position. For a discussion of the selection anchor, see the canvas `select_from` method below.

For the values of *specifier*, see the canvas `insert` method above.

.select_clear()

Removes the current text selection, if it is set. If there is no current selection, does nothing.

.select_from(*oid*, *specifier*)

This method sets the *selection anchor* to the position given by the *specifier* argument, within the text item whose object ID is given by *oid*.

The currently selected text on a given canvas is specified by three positions: the start position, the end position, and the selection anchor, which may be anywhere within those two positions.

To change the position of the currently selected text, use this method in combination with the `select_adjust`, `select_from`, and `select_to` canvas methods (q.v.).

.select_item()

If there is a current text selection on this canvas, return the object ID of the text item containing the selection. If there is no current selection, this method returns `None`.

.select_to(*oid*, *specifier*)

This method changes the current text selection so that it includes the select anchor and the position given by *specifier* within the text item whose object ID is given by *oid*. For the values of *specifier*, see the canvas `insert` method above.

.tag_bind(*tagOrId*, *sequence*=None, *function*=None, *add*=None)

Binds events to objects on the canvas. For the object or objects selected by *tagOrId*, associates the handler *function* with the event *sequence*. If the *add* argument is a string starting with '+', the new binding is added to existing bindings for the given *sequence*, otherwise the new binding replaces that for the given *sequence*.

For general information on event bindings, see Section 54, "Events" (p. 157).

Note that the bindings are applied to items that have this tag at the time of the `tag_bind` method call. If tags are later removed from those items, the bindings will persist on those items. If the tag you specify is later applied to items that did not have that tag when you called `tag_bind`, that binding will *not* be applied to the newly tagged items.

.tag_lower(*tagOrId*, *belowThis*)

Moves the object or objects selected by *tagOrId* within the display list to a position just below the first or only object specied by the tag or ID *belowThis*.

If there are multiple items with tag *tagOrId*, their relative stacking order is preserved.

This method does not affect canvas window items. To change a window item's stacking order, use a *lower* or *lift* method on the window.

.tag_raise(*tagOrId*, *aboveThis*)

Moves the object or objects selected by *tagOrId* within the display list to a position just above the first or only object specied by the tag or ID *aboveThis*.

If there are multiple items with tag *tagOrId*, their relative stacking order is preserved.

This method does not affect canvas window items. To change a window item's stacking order, use a *lower* or *lift* method on the window.

.tag_unbind(*tagOrId*, *sequence*, *funcId=None*)

Removes bindings for handler *funcId* and event *sequence* from the canvas object or objects specified by *tagOrId*. See Section 54, “Events” (p. 157).

.type(*tagOrId*)

Returns the type of the first or only object specied by *tagOrId*. The return value will be one of the strings 'arc', 'bitmap', 'image', 'line', 'oval', 'polygon', 'rectangle', 'text', or 'window'.

.xview(*tk.MOVETO*, *fraction*)

This method scrolls the canvas relative to its image, and is intended for binding to the *command* option of a related scrollbar. The canvas is scrolled horizontally to a position given by *offset*, where 0.0 moves the canvas to its leftmost position and 1.0 to its rightmost position.

.xview(*tk.SCROLL*, *n*, *what*)

This method moves the canvas left or right: the *what* argument specifies how much to move and can be either *tk.UNITS* or *tk.PAGES*, and *n* tells how many units to move the canvas to the right relative to its image (or left, if negative).

The size of the move for *tk.UNITS* is given by the value of the canvas's *xscrollincrement* option; see Section 22, “The Scrollbar widget” (p. 74).

For movements by *tk.PAGES*, *n* is multiplied by nine-tenths of the width of the canvas.

.xview_moveto(*fraction*)

This method scrolls the canvas in the same way as *.xview(*tk.MOVETO*, *fraction*)*.

.xview_scroll(*n*, *what*)

Same as *.xview(*tk.SCROLL*, *n*, *what*)*.

.yview(*tk.MOVETO*, *fraction*)

The vertical scrolling equivalent of *.xview(*tk.MOVETO*, ...)*.

.yview(*tk.SCROLL*, *n*, *what*)

The vertical scrolling equivalent of *.xview(*tk.SCROLL*, ...)*.

.yview_moveto(*fraction*)

The vertical scrolling equivalent of *.xview()*.

.yview_scroll(*n*, *what*)

The vertical scrolling equivalents of *.xview()*, *.xview_moveto()*, and *.xview_scroll()*.

8.7. Canvas arc objects

An *arc object* on a canvas, in its most general form, is a wedge-shaped slice taken out of an ellipse. This includes whole ellipses and circles as special cases. See Section 8.11, “Canvas oval objects” (p. 32) for more on the geometry of the ellipse drawn.

To create an arc object on a canvas C , use:

```
id = C.create_arc(x0, y0, x1, y1, option, ...)
```

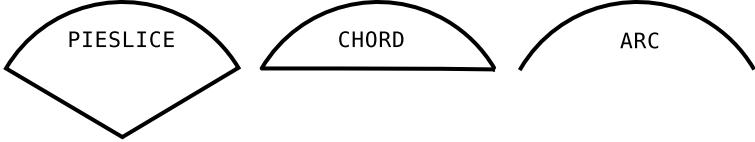
The constructor returns the object ID of the new arc object on canvas C .

Point $(x0, y0)$ is the top left corner and $(x1, y1)$ the lower right corner of a rectangle into which the ellipse is fit. If this rectangle is square, you get a circle.

The various options include:

Table 7. Canvas arc options

activedash	These options apply when the arc is in the <code>tk.ACTIVE</code> state, that is, when the mouse is over the arc. For example, the <code>activefill</code> option specifies the interior color when the arc is active. For option values, see <code>dash</code> , <code>fill</code> , <code>outline</code> , <code>outlinestipple</code> , <code>stipple</code> , and <code>width</code> , respectively.
activefill	
activeoutline	
activeoutlinestipple	
activestipple	
activewidth	
dash	Dash pattern for the outline. See Section 5.13, “Dash patterns” (p. 17).
dashoffset	Dash pattern offset for the outline. See Section 5.13, “Dash patterns” (p. 17).
disableddash	These options apply when the arc's <code>state</code> is <code>tk.DISABLED</code> .
disabledfill	
disabledoutline	
disabledoutlinestipple	
disabledstipple	
disabledwidth	
extent	Width of the slice in degrees. The slice starts at the angle given by the <code>start</code> option and extends counterclockwise for <code>extent</code> degrees.
fill	By default, the interior of an arc is transparent, and <code>fill=''</code> will select this behavior. You can also set this option to any color and the interior of the arc will be filled with that color.
offset	Stipple pattern offset for the interior of the arc. See Section 5.14, “Matching stipple patterns” (p. 17).
outline	The color of the border around the outside of the slice. Default is black.
outlineoffset	Stipple pattern offset for the outline. See Section 5.14, “Matching stipple patterns” (p. 17).
outlinestipple	If the <code>outline</code> option is used, this option specifies a bitmap used to stipple the border. Default is black, and that default can be specified by setting <code>outlinestipple=''</code> .

<code>start</code>	Starting angle for the slice, in degrees, measured from <code>+x</code> direction. If omitted, you get the entire ellipse.
<code>state</code>	This option is <code>tk.NORMAL</code> by default. It may be set to <code>tk.HIDDEN</code> to make the arc invisible or to <code>tk.DISABLED</code> to gray out the arc and make it unresponsive to events.
<code>stipple</code>	A bitmap indicating how the interior fill of the arc will be stippled. Default is <code>stipple=' '</code> (solid). You'll probably want something like <code>stipple='gray25'</code> . Has no effect unless <code>fill</code> has been set to some color.
<code>style</code>	The default is to draw the whole arc; use <code>style=tk.PIESLICE</code> for this style. To draw only the circular arc at the edge of the slice, use <code>style=tk.ARCH</code> . To draw the circular arc and the chord (a straight line connecting the endpoints of the arc), use <code>style=tk.CHORD</code> .
	 <p>The image shows three diagrams of an arc on a canvas. The first diagram, labeled 'PIESLICE', shows a quarter-circle arc with its interior filled. The second diagram, labeled 'CHORD', shows a semi-circle arc with a straight line chord drawn from its start point to its end point. The third diagram, labeled 'ARC', shows a full semi-circle arc without any chord.</p>
<code>tags</code>	If a single string, the arc is tagged with that string. Use a tuple of strings to tag the arc with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
<code>width</code>	Width of the border around the outside of the arc. Default is 1 pixel.

8.8. Canvas bitmap objects

A bitmap object on a canvas is shown as two colors, the background color (for 0 data values) and the foreground color (for 1 values).

To create a bitmap object on a canvas C , use:

```
id = C.create_bitmap(x, y, *options ...)
```

which returns the integer ID number of the image object for that canvas.

The x and y values are the reference point that specifies where the bitmap is placed.

Options include:

Table 8. Canvas bitmap options

<code>activebackground</code>	These options specify the <code>background</code> , <code>bitmap</code> , and <code>foreground</code> values when the bitmap is active, that is, when the mouse is over the bitmap.
<code>activebitmap</code>	
<code>activeforeground</code>	
<code>anchor</code>	The bitmap is positioned relative to point (x, y) . The default is <code>anchor=tk.CENTER</code> , meaning that the bitmap is centered on the (x, y) position. See Section 5.5, “Anchors” (p. 12) for the various <code>anchor</code> option values. For example, if you specify <code>anchor=tk.NE</code> , the bitmap will be positioned so that point (x, y) is located at the northeast (top right) corner of the bitmap.
<code>background</code>	The color that will appear where there are 0 values in the bitmap. The default is <code>background=' '</code> , meaning transparent.

bitmap	The bitmap to be displayed; see Section 5.7, “Bitmaps” (p. 12).
disabledbackground	These options specify the background, bitmap, and foreground to be used when the bitmap's <code>state</code> is <code>tk.DISABLED</code> .
disabledbitmap	
disabledforeground	
foreground	The color that will appear where there are 1 values in the bitmap. The default is <code>foreground='black'</code> .
state	By default, items are created with <code>state=tk.NORMAL</code> . Use <code>tk.DISABLED</code> to make the item grayed out and unresponsive to events; use <code>tk.HIDDEN</code> to make the item invisible.
tags	If a single string, the bitmap is tagged with that string. Use a tuple of strings to tag the bitmap with multiple tags. See Section 8.4, “Canvas tags” (p. 22).

8.9. Canvas image objects

To display a graphics image on a canvas C , use:

```
id = C.create_image(x, y, option, ...)
```

This constructor returns the integer ID number of the image object for that canvas.

The image is positioned relative to point (x, y) . Options include:

Table 9. Canvas image options

activeimage	Image to be displayed when the mouse is over the item. For option values, see <code>image</code> below.
anchor	The default is <code>anchor=tk.CENTER</code> , meaning that the image is centered on the (x, y) position. See Section 5.5, “Anchors” (p. 12) for the possible values of this option. For example, if you specify <code>anchor=tk.S</code> , the image will be positioned so that point (x, y) is located at the center of the bottom (south) edge of the image.
disabledimage	Image to be displayed when the item is inactive. For option values, see <code>image</code> below.
image	The image to be displayed. See Section 5.9, “Images” (p. 14), above, for information about how to create images that can be loaded onto canvases.
state	Normally, image objects are created in state <code>tk.NORMAL</code> . Set this value to <code>tk.DISABLED</code> to make it grayed-out and unresponsive to the mouse. If you set it to <code>tk.HIDDEN</code> , the item is invisible.
tags	If a single string, the image is tagged with that string. Use a tuple of strings to tag the image with multiple tags. See Section 8.4, “Canvas tags” (p. 22).

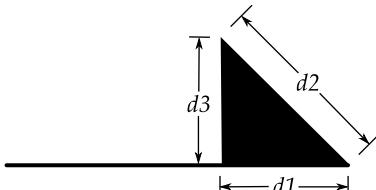
8.10. Canvas line objects

In general, a line can consist of any number of segments connected end to end, and each segment can be straight or curved. To create a canvas line object on a canvas C , use:

```
id = C.create_line(x0, y0, x1, y1, ..., xn, yn, option, ...)
```

The line goes through the series of points $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$. Options include:

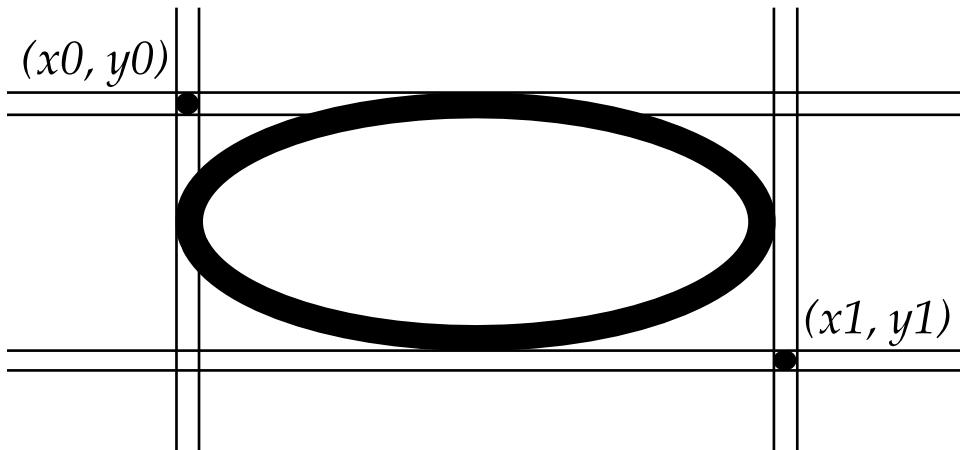
Table 10. Canvas line options

activedash	These options specify the <code>dash</code> , <code>fill</code> , <code>stipple</code> , and <code>width</code> values to be used when the line is active, that is, when the mouse is over it.
activefill	
activestipple	
activewidth	
arrow	The default is for the line to have no arrowheads. Use <code>arrow=tk.FIRST</code> to get an arrowhead at the $(x\theta, y\theta)$ end of the line. Use <code>arrow=tk.LAST</code> to get an arrowhead at the far end. Use <code>arrow=tk.BOTH</code> for arrowheads at both ends.
arrowshape	A tuple $(d1, d2, d3)$ that describes the shape of the arrowheads added by the <code>arrow</code> option. Default is $(8, 10, 3)$.
	 <p>The diagram shows a black triangle representing an arrowhead. A horizontal line segment below it represents the base of the arrow. A vertical double-headed arrow labeled $d3$ indicates the height of the arrowhead from the base to its apex. Two diagonal double-headed arrows labeled $d2$ indicate the slant height of the sides of the triangle. A horizontal double-headed arrow labeled $d1$ indicates the width of the base of the triangle.</p>
capstyle	You can specify the shape of the ends of the line with this option; see Section 5.12, “Cap and join styles” (p. 16). The default option is <code>tk.BUTT</code> .
dash	To produce a dashed line, specify this option; see Section 5.13, “Dash patterns” (p. 17). The default appearance is a solid line.
dashoffset	If you specify a <code>dash</code> pattern, the default is to start the specified pattern at the beginning of the line. The <code>dashoffset</code> option allows you to specify that the start of the dash pattern occurs at a given distance after the start of the line. See Section 5.13, “Dash patterns” (p. 17).
disableddash	The <code>dash</code> , <code>fill</code> , <code>stipple</code> , and <code>width</code> values to be used when the item is in the <code>tk.DISABLED</code> state.
disabledfill	
disabledstipple	
disabledwidth	
fill	The color to use in drawing the line. Default is <code>fill='black'</code> .
joinstyle	For lines that are made up of more than one line segment, this option controls the appearance of the junction between segments. For more details, see Section 5.12, “Cap and join styles” (p. 16). The default style is <code>ROUND</code> .
offset	For stippled lines, the purpose of this option is to match the item's stippling pattern with those of adjacent objects. See Section 5.14, “Matching stipple patterns” (p. 17)..
smooth	If true, the line is drawn as a series of parabolic splines fitting the point set. Default is false, which renders the line as a set of straight segments.
splinesteps	If the <code>smooth</code> option is true, each spline is rendered as a number of straight line segments. The <code>splinesteps</code> option specifies the number of segments used to approximate each section of the line; the default is <code>splinesteps=12</code> .

state	Normally, line items are created in state <code>tk.NORMAL</code> . Set this option to <code>tk.HIDDEN</code> to make the line invisible; set it to <code>tk.DISABLED</code> to make it unresponsive to the mouse.
stipple	To draw a stippled line, set this option to a bitmap that specifies the stippling pattern, such as <code>stipple='gray25'</code> . See Section 5.7, “Bitmaps” (p. 12) for the possible values.
tags	If a single string, the line is tagged with that string. Use a tuple of strings to tag the line with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
width	The line’s width. Default is 1 pixel. See Section 5.1, “Dimensions” (p. 9) for possible values.

8.11. Canvas oval objects

Ovals, mathematically, are ellipses, including circles as a special case. The ellipse is fit into a rectangle defined by the coordinates $(x0, y0)$ of the top left corner and the coordinates $(x1, y1)$ of a point just *outside* of the bottom right corner.



The oval will coincide with the top and left-hand lines of this box, but will fit just inside the bottom and right-hand sides.

To create an ellipse on a canvas C , use:

```
id = C.create_oval(x0, y0, x1, y1, option, ...)
```

which returns the object ID of the new oval object on canvas C .

Options for ovals:

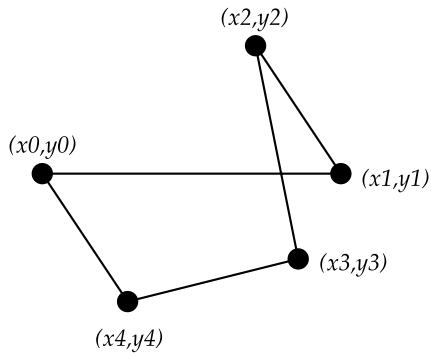
Table 11. Canvas oval options

activedash	These options specify the dash pattern, fill color, outline color, outline stipple pattern, interior stipple pattern, and outline width values to be used when the oval is in the <code>tk.ACTIVE</code> state, that is, when the mouse is over the oval. For option values, see <code>dash</code> , <code>fill</code> , <code>outline</code> , <code>outlinestipple</code> , <code>stipple</code> , and <code>width</code> .
activefill	
activeoutline	
activeoutlinestipple	

<code>activestipple</code>	
<code>activewidth</code>	
<code>dash</code>	To produce a dashed border around the oval, set this option to a dash pattern; see Section 5.13, “Dash patterns” (p. 17)
<code>dashoffset</code>	When using the <code>dash</code> option, the <code>dashoffset</code> option is used to change the alignment of the border's dash pattern relative to the oval. See Section 5.14, “Matching stipple patterns” (p. 17).
<code>disableddash</code>	These options specify the appearance of the oval when the item's <code>state</code> is <code>tk.DISABLED</code> .
<code>disabledfill</code>	
<code>disabledoutline</code>	
<code>disabledoutlinestipple</code>	
<code>disabledstipple</code>	
<code>disabledwidth</code>	
<code>fill</code>	The default appearance of an oval's interior is transparent, and a value of <code>fill=''</code> will select this behavior. You can also set this option to any color and the interior of the ellipse will be filled with that color; see Section 5.3, “Colors” (p. 10).
<code>offset</code>	Stipple pattern offset of the interior. See Section 5.14, “Matching stipple patterns” (p. 17).
<code>outline</code>	The color of the border around the outside of the ellipse. Default is <code>outline='black'</code> .
<code>outlineoffset</code>	Stipple pattern offset of the border. See Section 5.14, “Matching stipple patterns” (p. 17).
<code>stipple</code>	A bitmap indicating how the interior of the ellipse will be stippled. Default is <code>stipple=''</code> , which means a solid color. A typical value would be <code>stipple='gray25'</code> . Has no effect unless the <code>fill</code> has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
<code>outlinestipple</code>	Stipple pattern to be used for the border. For option values, see <code>stipple</code> below.
<code>state</code>	By default, oval items are created in state <code>tk.NORMAL</code> . Set this option to <code>tk.DISABLED</code> to make the oval unresponsive to mouse actions. Set it to <code>tk.HIDDEN</code> to make the item invisible.
<code>tags</code>	If a single string, the oval is tagged with that string. Use a tuple of strings to tag the oval with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
<code>width</code>	Width of the border around the outside of the ellipse. Default is 1 pixel; see Section 5.1, “Dimensions” (p. 9) for possible values. If you set this to zero, the border will not appear. If you set this to zero and make the fill transparent, you can make the entire oval disappear.

8.12. Canvas polygon objects

As displayed, a polygon has two parts: its outline and its interior. Its geometry is specified as a series of vertices $[(x_0, y_0), (x_1, y_1), \dots (x_n, y_n)]$, but the actual perimeter includes one more segment from (x_n, y_n) back to (x_0, y_0) . In this example, there are five vertices:



To create a new polygon object on a canvas C :

```
id = C.create_polygon(x0, y0, x1, y1, ..., option, ...)
```

The constructor returns the object ID for that object. Options:

Table 12. Canvas polygon options

<code>activedash</code>	These options specify the appearance of the polygon when it is in the <code>tk.ACTIVE</code> state, that is, when the mouse is over it. For option values, see <code>dash</code> , <code>fill</code> , <code>outline</code> , <code>outlinestipple</code> , <code>stipple</code> , and <code>width</code> .
<code>activefill</code>	
<code>activeoutline</code>	
<code>activeoutlinestipple</code>	
<code>activestipple</code>	
<code>activewidth</code>	
<code>dash</code>	Use this option to produce a dashed border around the polygon. See Section 5.13, “Dash patterns” (p. 17).
<code>dashoffset</code>	Use this option to start the dash pattern at some point in its cycle other than the beginning. See Section 5.13, “Dash patterns” (p. 17).
<code>disableddash</code>	These options specify the appearance of the polygon when its <code>state</code> is <code>tk.DISABLED</code> .
<code>disabledfill</code>	
<code>disabledoutline</code>	
<code>disabledoutlinestipple</code>	
<code>disabledstipple</code>	
<code>disabledwidth</code>	
<code>fill</code>	You can color the interior by setting this option to a color. The default appearance for the interior of a polygon is transparent, and you can set <code>fill=''</code> to get this behavior. See Section 5.3, “Colors” (p. 10).
<code>joinstyle</code>	This option controls the appearance of the intersections between adjacent sides of the polygon. See Section 5.12, “Cap and join styles” (p. 16).
<code>offset</code>	Offset of the stipple pattern in the interior of the polygon. See Section 5.14, “Matching stipple patterns” (p. 17).
<code>outline</code>	Color of the outline; defaults to <code>outline=''</code> , which makes the outline transparent.

<code>outlineoffset</code>	Stipple offset for the border. See Section 5.14, “Matching stipple patterns” (p. 17).
<code>outlinestipple</code>	Use this option to get a stippled border around the polygon. The option value must be a bitmap; see Section 5.7, “Bitmaps” (p. 12).
<code>smooth</code>	The default outline uses straight lines to connect the vertices; use <code>smooth=0</code> to get that behavior. If you use <code>smooth=1</code> , you get a continuous spline curve. Moreover, if you set <code>smooth=1</code> , you can make any segment straight by duplicating the coordinates at each end of that segment.
<code>splinesteps</code>	If the <code>smooth</code> option is true, each spline is rendered as a number of straight line segments. The <code>splinesteps</code> option specifies the number of segments used to approximate each section of the line; the default is <code>splinesteps=12</code> .
<code>state</code>	By default, polygons are created in the <code>tk.NORMAL</code> state. Set this option to <code>tk.HIDDEN</code> to make the polygon invisible, or set it to <code>tk.DISABLED</code> to make it unresponsive to the mouse.
<code>stipple</code>	A bitmap indicating how the interior of the polygon will be stippled. Default is <code>stipple=''</code> , which means a solid color. A typical value would be <code>stipple='gray25'</code> . Has no effect unless the <code>fill</code> has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
<code>tags</code>	If a single string, the polygon is tagged with that string. Use a tuple of strings to tag the polygon with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
<code>width</code>	Width of the outline; defaults to 1. See Section 5.1, “Dimensions” (p. 9).

8.13. Canvas rectangle objects

Each rectangle is specified as two points: (x_0, y_0) is the top left corner, and (x_1, y_1) is the location of the pixel just *outside* of the bottom right corner.

For example, the rectangle specified by top left corner (100,100) and bottom right corner (102,102) is a square two pixels by two pixels, including pixel (101,101) but *not* including (102,102).

Rectangles are drawn in two parts:

- The outline lies inside the rectangle on its top and left sides, but *outside* the rectangle on its bottom and right side. The default appearance is a one-pixel-wide black border.

For example, consider a rectangle with top left corner (10,10) and bottom right corner (11,11). If you request no border (`width=0`) and green fill (`fill='green'`), you will get one green pixel at (10,10). However, if you request the same options with a black border (`width=1`), you will get four black pixels at (10,10), (10,11), (11,10), and (11,11).

- The fill is the area inside the outline. Its default appearance is transparent.

To create a rectangle object on canvas `C`:

```
id = C.create_rectangle(x0, y0, x1, y1, option, ...)
```

This constructor returns the object ID of the rectangle on that canvas. Options include:

Table 13. Canvas rectangle options

activedash	These options specify the appearance of the rectangle when its <code>state</code> is <code>tk.ACTIVE</code> , that is, when the mouse is on top of the rectangle. For option values, refer to <code>dash</code> , <code>fill</code> , <code>outline</code> , <code>outlinestipple</code> , <code>stipple</code> , and <code>width</code> below.
activefill	
activeoutline	
activeoutlinestipple	
activestipple	
activewidth	
dash	To produce a dashed border around the rectangle, use this option to specify a dash pattern. See Section 5.13, “Dash patterns” (p. 17).
dashoffset	Use this option to start the border's dash pattern at a different point in the cycle; see Section 5.13, “Dash patterns” (p. 17).
disableddash	These options specify the appearance of the rectangle when its <code>state</code> is <code>tk.DISABLED</code> .
disabledfill	
disabledoutline	
disabledoutlinestipple	
disabledstipple	
disabledwidth	
fill	By default, the interior of a rectangle is empty, and you can get this behavior with <code>fill=''</code> . You can also set the option to a color; see Section 5.3, “Colors” (p. 10).
offset	Use this option to change the offset of the interior stipple pattern. See Section 5.14, “Matching stipple patterns” (p. 17).
outline	The color of the border. Default is <code>outline='black'</code> .
outlineoffset	Use this option to adjust the offset of the stipple pattern in the outline; see Section 5.14, “Matching stipple patterns” (p. 17).
outlinestipple	Use this option to produce a stippled outline. The pattern is specified by a bitmap; see Section 5.7, “Bitmaps” (p. 12).
state	By default, rectangles are created in the <code>tk.NORMAL</code> state. The state is <code>tk.ACTIVE</code> when the mouse is over the rectangle. Set this option to <code>tk.DISABLED</code> to gray out the rectangle and make it unresponsive to mouse events.
stipple	A bitmap indicating how the interior of the rectangle will be stippled. Default is <code>stipple=''</code> , which means a solid color. A typical value would be <code>stipple='gray25'</code> . Has no effect unless the <code>fill</code> has been set to some color. See Section 5.7, “Bitmaps” (p. 12).
tags	If a single string, the rectangle is tagged with that string. Use a tuple of strings to tag the rectangle with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
width	Width of the border. Default is 1 pixel. Use <code>width=0</code> to make the border invisible. See Section 5.1, “Dimensions” (p. 9).

8.14. Canvas text objects

You can display one or more lines of text on a canvas C by creating a text object:

```
id = C.create_text(x, y, option, ...)
```

This returns the object ID of the text object on canvas C . Options include:

Table 14. Canvas text options

activefill	The text color to be used when the text is active, that is, when the mouse is over it. For option values, see <code>fill</code> below.
activestipple	The stipple pattern to be used when the text is active. For option values, see <code>stipple</code> below.
anchor	The default is <code>anchor=tk.CENTER</code> , meaning that the text is centered vertically and horizontally around position (x, y) . See Section 5.5, “Anchors” (p. 12) for possible values. For example, if you specify <code>anchor=tk.SW</code> , the text will be positioned so its lower left corner is at point (x, y) .
disabledfill	The text color to be used when the text object's <code>state</code> is <code>tk.DISABLED</code> . For option values, see <code>fill</code> below.
disabledstipple	The stipple pattern to be used when the text is disabled. For option values, see <code>stipple</code> below.
fill	The default text color is black, but you can render it in any color by setting the <code>fill</code> option to that color. See Section 5.3, “Colors” (p. 10).
font	If you don't like the default font, set this option to any font value. See Section 5.4, “Type fonts” (p. 10).
justify	For multi-line textual displays, this option controls how the lines are justified: <code>tk.LEFT</code> (the default), <code>tk.CENTER</code> , or <code>tk.RIGHT</code> .
offset	The stipple offset to be used in rendering the text. For more information, see Section 5.14, “Matching stipple patterns” (p. 17).
state	By default, the text item's state is <code>tk.NORMAL</code> . Set this option to <code>tk.DISABLED</code> to make it unresponsive to mouse events, or set it to <code>tk.HIDDEN</code> to make it invisible.
stipple	A bitmap indicating how the text will be stippled. Default is <code>stipple=''</code> , which means solid. A typical value would be <code>stipple='gray25'</code> . See Section 5.7, “Bitmaps” (p. 12).
tags	If a single string, the text object is tagged with that string. Use a tuple of strings to tag the object with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
text	The text to be displayed in the object, as a string. Use newline characters ('\\n') to force line breaks.
width	If you don't specify a <code>width</code> option, the text will be set inside a rectangle as long as the longest line. However, you can also set the <code>width</code> option to a dimension, and each line of the text will be broken into shorter lines, if necessary, or even broken within words, to fit within the specified width. See Section 5.1, “Dimensions” (p. 9).

You can change the text displayed in a text item.

- To retrieve the text from an item with object ID I on a canvas C , call `C.itemcget(I, 'text')`.

- To replace the text in an item with object ID I on a canvas C with the text from a string S , call $C.itemconfigure(I, text=S)$.

A number of canvas methods allow you to manipulate text items. See Section 8.6, “Methods on Canvas widgets” (p. 22), especially `dchars`, `focus`, `icursor`, `index`, and `insert`.

8.15. Canvas window objects

You can place any *Tkinter* widget onto a canvas by using a *canvas window* object. A window is a rectangular area that can hold one *Tkinter* widget. The widget must be the child of the same top-level window as the canvas, or the child of some widget located in the same top-level window.

If you want to put complex multi-widget objects on a canvas, you can use this method to place a `Frame` widget on the canvas, and then place other widgets inside that frame.

To create a new canvas window object on a canvas C :

```
id = C.create_window(x, y, option, ...)
```

This returns the object ID for the window object. Options include:

Table 15. Canvas window options

<code>anchor</code>	The default is <code>anchor=tk.CENTER</code> , meaning that the window is centered on the (x, y) position. See Section 5.5, “Anchors” (p. 12) for the possible values. For example, if you specify <code>anchor=tk.E</code> , the window will be positioned so that point (x, y) is on the midpoint of its right-hand (east) edge.
<code>height</code>	The height of the area reserved for the window. If omitted, the window will be sized to fit the height of the contained widget. See Section 5.1, “Dimensions” (p. 9) for possible values.
<code>state</code>	By default, window items are in the <code>tk.NORMAL</code> state. Set this option to <code>tk.DISABLED</code> to make the window unresponsive to mouse input, or to <code>tk.HIDDEN</code> to make it invisible.
<code>tags</code>	If a single string, the window is tagged with that string. Use a tuple of strings to tag the window with multiple tags. See Section 8.4, “Canvas tags” (p. 22).
<code>width</code>	The width of the area reserved for the window. If omitted, the window will be sized to fit the width of the contained widget.
<code>window</code>	Use <code>window=w</code> where w is the widget you want to place onto the canvas. If this is omitted initially, you can later call $C.itemconfigure(id, window=w)$ to place the widget w onto the canvas, where id is the window's object ID..

9. The Checkbutton widget



The purpose of a checkbutton widget (sometimes called “checkbox”) is to allow the user to read and select a two-way choice. The graphic above shows how checkbuttons look in the off (0) and on (1) state in one implementation: this is a screen shot of two checkbuttons using 24-point Times font.

The *indicator* is the part of the checkbutton that shows its state, and the *label* is the text that appears beside it.

- You will need to create a control variable, an instance of the `IntVar` class, so your program can query and set the state of the checkbutton. See Section 52, “Control variables: the values behind the widgets” (p. 153), below.
- You can also use event bindings to react to user actions on the checkbutton; see Section 54, “Events” (p. 157), below.
- You can disable a checkbutton. This changes its appearance to “grayed out” and makes it unresponsive to the mouse.
- You can get rid of the checkbutton indicator and make the whole widget a “push-push” button that looks recessed when it is set, and looks raised when it is cleared.

To create a checkbutton in an existing parent window or frame `parent`:

```
w = tk.Checkbutton(parent, option, ...)
```

The constructor returns a new `Checkbutton` widget. Options include:

Table 16. Checkbutton widget options

<code>activebackground</code>	Background color when the checkbutton is under the cursor. See Section 5.3, “Colors” (p. 10).
<code>activeforeground</code>	Foreground color when the checkbutton is under the cursor.
<code>anchor</code>	If the widget inhabits a space larger than it needs, this option specifies where the checkbutton will sit in that space. The default is <code>anchor=tk.CENTER</code> . See Section 5.5, “Anchors” (p. 12) for the allowable values. For example, if you use <code>anchor=NW</code> , the widget will be placed in the upper left corner of the space.
<code>bg</code> or <code>background</code>	The normal background color displayed behind the label and indicator. See Section 5.3, “Colors” (p. 10). For the <code>bitmap</code> option, this specifies the color displayed for 0-bits in the bitmap.
<code>bitmap</code>	To display a monochrome image on a button, set this option to a bitmap; see Section 5.7, “Bitmaps” (p. 12).
<code>bd</code> or <code>borderwidth</code>	The size of the border around the indicator. Default is two pixels. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>command</code>	A procedure to be called every time the user changes the state of this checkbutton.
<code>compound</code>	Use this option to display both text and a graphic, which may be either a bitmap or an image, on the button. Allowable values describe the position of the graphic relative to the text, and may be any of <code>tk.BOTTOM</code> , <code>tk.TOP</code> , <code>tk.LEFT</code> , <code>tk.RIGHT</code> , or <code>tk.CENTER</code> . For example, <code>compound=tk.LEFT</code> would position the graphic to the left of the text.
<code>cursor</code>	If you set this option to a cursor name (see Section 5.8, “Cursors” (p. 13)), the mouse cursor will change to that pattern when it is over the checkbutton.
<code>disabledforeground</code>	The foreground color used to render the text of a disabled checkbutton. The default is a stippled version of the default foreground color.
<code>font</code>	The font used for the <code>text</code> . See Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	The color used to render the <code>text</code> . For the <code>bitmap</code> option, this specifies the color displayed for 1-bits in the bitmap.
<code>height</code>	The number of lines of text on the checkbutton. Default is 1.

<code>highlightbackground</code>	The color of the focus highlight when the checkbutton does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	The color of the focus highlight when the checkbutton has the focus.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
<code>image</code>	To display a graphic image on the button, set this option to an image object. See Section 5.9, “Images” (p. 14).
<code>indicatoron</code>	Normally a checkbutton displays as its indicator a box that shows whether the checkbutton is set or not. You can get this behavior by setting <code>indicatoron=1</code> . However, if you set <code>indicatoron=0</code> , the indicator disappears, and the entire widget becomes a push-push button that looks raised when it is cleared and sunken when it is set. You may want to increase the <code>borderwidth</code> value to make it easier to see the state of such a control.
<code>justify</code>	If the <code>text</code> contains multiple lines, this option controls how the text is justified: <code>tk.CENTER</code> , <code>tk.LEFT</code> , or <code>tk.RIGHT</code> .
<code>offrelief</code>	By default, checkbuttons use the <code>tk.RAISED</code> relief style when the button is off (cleared); use this option to specify a different relief style to be displayed when the button is off. See Section 5.6, “Relief styles” (p. 12) for values.
<code>offvalue</code>	Normally, a checkbutton’s associated control variable will be set to 0 when it is cleared (off). You can supply an alternate value for the off state by setting <code>offvalue</code> to that value.
<code>onvalue</code>	Normally, a checkbutton’s associated control variable will be set to 1 when it is set (on). You can supply an alternate value for the on state by setting <code>onvalue</code> to that value.
<code>overrelief</code>	Use this option to specify a relief style to be displayed when the mouse is over the checkbutton; see Section 5.6, “Relief styles” (p. 12).
<code>padx</code>	How much space to leave to the left and right of the checkbutton and text. Default is 1 pixel. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>pady</code>	How much space to leave above and below the checkbutton and text. Default is 1 pixel.
<code>relief</code>	With the default value, <code>relief=tk.FLAT</code> , the checkbutton does not stand out from its background. You may set this option to any of the other styles (see Section 5.6, “Relief styles” (p. 12)), or use <code>relief=tk.SOLID</code> , which gives you a solid black frame around it.
<code>selectcolor</code>	The color of the checkbutton when it is set. Default is <code>selectcolor='red'</code> .
<code>selectimage</code>	If you set this option to an image, that image will appear in the checkbutton when it is set. See Section 5.9, “Images” (p. 14).
<code>state</code>	The default is <code>state=tk.NORMAL</code> , but you can use <code>state=tk.DISABLED</code> to gray out the control and make it unresponsive. If the cursor is currently over the checkbutton, the state is <code>tk.ACTIVE</code> .
<code>takefocus</code>	The default is that the input focus (see Section 53, “Focus: routing keyboard input” (p. 155)) will pass through a checkbutton. If you set <code>takefocus=0</code> , focus will not pass through it.
<code>text</code>	The label displayed next to the checkbutton. Use newlines ('\\n') to display multiple lines of text.

textvariable	If you need to change the label on a checkbutton during execution, create a StringVar (see Section 52, “Control variables: the values behind the widgets” (p. 153)) to manage the current value, and set this option to that control variable. Whenever the control variable’s value changes, the checkbutton’s annotation will automatically change as well.
underline	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
variable	The control variable that tracks the current state of the checkbutton; see Section 52, “Control variables: the values behind the widgets” (p. 153). Normally this variable is an IntVar, and 0 means cleared and 1 means set, but see the offvalue and onvalue options above.
width	The default width of a checkbutton is determined by the size of the displayed image or text. You can set this option to a number of characters and the checkbutton will always have room for that many characters.
wraplength	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods on checkbuttons include:

.deselect()

Clears (turns off) the checkbutton.

.flash()

Flashes the checkbutton a few times between its active and normal colors, but leaves it the way it started.

.invoke()

You can call this method to get the same actions that would occur if the user clicked on the checkbutton to change its state.

.select()

Sets (turns on) the checkbutton.

.toggle()

Clears the checkbutton if set, sets it if cleared.

10. The Entry widget

The purpose of an **Entry** widget is to let the user see and modify a *single* line of text.

- If you want to display *multiple* lines of text that can be edited, see Section 24, “The **Text** widget” (p. 82).
- If you want to display one or more lines of text that *cannot* be modified by the user, see Section 12, “The **Label** widget” (p. 48).

Some definitions:

- The *selection* is a highlighted region of the text in an **Entry** widget, if there is one.

Typically the selection is made by the user with the mouse, and selected text is copied to the system’s clipboard. However, *Tkinter* allows you to control whether or not selected text gets copied to the clipboard. You can also select text in an **Entry** under program control.

- The *insertion cursor* shows where new text will be inserted. It is displayed only when the user clicks the mouse somewhere in the widget. It usually appears as a blinking vertical line inside the widget. You can customize its appearance in several ways.
- Positions within the widget's displayed text are given as an *index*. There are several ways to specify an index:
 - As normal Python indexes, starting from 0.
 - The constant `tk.END` refers to the position after the existing text.
 - The constant `tk.INSERT` refers to the current position of the insertion cursor.
 - The constant `tk.ANCHOR` refers to the first character of the selection, if there is a selection.
 - You may need to figure out which character position in the widget corresponds to a given mouse position. To simplify that process, you can use as an index a string of the form '`@n`', where *n* is the horizontal distance in pixels between the left edge of the `Entry` widget and the mouse. Such an index will specify the character at that horizontal mouse position.

To create a new `Entry` widget in a root window or frame named *parent*:

```
w = tk.Entry(parent, option, ...)
```

This constructor returns the new `Entry` widget. Options include:

Table 17. `Entry` widget options

<code>bg</code> or <code>background</code>	The background color inside the entry area. Default is a light gray.
<code>bd</code> or <code>borderwidth</code>	The width of the border around the entry area; see Section 5.1, "Dimensions" (p. 9). The default is two pixels.
<code>cursor</code>	The cursor used when the mouse is within the entry widget; see Section 5.8, "Cursors" (p. 13).
<code>disabledbackground</code>	The background color to be displayed when the widget is in the <code>tk.DISABLED</code> state. For option values, see <code>bg</code> above.
<code>disabledforeground</code>	The foreground color to be displayed when the widget is in the <code>tk.DISABLED</code> state. For option values, see <code>fg</code> below.
<code>exportselection</code>	By default, if you select text within an <code>Entry</code> widget, it is automatically exported to the clipboard. To avoid this exportation, use <code>exportselection=0</code> .
<code>fg</code> or <code>foreground</code>	The color used to render the text. Default is black.
<code>font</code>	The font used for text entered in the widget by the user. See Section 5.4, "Type fonts" (p. 10).
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight when the widget has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>insertbackground</code>	By default, the insertion cursor (which shows the point within the text where new keyboard input will be inserted) is black. To get a different color of insertion cursor, set <code>insertbackground</code> to any color; see Section 5.3, "Colors" (p. 10).

<code>insertborderwidth</code>	By default, the insertion cursor is a simple rectangle. You can get the cursor with the <code>tk.RAISED</code> relief effect (see Section 5.6, “Relief styles” (p. 12)) by setting <code>insertborderwidth</code> to the dimension of the 3-d border. If you do, make sure that the <code>insertwidth</code> option is at least twice that value.
<code>insertofftime</code>	By default, the insertion cursor blinks. You can set <code>insertofftime</code> to a value in milliseconds to specify how much time the insertion cursor spends off. Default is 300. If you use <code>insertofftime=0</code> , the insertion cursor won't blink at all.
<code>insertontime</code>	Similar to <code>insertofftime</code> , this option specifies how much time the cursor spends on per blink. Default is 600 (milliseconds).
<code>insertwidth</code>	By default, the insertion cursor is 2 pixels wide. You can adjust this by setting <code>insertwidth</code> to any dimension.
<code>justify</code>	This option controls how the text is justified when the text doesn't fill the widget's width. The value can be <code>tk.LEFT</code> (the default), <code>tk.CENTER</code> , or <code>tk.RIGHT</code> .
<code>readonlybackground</code>	The background color to be displayed when the widget's <code>state</code> option is ' <code>readonly</code> '.
<code>relief</code>	Selects three-dimensional shading effects around the text entry. See Section 5.6, “Relief styles” (p. 12). The default is <code>relief=tk.SUNKEN</code> .
<code>selectbackground</code>	The background color to use displaying selected text. See Section 5.3, “Colors” (p. 10).
<code>selectborderwidth</code>	The width of the border to use around selected text. The default is one pixel.
<code>selectforeground</code>	The foreground (text) color of selected text.
<code>show</code>	Normally, the characters that the user types appear in the entry. To make a “password” entry that echoes each character as an asterisk, set <code>show='*'.</code>
<code>state</code>	Use this option to disable the <code>Entry</code> widget so that the user can't type anything into it. Use <code>state=tk.DISABLED</code> to disable the widget, <code>state=tk.NORMAL</code> to allow user input again. Your program can also find out whether the cursor is currently over the widget by interrogating this option; it will have the value <code>tk.ACTIVE</code> when the mouse is over it. You can also set this option to ' <code>disabled</code> ', which is like the <code>tk.DISABLED</code> state, but the contents of the widget can still be selected or copied.
<code>takefocus</code>	By default, the focus will tab through entry widgets. Set this option to 0 to take the widget out of the sequence. For a discussion of focus, see Section 53, “Focus: routing keyboard input” (p. 155).
<code>textvariable</code>	In order to be able to retrieve the current text from your entry widget, you must set this option to an instance of the <code>StringVar</code> class; see Section 52, “Control variables: the values behind the widgets” (p. 153). You can retrieve the text using <code>v.get()</code> , or set it using <code>v.set()</code> , where <code>v</code> is the associated control variable.
<code>validate</code>	You can use this option to set up the widget so that its contents are checked by a validation function at certain times. See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
<code>validatecommand</code>	A callback that validates the text of the widget. See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).

<code>width</code>	The size of the entry in characters. The default is 20. For proportional fonts, the physical length of the widget will be based on the average width of a character times the value of the <code>width</code> option.
<code>xscrollcommand</code>	If you expect that users will often enter more text than the onscreen size of the widget, you can link your entry widget to a scrollbar. Set this option to the <code>.set</code> method of the scrollbar. For more information, see Section 10.1, “Scrolling an Entry widget” (p. 45).

Methods on `Entry` objects include:

`.delete(first, last=None)`

Deletes characters from the widget, starting with the one at index `first`, up to but *not* including the character at position `last`. If the second argument is omitted, only the single character at position `first` is deleted.

`.get()`

Returns the entry's current text as a string.

`.icursor(index)`

Set the insertion cursor just before the character at the given `index`.

`.index(index)`

Shift the contents of the entry so that the character at the given `index` is the leftmost visible character.
Has no effect if the text fits entirely within the entry.

`.insert(index, s)`

Inserts string `s` before the character at the given `index`.

`.scan_dragto(x)`

See the `scan_mark` method below.

`.scan_mark(x)`

Use this option to set up fast scanning of the contents of the `Entry` widget that has a scrollbar that supports horizontal scrolling.

To implement this feature, bind the mouse's button-down event to a handler that calls `scan_mark(x)`, where `x` is the current mouse `x` position. Then bind the `<Motion>` event to a handler that calls `scan_dragto(x)`, where `x` is the current mouse `x` position. The `scan_dragto` method scrolls the contents of the `Entry` widget continuously at a rate proportional to the horizontal distance between the position at the time of the `scan_mark` call and the current position.

`.select_adjust(index)`

This method is used to make sure that the selection includes the character at the specified `index`. If the selection already includes that character, nothing happens. If not, the selection is expanded from its current position (if any) to include position `index`.

`.select_clear()`

Clears the selection. If there isn't currently a selection, has no effect.

`.select_from(index)`

Sets the `tk.ANCHOR` index position to the character selected by `index`, and selects that character.

`.select_present()`

If there is a selection, returns true, else returns false.

`.select_range(start, end)`

Sets the selection under program control. Selects the text starting at the `start` index, up to but *not* including the character at the `end` index. The `start` position must be before the `end` position.

To select all the text in an entry widget `e`, use `e.select_range(0, tk.END)`.

.select_to(*index*)

Selects all the text from the `tk.ANCHOR` position up to but not including the character at the given *index*.

.xview(*index*)

Same as `.xview()`. This method is useful in linking the `Entry` widget to a horizontal scrollbar. See Section 10.1, “Scrolling an Entry widget” (p. 45).

.xview_moveto(*f*)

Positions the text in the entry so that the character at position *f*, relative to the entire text, is positioned at the left edge of the window. The *f* argument must be in the range [0,1], where 0 means the left end of the text and 1 the right end.

.xview_scroll(*number*, *what*)

Used to scroll the entry horizontally. The *what* argument must be either `tk.UNITS`, to scroll by character widths, or `tk.PAGES`, to scroll by chunks the size of the entry widget. The *number* is positive to scroll left to right, negative to scroll right to left. For example, for an entry widget `e`, `e.xview_scroll(-1, tk.PAGES)` would move the text one “page” to the right, and `e.xview_scroll(4, tk.UNITS)` would move the text four characters to the left.

10.1. Scrolling an Entry widget

Making an `Entry` widget scrollable requires a little extra code on your part to adapt the `Scrollbar` widget’s callback to the methods available on the `Entry` widget. Here are some code fragments illustrating the setup. First, the creation and linking of the `Entry` and `Scrollbar` widgets:

```
self.entry = tk.Entry(self, width=10)
self.entry.grid(row=0, sticky=tk.E+tk.W)

self.entryScroll = tk.Scrollbar(self, orient=tk.HORIZONTAL,
                               command=self.__scrollHandler)
self.entryScroll.grid(row=1, sticky=tk.E+tk.W)
self.entry['xscrollcommand'] = self.entryScroll.set
```

Here’s the adapter function referred to above:

```
def __scrollHandler(self, *L):
    op, howMany = L[0], L[1]

    if op == 'scroll':
        units = L[2]
        self.entry.xview_scroll(howMany, units)
    elif op == 'moveto':
        self.entry.xview_moveto(howMany)
```

10.2. Adding validation to an Entry widget

In some applications, you will want to check the contents of an `Entry` widget to make sure they are valid according to some rule that your application must enforce. You define what is valid by writing a callback function that checks the contents and signals whether it is valid or not.

Here is the procedure for setting up validation on a widget.

1. Write a callback function that checks the text in the `Entry` and returns `True` if the text is valid, or `False` if not. If the callback returns `False`, the user's attempt to edit the text will be refused, and the text will be unchanged.
2. Register the callback function. In this step, you will produce a `Tcl` wrapper around a Python function. Suppose your callback function is a function named `isOkay`. To register this function, use the universal widget method `.register(isOkay)`. This method returns a character string that `Tkinter` can use to call your function.
3. When you call the `Entry` constructor, use the `validatecommand` option in the `Entry` constructor to specify your callback, and use the `validate` option to specify when the callback will be called to validate the text in the callback. The values of these options are discussed in more detail below.

Here are the values of the `validate` option and what they mean.

'focus'

Validate whenever the `Entry` widget gets or loses focus (see Section 53, “Focus: routing keyboard input” (p. 155)).

'focusin'

Validate whenever the widget gets focus.

'focusout'

Validate whenever the widget loses focus.

'key'

Validate whenever any keystroke changes the widget's contents.

'all'

Validate in all the above situations.

'none'

Turn off validation. This is the default option value. Note that this is the string '`'none'`', not the special Python value `None`.

The value of the `validatecommand` option depends on what arguments you would like your callback to receive.

- Perhaps the only thing the callback needs to know is what text currently appears in the `Entry`. If that is the case, it can use the `.get()` method of the `textvariable` associated with the widget to retrieve that text.

In this case, all you need is the option “`validatecommand=f`”, where `f` is the name of your callback function.

- `Tkinter` can also provide a number of items of information to the callback. If you would like to use some of these items, when you call the `Entry` constructor, use the option `validatecommand=(f, s1, s2, ...)`, where `f` is the name of your callback function, and each additional `si` is a substitution code. For each substitution code that you provide, the callback will receive a positional argument containing the appropriate value.

Here are the substitution codes.

Table 18. Callback substitution codes

<code>'%d'</code>	Action code: 0 for an attempted deletion, 1 for an attempted insertion, or -1 if the callback was called for focus in, focus out, or a change to the <code>textvariable</code> .
-------------------	--

'%i'	When the user attempts to insert or delete text, this argument will be the index of the beginning of the insertion or deletion. If the callback was due to focus in, focus out, or a change to the <code>textvariable</code> , the argument will be -1.
'%P'	The value that the text will have if the change is allowed.
'%S'	The text in the entry before the change.
'%S'	If the call was due to an insertion or deletion, this argument will be the text being inserted or deleted.
'%V'	The current value of the widget's <code>validate</code> option.
'%V'	The reason for this callback: one of ' <code>focusin</code> ', ' <code>focusout</code> ', ' <code>key</code> ', or ' <code>forced</code> ' if the <code>textvariable</code> was changed.
'%W'	The name of the widget.

Here is a small example. Suppose you want your callback to receive the '%d' to find out why it was called; '%i' to find out where the insertion or deletion would occur; and '%S' to find out what is to be inserted or deleted. Your method might look like this:

```
def is0kay(self, why, where, what):
    ...
```

Next you use the universal `.register()` method to wrap this function. We assume that `self` is some widget.

```
okayCommand = self.register(is0kay)
```

To set up this callback, you would use these two options in the `Entry` constructor:

```
self.w = Entry(self, validate='all',
               validatecommand=(okayCommand, '%d', '%i', '%S'), ...)
```

Suppose that the `Entry` currently contains the string '`abcdefg`', and the user selects '`cde`' and then presses *Backspace*. This would result in a call `is0kay(0, 2, 'cde')`: 0 for deletion, 2 for the position before '`c`', and '`cde`' for the string to be deleted. If `is0kay()` returns `True`, the new text will be '`abfg`'; if it returns `False`, the text will not change.

The `Entry` widget also supports an `invalidcommand` option that specifies a callback function that is called whenever the `validatecommand` returns `False`. This command may modify the text in the widget by using the `.set()` method on the widget's associated `textvariable`. Setting up this option works the same as setting up the `validatecommand`. You must use the `.register()` method to wrap your Python function; this method returns the name of the wrapped function as a string. Then you will pass as the value of the `invalidcommand` option either that string, or as the first element of a tuple containing substitution codes.

11. The Frame widget

A frame is basically just a container for other widgets.

- Your application's root window is basically a frame.
- Each frame has its own grid layout, so the gridding of widgets within each frame works independently.
- Frame widgets are a valuable tool in making your application modular. You can group a set of related widgets into a compound widget by putting them into a frame. Better yet, you can declare a new

class that inherits from `Frame`, adding your own interface to it. This is a good way to hide the details of interactions within a group of related widgets from the outside world.

To create a new frame widget in a root window or frame named `parent`:

```
w = Frame(parent, option, ...)
```

The constructor returns the new `Frame` widget. Options:

Table 19. `Frame` widget options

<code>bg</code> or <code>background</code>	The frame's background color. See Section 5.3, "Colors" (p. 10).
<code>bd</code> or <code>borderwidth</code>	Width of the frame's border. The default is 0 (no border). For permitted values, see Section 5.1, "Dimensions" (p. 9).
<code>cursor</code>	The cursor used when the mouse is within the frame widget; see Section 5.8, "Cursors" (p. 13).
<code>height</code>	The vertical dimension of the new frame. This will be ignored unless you also call <code>.grid_propagate(0)</code> on the frame; see Section 4.2, "Other grid management methods" (p. 7).
<code>highlightbackground</code>	Color of the focus highlight when the frame does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight when the frame has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>padx</code>	Normally, a <code>Frame</code> fits tightly around its contents. To add <i>N</i> pixels of horizontal space inside the frame, set <code>padx=N</code> .
<code>pady</code>	Used to add vertical space inside a frame. See <code>padx</code> above.
<code>relief</code>	The default relief for a frame is <code>tk.FLAT</code> , which means the frame will blend in with its surroundings. To put a border around a frame, set its <code>borderwidth</code> to a positive value and set its relief to one of the standard relief types; see Section 5.6, "Relief styles" (p. 12).
<code>takefocus</code>	Normally, frame widgets are not visited by input focus (see Section 53, "Focus: routing keyboard input" (p. 155) for an overview of this topic). However, you can set <code>takefocus=1</code> if you want the frame to receive keyboard input. To handle such input, you will need to create bindings for keyboard events; see Section 54, "Events" (p. 157) for more on events and bindings.
<code>width</code>	The horizontal dimension of the new frame. See Section 5.1, "Dimensions" (p. 9). This value be ignored unless you also call <code>.grid_propagate(0)</code> on the frame; see Section 4.2, "Other grid management methods" (p. 7).

12. The Label widget

Label widgets can display one or more lines of text in the same style, or a bitmap or image. To create a label widget in a root window or frame `parent`:

```
w = tk.Label(parent, option, ...)
```

The constructor returns the new `Label` widget. Options include:

Table 20. Label widget options

<code>activebackground</code>	Background color to be displayed when the mouse is over the widget.
<code>activeforeground</code>	Foreground color to be displayed when the mouse is over the widget.
<code>anchor</code>	This option controls where the text is positioned if the widget has more space than the text needs. The default is <code>anchor=tk.CENTER</code> , which centers the text in the available space. For other values, see Section 5.5, “Anchors” (p. 12). For example, if you use <code>anchor=tk.NW</code> , the text would be positioned in the upper left-hand corner of the available space.
<code>bg or background</code>	The background color of the label area. See Section 5.3, “Colors” (p. 10).
<code>bitmap</code>	Set this option equal to a bitmap or image object and the label will display that graphic. See Section 5.7, “Bitmaps” (p. 12) and Section 5.9, “Images” (p. 14).
<code>bd or borderwidth</code>	Width of the border around the label; see Section 5.1, “Dimensions” (p. 9). The default value is two pixels.
<code>compound</code>	If you would like the <code>Label</code> widget to display both text and a graphic (either a bitmap or an image), the <code>compound</code> option specifies the relative orientation of the graphic relative to the text. Values may be any of <code>tk.LEFT</code> , <code>tk.RIGHT</code> , <code>tk.CENTER</code> , <code>tk.BOTTOM</code> , or <code>tk.TOP</code> . For example, if you specify <code>compound=BOTTOM</code> , the graphic will be displayed below the text.
<code>cursor</code>	Cursor that appears when the mouse is over this label. See Section 5.8, “Cursors” (p. 13).
<code>disabledforeground</code>	The foreground color to be displayed when the widget's <code>state</code> is <code>tk.DISABLED</code> .
<code>font</code>	If you are displaying text in this label (with the <code>text</code> or <code>textvariable</code> option, the <code>font</code> option specifies in what font that text will be displayed. See Section 5.4, “Type fonts” (p. 10).
<code>fg or foreground</code>	If you are displaying text or a bitmap in this label, this option specifies the color of the text. If you are displaying a bitmap, this is the color that will appear at the position of the 1-bits in the bitmap. See Section 5.3, “Colors” (p. 10).
<code>height</code>	Height of the label in <i>lines</i> (not pixels!). If this option is not set, the label will be sized to fit its contents.
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus.
<code>highlightcolor</code>	The color of the focus highlight when the widget has focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>image</code>	To display a static image in the label widget, set this option to an image object. See Section 5.9, “Images” (p. 14).
<code>justify</code>	Specifies how multiple lines of text will be aligned with respect to each other: <code>tk.LEFT</code> for flush left, <code>tk.CENTER</code> for centered (the default), or <code>tk.RIGHT</code> for right-justified.
<code>padx</code>	Extra space added to the left and right of the text within the widget. Default is 1.
<code>pady</code>	Extra space added above and below the text within the widget. Default is 1.

<code>relief</code>	Specifies the appearance of a decorative border around the label. The default is <code>tk.FLAT</code> ; for other values, see Section 5.6, “Relief styles” (p. 12).
<code>state</code>	By default, an <code>Entry</code> widget is in the <code>tk.NORMAL</code> state. Set this option to <code>tk.DISABLED</code> to make it unresponsive to mouse events. The state will be <code>tk.ACTIVE</code> when the mouse is over the widget.
<code>takefocus</code>	Normally, focus does not cycle through <code>Label</code> widgets; see Section 53, “Focus: routing keyboard input” (p. 155). If you want this widget to be visited by the focus, set <code>takefocus=1</code> .
<code>text</code>	To display one or more lines of text in a label widget, set this option to a string containing the text. Internal newlines ('\\n') will force a line break.
<code>textvariable</code>	To slave the text displayed in a label widget to a control variable of class <code>StringVar</code> , set this option to that variable. See Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>underline</code>	You can display an underline (_) below the <i>n</i> th letter of the text, counting from 0, by setting this option to <i>n</i> . The default is <code>underline=-1</code> , which means no underlining.
<code>width</code>	Width of the label in <i>characters</i> (not pixels!). If this option is not set, the label will be sized to fit its contents.
<code>wraplength</code>	You can limit the number of characters in each line by setting this option to the desired number. The default value, 0, means that lines will be broken only at newlines.

There are no special methods for label widgets other than the common ones (see Section 26, “Universal widget methods” (p. 97)).

13. The `LabelFrame` widget

The `LabelFrame` widget, like the `Frame` widget, is a spatial container—a rectangular area that can contain other widgets. However, unlike the `Frame` widget, the `LabelFrame` widget allows you to display a label as part of the border around the area.



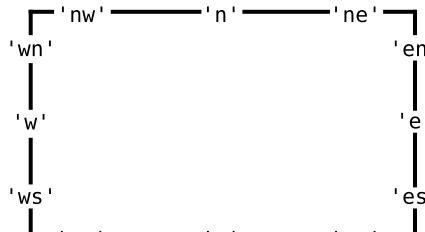
Here is an example of a `LabelFrame` widget containing two `Button` widgets. Note that the label “Important controls” interrupts the border. This widget illustrates the default `GROOVE` relief (see Section 5.6, “Relief styles” (p. 12)) and the default '`nw`' label anchor, which positions the label at the left side of the top of the frame.

To create a new `LabelFrame` widget inside a root window or frame `parent`:

```
w = tk.LabelFrame(parent, option, ...)
```

This constructor returns the new `LabelFrame` widget. Options:

Table 21. LabelFrame widget options

<code>bg</code> or <code>background</code>	The background color to be displayed inside the widget; see Section 5.3, “Colors” (p. 10).
<code>bd</code> or <code>borderwidth</code>	Width of the border drawn around the perimeter of the widget; see Section 5.1, “Dimensions” (p. 9). The default value is two pixels.
<code>cursor</code>	Selects the cursor that appears when the mouse is over the widget; see Section 5.8, “Cursors” (p. 13).
<code>fg</code> or <code>foreground</code>	Color to be used for the label text.
<code>height</code>	The vertical dimension of the new frame. This will be ignored unless you also call <code>.grid_propagate(0)</code> on the frame; see Section 4.2, “Other grid management methods” (p. 7).
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus.
<code>highlightcolor</code>	The color of the focus highlight when the widget has focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>labelanchor</code>	Use this option to specify the position of the label on the widget's border. The default position is ' <code>nw</code> ', which places the label at the left end of the top border. For the nine possible label positions, refer to this diagram: 
<code>labelwidget</code>	Instead of a text label, you can use any widget as the label by passing that widget as the value of this option. If you supply both <code>labelwidget</code> and <code>text</code> options, the <code>text</code> option is ignored.
<code>padx</code>	Use this option to add additional padding inside the left and right sides of the widget's frame. The value is in pixels.
<code>pady</code>	Use this option to add additional padding inside the top and bottom of the widget's frame. The value is in pixels.
<code>relief</code>	This option controls the appearance of the border around the outside of the widget. The default style is <code>tk.GROOVE</code> ; for other values, see Section 5.6, “Relief styles” (p. 12).
<code>takefocus</code>	Normally, the widget will not receive focus; supply a <code>True</code> value to this option to make the widget part of the focus traversal sequence. For more information, see Section 53, “Focus: routing keyboard input” (p. 155).
<code>text</code>	Text of the label.
<code>width</code>	The horizontal dimension of the new frame. This will be ignored unless you also call <code>.grid_propagate(0)</code> on the frame; see Section 4.2, “Other grid management methods” (p. 7).

14. The Listbox widget

The purpose of a listbox widget is to display a set of lines of text. Generally they are intended to allow the user to select one or more items from a list. All the lines of text use the same font. If you need something more like a text editor, see Section 24, “The Text widget” (p. 82).

To create a new listbox widget inside a root window or frame *parent*:

```
w = tk.Listbox(parent, option, ...)
```

This constructor returns the new `Listbox` widget. Options:

Table 22. Listbox widget options

<code>activestyle</code>	This option specifies the appearance of the active line. It may have any of these values: 'underline' The active line is underlined. This is the default option. 'dotbox' The active line is enclosed in a dotted line on all four sides. 'none' The active line is given no special appearance.
<code>bg</code> or <code>background</code>	The background color in the listbox.
<code>bd</code> or <code>borderwidth</code>	The width of the border around the listbox. Default is two pixels. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>cursor</code>	The cursor that appears when the mouse is over the listbox. See Section 5.8, “Cursors” (p. 13).
<code>disabledforeground</code>	The color of the text in the listbox when its <code>state</code> is <code>tk.DISABLED</code> .
<code>exportselection</code>	By default, the user may select text with the mouse, and the selected text will be exported to the clipboard. To disable this behavior, use <code>exportselection=0</code> .
<code>font</code>	The font used for the text in the listbox. See Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	The color used for the text in the listbox. See Section 5.3, “Colors” (p. 10).
<code>height</code>	Number of <i>lines</i> (not pixels!) shown in the listbox. Default is 10.
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight when the widget has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>listvariable</code>	A <code>StringVar</code> that is connected to the complete list of values in the listbox (see Section 52, “Control variables: the values behind the widgets” (p. 153)). If you call the <code>.get()</code> method of the <code>listvariable</code> , you will get back a <i>string</i> of the form " <code>('v₀', 'v₁', ...)</code> ", where each v_i is the contents of one line of the listbox. To change the entire set of lines in the listbox at once, call <code>.set(s)</code> on the <code>listvariable</code> , where <i>s</i> is a string containing the line values with spaces between them.

	<p>For example, if <code>listCon</code> is a <code>StringVar</code> associated with a listbox's <code>listvariable</code> option, this call would set the listbox to contain three lines:</p> <pre>listCon.set('ant bee cicada')</pre> <p>This call would return the string "('ant', 'bee', 'cicada')":</p> <pre>listCon.get()</pre>
<code>relief</code>	Selects three-dimensional border shading effects. The default is <code>tk.SUNKEN</code> . For other values, see Section 5.6, “Relief styles” (p. 12).
<code>selectbackground</code>	The background color to use displaying selected text.
<code>selectborderwidth</code>	The width of the border to use around selected text. The default is that the selected item is shown in a solid block of color <code>selectbackground</code> ; if you increase the <code>selectborderwidth</code> , the entries are moved farther apart and the selected entry shows <code>tk.RAISED</code> relief (see Section 5.6, “Relief styles” (p. 12)).
<code>selectforeground</code>	The foreground color to use displaying selected text.
<code>selectmode</code>	<p>Determines how many items can be selected, and how mouse drags affect the selection:</p> <ul style="list-style-type: none"> • <code>tk.BROWSE</code>: Normally, you can only select one line out of a listbox. If you click on an item and then drag to a different line, the selection will follow the mouse. This is the default. • <code>tk.SINGLE</code>: You can only select one line, and you can't drag the mouse—wherever you click button 1, that line is selected. • <code>tk.MULTIPLE</code>: You can select any number of lines at once. Clicking on any line toggles whether or not it is selected. • <code>tk.EXTENDED</code>: You can select any adjacent group of lines at once by clicking on the first line and dragging to the last line.
<code>state</code>	By default, a listbox is in the <code>tk.NORMAL</code> state. To make the listbox unresponsive to mouse events, set this option to <code>tk.DISABLED</code> .
<code>takefocus</code>	Normally, the focus will tab through listbox widgets. Set this option to 0 to take the widget out of the sequence. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>width</code>	The width of the widget in <i>characters</i> (not pixels!). The width is based on an average character, so some strings of this length in proportional fonts may not fit. The default is 20.
<code>xscrollcommand</code>	If you want to allow the user to scroll the listbox horizontally, you can link your listbox widget to a horizontal scrollbar. Set this option to the <code>.set</code> method of the scrollbar. See Section 14.1, “Scrolling a Listbox widget” (p. 56) for more on scrollable listbox widgets.
<code>yscrollcommand</code>	If you want to allow the user to scroll the listbox vertically, you can link your listbox widget to a vertical scrollbar. Set this option to the <code>.set</code> method of the scrollbar. See Section 14.1, “Scrolling a Listbox widget” (p. 56).

A special set of index forms is used for many of the methods on listbox objects:

- If you specify an index as an integer, it refers to the line in the listbox with that index, counting from 0.
- Index `tk.END` refers to the last line in the listbox.
- Index `tk.ACTIVE` refers to the selected line. If the listbox allows multiple selections, it refers to the line that was last selected.
- An index string of the form '`@x,y`' refers to the line closest to coordinate (x,y) relative to the widget's upper left corner.

Methods on `Listbox` objects include:

.activate(*index*)

Selects the line specified by the given *index*.

.bbox(*index*)

Returns the bounding box of the line specified by *index* as a 4-tuple `(xoffset, yoffset, width, height)`, where the upper left pixel of the box is at `(xoffset, yoffset)` and the `width` and `height` are given in pixels. The returned `width` value includes only the part of the line occupied by text.

If the line specified by the *index* argument is not visible, this method returns `None`. If it is partially visible, the returned bounding box may extend outside the visible area.

.curselection()

Returns a tuple containing the line numbers of the selected element or elements, counting from 0. If nothing is selected, returns an empty tuple.

.delete(*first*, *last=None*)

Deletes the lines whose indices are in the range `[first, last]`, **inclusive** (contrary to the usual Python idiom, where deletion stops short of the last index), counting from 0. If the second argument is omitted, the single line with index *first* is deleted.

.get(*first*, *last=None*)

Returns a tuple containing the text of the lines with indices from *first* to *last*, inclusive. If the second argument is omitted, returns the text of the line closest to *first*.

.index(*i*)

If possible, positions the visible part of the listbox so that the line containing index *i* is at the top of the widget.

.insert(*index*, **elements*)

Insert one or more new lines into the listbox before the line specified by *index*. Use `tk.END` as the first argument if you want to add new lines to the end of the listbox.

.itemconfig(*index*, *option*=*value*, ...)

Change a configuration option for the line specified by *index*. Option names include:

background

The background color of the given line.

foreground

The text color of the given line.

`selectbackground`

The background color of the given line when it is selected.

`selectforeground`

The text color of the given line when it is selected.

`.nearest(y)`

Return the index of the visible line closest to the y-coordinate *y* relative to the listbox widget.

`.scan_dragto(x, y)`

See `scan_mark` below.

`.scan_mark(x, y)`

Use this method to implement scanning—fast steady scrolling—of a listbox. To get this feature, bind some mouse button event to a handler that calls `scan_mark` with the current mouse position. Then bind the `<Motion>` event to a handler that calls `scan_dragto` with the current mouse position, and the listbox will be scrolled at a rate proportional to the distance between the position recorded by `scan_mark` and the current position.

`.see(index)`

Adjust the position of the listbox so that the line referred to by *index* is visible.

`.selection_anchor(index)`

Place the “selection anchor” on the line selected by the *index* argument. Once this anchor has been placed, you can refer to it with the special index form `tk.ANCHOR`.

For example, for a listbox named `lbox`, this sequence would select lines 3, 4, and 5:

```
lbox.selection_anchor(3)
lbox.selection_set(tk.ANCHOR,5)
```

`.selection_clear(first, last=None)`

Unselects all of the lines between indices *first* and *last*, inclusive. If the second argument is omitted, unselects the line with index *first*.

`.selection_includes(index)`

Returns 1 if the line with the given *index* is selected, else returns 0.

`.selection_set(first, last=None)`

Selects all of the lines between indices *first* and *last*, inclusive. If the second argument is omitted, selects the line with index *first*.

`.size()`

Returns the number of lines in the listbox.

`.xview()`

To make the listbox horizontally scrollable, set the `command` option of the associated horizontal scrollbar to this method. See Section 14.1, “Scrolling a Listbox widget” (p. 56).

`.xview_moveto(fraction)`

Scroll the listbox so that the leftmost *fraction* of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].

`.xview_scroll(number, what)`

Scrolls the listbox horizontally. For the *what* argument, use either `tk.UNITS` to scroll by characters, or `tk.PAGES` to scroll by pages, that is, by the width of the listbox. The *number* argument tells how many to scroll; negative values move the text to the right within the listbox, positive values leftward.

.yview()

To make the listbox vertically scrollable, set the *command* option of the associated vertical scrollbar to this method. See Section 14.1, “Scrolling a Listbox widget” (p. 56).

.yview_moveto(*fraction*)

Scroll the listbox so that the top *fraction* of the width of its longest line is outside the left side of the listbox. Fraction is in the range [0,1].

.yview_scroll(*number*, *what*)

Scrolls the listbox vertically. For the *what* argument, use either `tk.UNITS` to scroll by lines, or `tk.PAGES` to scroll by pages, that is, by the height of the listbox. The *number* argument tells how many to scroll; negative values move the text downward inside the listbox, and positive values move the text up.

14.1. Scrolling a Listbox widget

Here is a code fragment illustrating the creation and linking of a listbox to both a horizontal and a vertical scrollbar.

```
self.yScroll = tk.Scrollbar(self, orient=tk.VERTICAL)
self.yScroll.grid(row=0, column=1, sticky=tk.N+tk.S)

self.xScroll = tk.Scrollbar(self, orient=tk.HORIZONTAL)
self.xScroll.grid(row=1, column=0, sticky=tk.E+tk.W)

self.listbox = tk.Listbox(self,
    xscrollcommand=self.xScroll.set,
    yscrollcommand=self.yScroll.set)
self.listbox.grid(row=0, column=0, sticky=tk.N+tk.S+tk.E+tk.W)
self.xScroll['command'] = self.listbox.xview
self.yScroll['command'] = self.listbox.yview
```

15. The Menu widget

“Drop-down” menus are a popular way to present the user with a number of choices, yet take up minimal space on the face of the application when the user is not making a choice.

- A *menubutton* is the part that always appears on the application.
- A *menu* is the list of choices that appears only after the user clicks on the menubutton.
- To select a choice, the user can drag the mouse from the menubutton down onto one of the choices. Alternatively, they can click and release the menubutton: the choices will appear and stay until the user clicks one of them.
- The Unix version of *Tkinter* (at least) supports “tear-off menus.” If you as the designer wish it, a dotted line will appear above the choices. The user can click on this line to “tear off” the menu: a new, separate, independent window appears containing the choices.

Refer to Section 16, “The Menubutton widget” (p. 61), below, to see how to create a menubutton and connect it to a menu widget. First let’s look at the **Menu** widget, which displays the list of choices.

The choices displayed on a menu may be any of these things:

- A simple command: a text string (or image) that the user can select to perform some operation.

- A *cascade*: a text string or image that the user can select to show another whole menu of choices.
- A checkbutton (see Section 9, “The Checkbutton widget” (p. 38)).
- A group of radiobuttons (see Section 20, “The Radiobutton widget” (p. 68)).

To create a menu widget, you must first have created a **Menubutton**, which we will call `mb`:

```
w = tk.Menu(mb, option, ...)
```

This constructor returns the new **Menu** widget. Options include:

Table 23. Menu widget options

<code>activebackground</code>	The background color that will appear on a choice when it is under the mouse. See Section 5.3, “Colors” (p. 10).
<code>activeborderwidth</code>	Specifies the width of a border drawn around a choice when it is under the mouse. Default is 1 pixel. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>activeforeground</code>	The foreground color that will appear on a choice when it is under the mouse.
<code>bg</code> or <code>background</code>	The background color for choices not under the mouse.
<code>bd</code> or <code>borderwidth</code>	The width of the border around all the choices; see Section 5.1, “Dimensions” (p. 9). The default is one pixel.
<code>cursor</code>	The cursor that appears when the mouse is over the choices, but only when the menu has been torn off. See Section 5.8, “Cursors” (p. 13).
<code>disabledforeground</code>	The color of the text for items whose <code>state</code> is <code>tk.DISABLED</code> .
<code>font</code>	The default font for textual choices. See Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	The foreground color used for choices not under the mouse.
<code>postcommand</code>	You can set this option to a procedure, and that procedure will be called every time someone brings up this menu.
<code>relief</code>	The default 3-D effect for menus is <code>relief=tk.RAISED</code> . For other options, see Section 5.6, “Relief styles” (p. 12).
<code>selectcolor</code>	Specifies the color displayed in checkbuttons and radiobuttons when they are selected.
<code>tearoff</code>	Normally, a menu can be torn off: the first position (position 0) in the list of choices is occupied by the tear-off element, and the additional choices are added starting at position 1. If you set <code>tearoff=0</code> , the menu will not have a tear-off feature, and choices will be added starting at position 0.
<code>tearoffcommand</code>	If you would like your program to be notified when the user clicks on the tear-off entry in a menu, set this option to your procedure. It will be called with two arguments: the window ID of the parent window, and the window ID of the new tear-off menu’s root window.
<code>title</code>	Normally, the title of a tear-off menu window will be the same as the text of the menubutton or cascade that lead to this menu. If you want to change the title of that window, set the <code>title</code> option to that string.

These methods are available on **Menu** objects. The ones that create choices on the menu have their own particular options; see Section 15.1, “Menu item creation (`option`) options” (p. 59).

.add(*kind*, *option*, ...)

Add a new element of the given *kind* as the next available choice in this menu. The *kind* argument may be any of 'cascade', 'checkbutton', 'command', 'radiobutton', or 'separator'. Depending on the *kind* argument, this method is equivalent to .add_cascade(), .add_checkbutton(), and so on; refer to those methods below for details.

.add_cascade(*option*, ...)

Add a new cascade element as the next available choice in this menu. Use the *menu* option in this call to connect the cascade to the next level's menu, an object of type **Menu**.

.add_checkbutton(*option*, ...)

Add a new checkbutton as the next available choice in self. The options allow you to set up the checkbutton much the same way as you would set up a **Checkbutton** object; see Section 15.1, "Menu item creation (*option*) options" (p. 59).

.add_command(*option*, ...)

Add a new command as the next available choice in self. Use the *label*, *bitmap*, or *image* option to place text or an image on the menu; use the *command* option to connect this choice to a procedure that will be called when this choice is picked.

.add_radiobutton(*option*, ...)

Add a new radiobutton as the next available choice in self. The options allow you to set up the radiobutton in much the same way as you would set up a **Radiobutton** object; see Section 20, "The Radiobutton widget" (p. 68).

.add_separator()

Add a separator after the last currently defined option. This is just a ruled horizontal line you can use to set off groups of choices. Separators are counted as choices, so if you already have three choices, and you add a separator, the separator will occupy position 3 (counting from 0).

.delete(*index1*, *index2=None*)

This method deletes the choices numbered from *index1* through *index2*, inclusive. To delete one choice, omit the *index2* argument. You can't use this method to delete a tear-off choice, but you can do that by setting the menu object's *tearoff* option to 0.

.entrycget(*index*, *option*)

To retrieve the current value of some *option* for a choice, call this method with *index* set to the index of that choice and *option* set to the name of the desired option.

.entryconfigure(*index*, *option*, ...)

To change the current value of some *option* for a choice, call this method with *index* set to the index of that choice and one or more *option=value* arguments.

.index(*i*)

Returns the position of the choice specified by index *i*. For example, you can use .index(`tk.END`) to find the index of the last choice (or `None` if there are no choices).

.insert_cascade(*index*, *option*, ...)

Inserts a new cascade at the position given by *index*, counting from 0. Any choices after that position move down one. The options are the same as for .add_cascade(), above.

.insert_checkbutton(*index*, *option*, ...)

Insert a new checkbutton at the position specified by *index*. Options are the same as for .add_checkbutton(), above.

.insert_command(*index*, *option*, ...)

Insert a new command at position *index*. Options are the same as for .add_command(), above.

.insert_radiobutton(*index*, *option*, ...)

Insert a new radiobutton at position *index*. Options are the same as for `.add_radiobutton()`, above.

.insert_separator(*index*)

Insert a new separator at the position specified by *index*.

.invoke(*index*)

Calls the `command` callback associated with the choice at position *index*. If a checkbutton, its state is toggled between set and cleared; if a radiobutton, that choice is set.

.post(*x*, *y*)

Display this menu at position (*x*, *y*) relative to the root window.

.type(*index*)

Returns the type of the choice specified by *index*: either `tk.CASCADE`, `tk.CHECKBUTTON`, `tk.COMMAND`, `tk.RADIOBUTTON`, `tk.SEPARATOR`, or `tk.TEAROFF`.

.yposition(*n*)

For the *n*th menu choice, return the vertical offset in pixels relative to the menu's top. The purpose of this method is to allow you to place a popup menu precisely relative to the current mouse position.

15.1. Menu item creation (*option*) options

Wherever the menu methods described above allow a `option`, you may apply a value to any of the option names below by using the option name as a keyword argument with the desired value. For example, to make a command's text appear with red letters, use “`foreground='red'`” as an option to the `add_command` method call.

Table 24. Menu item *option* values

accelerator	To display an “accelerator” keystroke combination on the right side of a menu choice, use the option “ <code>accelerator=s</code> ” where <i>s</i> is a string containing the characters to be displayed. For example, to indicate that a command has <i>Control-X</i> as its accelerator, use the option “ <code>accelerator='^X'</code> ”. Note that this option does not actually implement the accelerator; use a keystroke binding to do that.
activebackground	The background color used for choices when they are under the mouse.
activeforeground	The foreground color used for choices when they are under the mouse.
background	The background color used for choices when they are <i>not</i> under the mouse. Note that this <i>cannot</i> be abbreviated as <code>bg</code> .
bitmap	Display a bitmap for this choice; see Section 5.7, “Bitmaps” (p. 12).
columnbreak	Normally all the choices are displayed in one long column. If you set <code>columnbreak=1</code> , this choice will start a new column to the right of the one containing the previous choice.
columnbreak	Use option “ <code>columnbreak=True</code> ” to start a new column of choices with this choice.
command	A procedure to be called when this choice is activated.
compound	If you want to display both text and a graphic (either a bitmap or an image) on a menu choice, use this <code>option</code> to specify the location of the graphic relative to the text. Values may be any of <code>tk.LEFT</code> , <code>tk.RIGHT</code> , <code>tk.TOP</code> , <code>tk.BOTTOM</code> , <code>tk.CENTER</code> ,

	or <code>tk.NONE</code> . For example, a value of “ <code>compound=tk.TOP</code> ” would position the graphic above the text.
<code>font</code>	The font used to render the <code>label</code> text. See Section 5.4, “Type fonts” (p. 10)
<code>foreground</code>	The foreground color used for choices when they are <i>not</i> under the mouse. Note that this <i>cannot</i> be abbreviated as <code>fg</code> .
<code>hidemargin</code>	By default, a small margin separates adjacent choices in a menu. Use the <code>option “hidemargin=True”</code> to suppress this margin. For example, if your choices are color swatches on a palette, this option will make the swatches touch without any other intervening color.
<code>image</code>	Display an image for this choice; see Section 5.9, “Images” (p. 14).
<code>label</code>	The text string to appear for this choice.
<code>menu</code>	This option is used only for cascade choices. Set it to a <code>Menu</code> object that displays the next level of choices.
<code>offvalue</code>	Normally, the control variable for a checkbutton is set to <code>0</code> when the checkbutton is off. You can change the off value by setting this option to the desired value. See Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>onvalue</code>	Normally, the control variable for a checkbutton is set to <code>1</code> when the checkbutton is on. You can change the on value by setting this option to the desired value.
<code>selectcolor</code>	Normally, the color displayed in a set checkbutton or radiobutton is red. Change that color by setting this option to the color you want; see Section 5.3, “Colors” (p. 10).
<code>selectimage</code>	If you are using the <code>image</code> option to display a graphic instead of text on a menu radiobutton or checkbutton, if you use <code>selectimage=I</code> , image <code>I</code> will be displayed when the item is selected.
<code>state</code>	Normally, all choices react to mouse clicks, but you can set <code>state=tk.DISABLED</code> to gray it out and make it unresponsive. This <code>option</code> will be <code>tk.ACTIVE</code> when the mouse is over the choice.
<code>underline</code>	Normally none of the letters in the <code>label</code> are underlined. Set this option to the index of a letter to underline that letter.
<code>value</code>	Specifies the value of the associated control variable (see Section 52, “Control variables: the values behind the widgets” (p. 153)) for a radiobutton. This can be an integer if the control variable is an <code>IntVar</code> , or a string if the control variable is a <code>StringVar</code> .
<code>variable</code>	For checkbuttons or radiobuttons, this option should be set to the control variable associated with the checkbutton or group of radiobuttons. See Section 52, “Control variables: the values behind the widgets” (p. 153).

15.2. Top-level menus

Especially under MacOS, it is sometimes desirable to create menus that are shown as part of the top-level window. To do this, follow these steps.

1. Using any widget `W`, obtain the top-level window by using the `W.winfo_toplevel()` method.
2. Create a `Menu` widget, using the top-level window as the first argument.
3. Items added to this `Menu` widget will be displayed across the top of the application.

Here is a brief example. Assume that `self` is the application instance, an instance of a class that inherits from `Frame`. This code would create a top-level menu choice named “*Help*” with one choice named “*About*” that calls a handler named `self.__aboutHandler`:

```
top = self.winfo_toplevel()
self.menuBar = tk.Menu(top)
top['menu'] = self.menuBar

self.subMenu = tk.Menu(self.menuBar)
self.menuBar.add_cascade(label='Help', menu=self.subMenu)
self.subMenu.add_command(label='About', command=self.__aboutHandler)
```

There is some variation in behavior depending on your platform.

- Under Windows or Unix systems, the top-level menu choices appear at the top of your application's main window.
- Under MacOS X, the top-level menu choices appear at the top of the screen when the application is active, right where Mac users expect to see them.

You must use the `.add_cascade()` method for all the items you want on the top menu bar. Calls to `.add_checkbutton()`, `.add_command()`, or `.add_radiobutton()` will be ignored.

16. The Menubutton widget

A menubutton is the part of a drop-down menu that stays on the screen all the time. Every menubutton is associated with a `Menu` widget (see above) that can display the choices for that menubutton when the user clicks on it.

To create a menubutton within a root window or frame `parent`:

```
w = tk.Menubutton(parent, option, ...)
```

The constructor returns the new Menubutton widget. Options:

Table 25. Menubutton widget options

<code>activebackground</code>	The background color when the mouse is over the menubutton. See Section 5.3, “Colors” (p. 10).
<code>activeforeground</code>	The foreground color when the mouse is over the menubutton.
<code>anchor</code>	This option controls where the text is positioned if the widget has more space than the text needs. The default is <code>anchor=tk.CENTER</code> , which centers the text. For other options, see Section 5.5, “Anchors” (p. 12). For example, if you use <code>anchor=tk.W</code> , the text would be centered against the left side of the widget.
<code>bg</code> or <code>background</code>	The background color when the mouse is not over the menubutton.
<code>bitmap</code>	To display a bitmap on the menubutton, set this option to a bitmap name; see Section 5.7, “Bitmaps” (p. 12).
<code>bd</code> or <code>borderwidth</code>	Width of the border around the menubutton. Default is two pixels. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>compound</code>	If you specify both text and a graphic (either a bitmap or an image), this option specifies where the graphic appears relative to the text. Possible values are <code>tk.NONE</code> (the default value), <code>tk.TOP</code> , <code>tk.BOTTOM</code> , <code>tk.LEFT</code> ,

	<code>tk.RIGHT</code> , and <code>tk.CENTER</code> . For example, <code>compound=tk.RIGHT</code> would position the graphic to the right of the text. If you specify <code>compound=tk.NONE</code> , the graphic is displayed but the <code>text</code> (if any) is not.
<code>cursor</code>	The cursor that appears when the mouse is over this menubutton. See Section 5.8, “Cursors” (p. 13).
<code>direction</code>	Normally, the menu will appear below the menubutton. Set <code>direction=tk.LEFT</code> to display the menu to the left of the button; use <code>direction=tk.RIGHT</code> to display the menu to the right of the button; or use <code>direction='above'</code> to place the menu above the button.
<code>disabledforeground</code>	The foreground color shown on this menubutton when it is disabled.
<code>fg</code> or <code>foreground</code>	The foreground color when the mouse is not over the menubutton.
<code>font</code>	Specifies the font used to display the <code>text</code> ; see Section 5.4, “Type fonts” (p. 10).
<code>height</code>	The height of the menubutton in <i>lines</i> of text (not pixels!). The default is to fit the menubutton’s size to its contents.
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight when the widget has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>image</code>	To display an image on this menubutton, set this option to the image object. See Section 5.9, “Images” (p. 14).
<code>justify</code>	This option controls where the text is located when the text doesn’t fill the menubutton: use <code>justify=tk.LEFT</code> to left-justify the text (this is the default); use <code>justify=tk.CENTER</code> to center it, or <code>justify=tk.RIGHT</code> to right-justify.
<code>menu</code>	To associate the menubutton with a set of choices, set this option to the <code>Menu</code> object containing those choices. That menu object must have been created by passing the associated menubutton to the constructor as its first argument. See below for an example showing how to associate a menubutton and menu.
<code>padx</code>	How much space to leave to the left and right of the text of the menubutton. Default is 1.
<code>pady</code>	How much space to leave above and below the text of the menubutton. Default is 1.
<code>relief</code>	Normally, menubuttons will have <code>tk.RAISED</code> appearance. For other 3-d effects, see Section 5.6, “Relief styles” (p. 12).
<code>state</code>	Normally, menubuttons respond to the mouse. Set <code>state=tk.DISABLED</code> to gray out the menubutton and make it unresponsive.
<code>takefocus</code>	Normally, menubuttons do not take keyboard focus (see Section 53, “Focus: routing keyboard input” (p. 155)). Use <code>takefocus=True</code> to add the menubutton to the focus traversal order.
<code>text</code>	To display text on the menubutton, set this option to the string containing the desired text. Newlines ('\\n') within the string will cause line breaks.
<code>textvariable</code>	You can associate a control variable of class <code>StringVar</code> with this menubutton. Setting that control variable will change the displayed text. See Section 52, “Control variables: the values behind the widgets” (p. 153).

<code>underline</code>	Normally, no underline appears under the text on the menubutton. To underline one of the characters, set this option to the index of that character.
<code>width</code>	Width of the menubutton in <i>characters</i> (not pixels!). If this option is not set, the label will be sized to fit its contents.
<code>wraplength</code>	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Here is a brief example showing the creation of a menubutton and its associated menu with two checkboxes:

```
self.mb = tk.Menubutton(self, text='condiments',
                       relief=RAISED)
self.mb.grid()

self.mb.menu = tk.Menu(self.mb, tearoff=0)
self.mb['menu'] = self.mb.menu

self.mayoVar = tk.IntVar()
self.ketchVar = tk.IntVar()
self.mb.menu.add_checkbutton(label='mayo',
                             variable=self.mayoVar)
self.mb.menu.add_checkbutton(label='ketchup',
                             variable=self.ketchVar)
```

This example creates a menubutton labeled `condiments`. When clicked, two checkbuttons labeled `mayo` and `ketchup` will drop down.

17. The Message widget

This widget is similar to the `Label` widget (see Section 12, “The `Label` widget” (p. 48)), but it is intended for displaying messages over multiple lines. All the text will be displayed in the same font; if you need to display text with more than one font, see Section 24, “The `Text` widget” (p. 82).

To create a new `Message` widget as the child of a root window or frame named `parent`:

```
w = tk.Message(parent, option, ...)
```

This constructor returns the new `Message` widget. Options may be any of these:

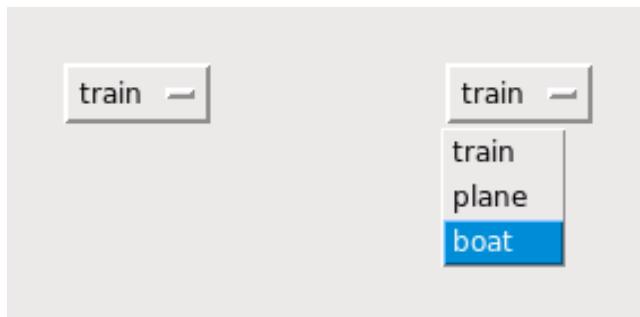
Table 26. Message widget options

<code>aspect</code>	Use this option to specify the ratio of width to height as a percentage. For example, <code>aspect=100</code> would give you a text message fit into a square; with <code>aspect=200</code> , the text area would be twice as wide as high. The default value is 150, that is, the text will be fit into a box 50% wider than it is high.
<code>bg</code> or <code>background</code>	The background color behind the text; see Section 5.3, “Colors” (p. 10).
<code>bd</code> or <code>borderwidth</code>	Width of the border around the widget; see Section 5.1, “Dimensions” (p. 9). The default is two pixels. This option is visible only when the <code>relief</code> option is not <code>tk.FLAT</code> .
<code>cursor</code>	Specifies the cursor that appears when the mouse is over the widget; see Section 5.8, “Cursors” (p. 13).

<code>font</code>	Specifies the font used to display the text in the widget; see Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	Specifies the text color; see Section 5.3, “Colors” (p. 10).
<code>highlightbackground</code>	Color of the focus highlight when the widget does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	Color shown in the focus highlight when the widget has the focus.
<code>highlightthickness</code>	Thickness of the focus highlight.
<code>justify</code>	Use this option to specify how multiple lines of text are aligned. Use <code>justify=tk.LEFT</code> to get a straight left margin; <code>justify=tk.CENTER</code> to center each line; and <code>justify=tk.RIGHT</code> to get a straight right margin.
<code>padx</code>	Use this option to add extra space inside the widget to the left and right of the text. The value is in pixels.
<code>pady</code>	Use this option to add extra space inside the widget above and below the text. The value is in pixels.
<code>relief</code>	This option specifies the appearance of the border around the outside of the widget; see Section 5.6, “Relief styles” (p. 12). The default style is <code>tk.FLAT</code> .
<code>takefocus</code>	Normally, a <code>Message</code> widget will not acquire focus (see Section 53, “Focus: routing keyboard input” (p. 155)). Use <code>takefocus=True</code> to add the widget to the focus traversal list.
<code>text</code>	The value of this option is the text to be displayed inside the widget.
<code>textvariable</code>	If you would like to be able to change the message under program control, associate this option with a <code>StringVar</code> instance (see Section 52, “Control variables: the values behind the widgets” (p. 153)). The value of this variable is the text to be displayed. If you specify both <code>text</code> and <code>textvariable</code> options, the <code>text</code> option is ignored.
<code>width</code>	Use this option to specify the width of the text area in the widget, in pixels. The default width depends on the displayed text and the value of the <code>aspect</code> option.

18. The `OptionMenu` widget

The purpose of this widget is to offer a fixed set of choices to the user in a drop-down menu.



The illustrations above shows an `OptionMenu` in two states. The left-hand example shows the widget in its initial form. The right-hand example shows how it looks when the mouse has clicked on it and dragged down to the 'boat' choice.

To create a new `OptionMenu` widget as the child of a root window or frame named *parent*:

```
w = tk.OptionMenu(parent, variable, choice1, choice2, ...)
```

This constructor returns the new `OptionMenu` widget. The `variable` is a `StringVar` instance (see Section 52, “Control variables: the values behind the widgets” (p. 153)) that is associated with the widget, and the remaining arguments are the choices to be displayed in the widget as strings.

The illustration above was created with this code snippet:

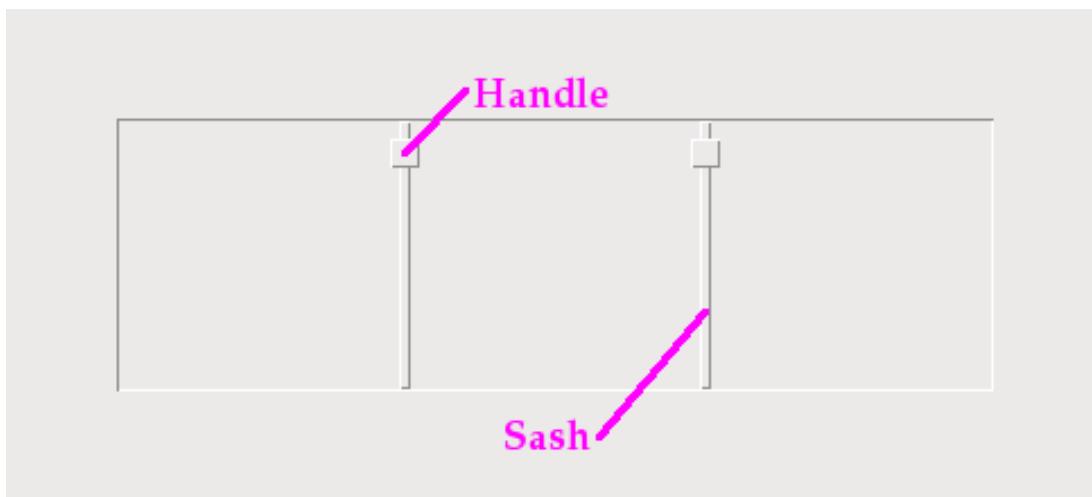
```
optionList = ('train', 'plane', 'boat')
self.v = tk.StringVar()
self.v.set(optionList[0])
self.om = tk.OptionMenu(self, self.v, *optionList)
```

To find out which choice is currently selected in an `OptionMenu` widget, the `.get()` method on the associated control variable will return that choice as a string.

19. The PanedWindow widget

The purpose of the `PanedWindow` widget is to give the application's user some control over how space is divided up within the application.

A `PanedWindow` is somewhat like a `Frame`: it is a container for *child widgets*. Each `PanedWindow` widget contains a horizontal or vertical stack of child widgets. Using the mouse, the user can drag the boundaries between the child widgets back and forth.



- You may choose to display *handles* within the widget. A handle is a small square that the user can drag with the mouse.
- You may choose to make *sashes* visible. A sash is a bar placed between the child widgets.
- A *pane* is the area occupied by one child widget.

To create a new `PanedWindow` widget as the child of a root window or frame named *parent*:

```
w = tk.PanedWindow(parent, option, ...)
```

This constructor returns the new `PanedWindow` widget. Here are the options:

Table 27. PanedWindow widget options

<code>bg</code> or <code>background</code>	The background color displayed behind the child widgets; see Section 5.3, “Colors” (p. 10).
<code>bd</code> or <code>borderwidth</code>	Width of the border around the outside of the widget; see Section 5.1, “Dimensions” (p. 9). The default is two pixels.
<code>cursor</code>	The cursor to be displayed when the mouse is over the widget; see Section 5.8, “Cursors” (p. 13).
<code>handlepad</code>	Use this option to specify the distance between the handle and the end of the sash. For <code>orient=tk.VERTICAL</code> , this is the distance between the left end of the sash and the handle; for <code>orient=tk.HORIZONTAL</code> , it is the distance between the top of the sash and the handle. The default value is eight pixels; for other values, see Section 5.1, “Dimensions” (p. 9).
<code>handlesize</code>	Use this option to specify the size of the handle, which is always a square; see Section 5.1, “Dimensions” (p. 9). The default value is eight pixels.
<code>height</code>	Specifies the height of the widget; see Section 5.1, “Dimensions” (p. 9). If you don't specify this option, the height is determined by the height of the child widgets.
<code>opaqueresize</code>	This option controls how a resizing operation works. For the default value, <code>opaqueresize=True</code> , the resizing is done continuously as the sash is dragged. If this option is set to <code>False</code> , the sash (and adjacent child widgets) stays put until the user releases the mouse button, and then it jumps to the new position.
<code>orient</code>	To stack child widgets side by side, use <code>orient=tk.HORIZONTAL</code> . To stack them top to bottom, use <code>orient=tk.VERTICAL</code> .
<code>relief</code>	Selects the relief style of the border around the widget; see Section 5.6, “Relief styles” (p. 12). The default is <code>tk.FLAT</code> .
<code>sashpad</code>	Use this option to allocate extra space on either side of each sash. The default is zero; for other values, see Section 5.1, “Dimensions” (p. 9).
<code>sashrelief</code>	This option specifies the relief style used to render the sashes; see Section 5.6, “Relief styles” (p. 12). The default style is <code>tk.FLAT</code> .
<code>sashwidth</code>	Specifies the width of the sash; see Section 5.1, “Dimensions” (p. 9). The default width is two pixels.
<code>showhandle</code>	Use <code>showhandle=True</code> to display the handles. For the default value, <code>False</code> , the user can still use the mouse to move the sashes. The handle is simply a visual cue.
<code>width</code>	Width of the widget; see Section 5.1, “Dimensions” (p. 9). If you don't specify a value, the width will be determined by the sizes of the child widgets.

To add child widgets to a `PanedWindow`, create the child widgets as children of the parent `PanedWindow`, but rather than using the `.grid()` method to register them, use the `.add()` method on the `PanedWindow`.

Here are the methods on `PanedWindow` widgets.

`.add(child[, option=value] ...)`

Use this method to add the given `child` widget as the next child of this `PanedWindow`. First create the `child` widget with the `PanedWindow` as its parent widget, but do not call the `.grid()` method to register it. Then call `.add(child)` and the child will appear inside the `PanedWindow` in the next available position.

Associated with each child is a set of configuration options that control its position and appearance. See Section 19.1, “PanedWindow child configuration options” (p. 67). You can supply these configuration options as keyword arguments to the `.add()` method. You can also set or change their values anytime with the `.paneconfig()` method, or retrieve the current value of any of these options using the `.panecget()` method; these methods are described below.

.forget(*child*)

Removes a child widget.

.identify(*x*, *y*)

For a given location (*x*, *y*) in window coordinates, this method returns a value that describes the feature at that location.

- If the feature is a child window, the method returns an empty string.
- If the feature is a sash, the method returns a tuple (*n*, 'sash') where *n* is 0 for the first sash, 1 for the second, and so on.
- If the feature is a handle, the method returns a tuple (*n*, 'handle') where *n* is 0 for the first handle, 1 for the second, and so on.

.panecget(*child*, *option*)

This method retrieves the value of a child widget configuration option, where *child* is the child widget and *option* is the name of the option as a string. For the list of child widget configuration options, see Section 19.1, “PanedWindow child configuration options” (p. 67).

.paneconfig(*child*, *option=value*, ...)

Use this method to configure options for child widgets. The options are described in Section 19.1, “PanedWindow child configuration options” (p. 67).

.panes()

This method returns a list of the child widgets, in order from left to right (for `orient=tk.HORIZONTAL`) or top to bottom (for `orient=tk.VERTICAL`).

.remove(*child*)

Removes the given *child*; this is the same action as the `.forget()` method.

.sash_coord(*index*)

This method returns the location of a sash. The *index* argument selects the sash: 0 for the sash between the first two children, 1 for the sash between the second and third child, and so forth. The result is a tuple (*x*, *y*) containing the coordinates of the upper left corner of the sash.

.sash_place(*index*, *x*, *y*)

Use this method to reposition the sash selected by *index* (0 for the first sash, and so on). The *x* and *y* coordinates specify the desired new position of the upper left corner of the sash. Tkinter ignores the coordinate orthogonal to the orientation of the widget: use the *x* value to reposition the sash for `orient=tk.HORIZONTAL`, and use the *y* coordinate to move the sash for option `orient=tk.VERTICAL`.

19.1. PanedWindow child configuration options

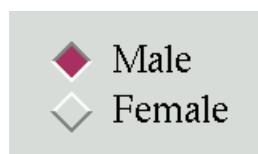
Each child of a PanedWindow has a set of configuration options that control its position and appearance. These options can be provided when a child is added with the `.add()` method, or set with the `.paneconfig()` method, or queried with the `.panecget()` methods described above.

Table 28. PanedWindow child widget options

<code>after</code>	Normally, when you <code>.add()</code> a new child to a <code>PanedWindow</code> , the new child is added after any existing child widgets. You may instead use the <code>after=w</code> option to insert the new widget at a position just after an existing child widget <code>w</code> .
<code>before</code>	When used as option <code>before=w</code> in a call to the <code>.add()</code> method, places the new widget at a position just before an existing child widget <code>w</code> .
<code>height</code>	This option specifies the desired height of the child widget; see Section 5.1, “Dimensions” (p. 9).
<code>minsize</code>	Use this option to specify a minimum size for the child widget in the direction of the <code>PanedWindow</code> ’s orientation. For <code>orient=tk.HORIZONTAL</code> , this is the minimum width; for <code>orient=tk.VERTICAL</code> , it is the minimum height. For permissible values, see Section 5.1, “Dimensions” (p. 9).
<code>padx</code>	The amount of extra space to be added to the left and right of the child widget; see Section 5.1, “Dimensions” (p. 9).
<code>pady</code>	The amount of extra space to be added above and below the child widget; see Section 5.1, “Dimensions” (p. 9).
<code>sticky</code>	This option functions like the <code>sticky</code> argument to the <code>.grid()</code> method; see Section 4.1, “The <code>.grid()</code> method” (p. 6). It specifies how to position a child widget if the pane is larger than the widget. For example, <code>sticky=tk.NW</code> would position the widget in the upper left (“northwest”) corner of the pane.
<code>width</code>	Desired width of the child widget; see Section 5.1, “Dimensions” (p. 9).

20. The Radiobutton widget

Radiobuttons are sets of related widgets that allow the user to select only one of a set of choices. Each radiobutton consists of two parts, the *indicator* and the *label*:



- The indicator is the diamond-shaped part that turns red in the selected item.
- The label is the text, although you can use an image or bitmap as the label.
- If you prefer, you can dispense with the indicator. This makes the radiobuttons look like “push-push” buttons, with the selected entry appearing sunken and the rest appearing raised.
- To form several radiobuttons into a functional group, create a single control variable (see Section 52, “Control variables: the values behind the widgets” (p. 153), below), and set the `variable` option of each radiobutton to that variable.

The control variable can be either an `IntVar` or a `StringVar`. If two or more radiobuttons share the same control variable, setting any of them will clear the others.

- Each radiobutton in a group must have a unique `value` option of the same type as the control variable. For example, a group of three radiobuttons might share an `IntVar` and have values of 0, 1, and 99. Or you can use a `StringVar` control variable and give the radiobuttons `value` options like ‘`too hot`’, ‘`too cold`’, and ‘`just right`’.

To create a new radiobutton widget as the child of a root window or frame named *parent*:

```
w = tk.Radiobutton(parent, option, ...)
```

This constructor returns the new Radiobutton widget. Options:

Table 29. Radiobutton widget options

activebackground	The background color when the mouse is over the radiobutton. See Section 5.3, “Colors” (p. 10).
activeforeground	The foreground color when the mouse is over the radiobutton.
anchor	If the widget inhabits a space larger than it needs, this option specifies where the radiobutton will sit in that space. The default is <code>anchor=tk.CENTER</code> . For other positioning options, see Section 5.5, “Anchors” (p. 12). For example, if you set <code>anchor=tk.NE</code> , the radiobutton will be placed in the top right corner of the available space.
bg or background	The normal background color behind the indicator and label.
bitmap	To display a monochrome image on a radiobutton, set this option to a bitmap; see Section 5.7, “Bitmaps” (p. 12).
bd or borderwidth	The size of the border around the indicator part itself. Default is two pixels. For possible values, see Section 5.1, “Dimensions” (p. 9).
command	A procedure to be called every time the user changes the state of this radiobutton.
compound	If you specify both text and a graphic (either a bitmap or an image), this option specifies where the graphic appears relative to the text. Possible values are <code>tk.NONE</code> (the default value), <code>tk.TOP</code> , <code>tk.BOTTOM</code> , <code>tk.LEFT</code> , <code>tk.RIGHT</code> , and <code>tk.CENTER</code> . For example, <code>compound=tk.BOTTOM</code> would position the graphic below the text. If you specify <code>compound=tk.NONE</code> , the graphic is displayed but the <code>text</code> (if any) is not.
cursor	If you set this option to a cursor name (see Section 5.8, “Cursors” (p. 13)), the mouse cursor will change to that pattern when it is over the radiobutton.
disabledforeground	The foreground color used to render the text of a disabled radiobutton. The default is a stippled version of the default foreground color.
font	The font used for the <code>text</code> . See Section 5.4, “Type fonts” (p. 10).
fg or foreground	The color used to render the <code>text</code> .
height	The number of lines (<i>not</i> pixels) of text on the radiobutton. Default is 1.
highlightbackground	The color of the focus highlight when the radiobutton does not have focus. See Section 5.3, “Focus: routing keyboard input” (p. 155).
highlightcolor	The color of the focus highlight when the radiobutton has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set <code>highlightthickness=0</code> to suppress display of the focus highlight.
image	To display a graphic image instead of text for this radiobutton, set this option to an image object. See Section 5.9, “Images” (p. 14). The image appears when the radiobutton is <i>not</i> selected; compare <code>selectimage</code> , below.
indicatoron	Normally a radiobutton displays its indicator. If you set this option to zero, the indicator disappears, and the entire widget becomes a “push-push” button that looks raised when it is cleared and sunken when it is set. You

	may want to increase the <code>borderwidth</code> value to make it easier to see the state of such a control.
<code>justify</code>	If the <code>text</code> contains multiple lines, this option controls how the text is justified: <code>tk.CENTER</code> (the default), <code>tk.LEFT</code> , or <code>tk.RIGHT</code> .
<code>offrelief</code>	If you suppress the indicator by asserting <code>indicatoron=False</code> , the <code>offrelief</code> option specifies the relief style to be displayed when the radiobutton is not selected. The default values is <code>tk.RAISED</code> .
<code>overrelief</code>	Specifies the relief style to be displayed when the mouse is over the radiobutton.
<code>padx</code>	How much space to leave to the left and right of the radiobutton and text. Default is 1.
<code>pady</code>	How much space to leave above and below the radiobutton and text. Default is 1.
<code>relief</code>	By default, a radiobutton will have <code>tk.FLAT</code> relief, so it doesn't stand out from its background. See Section 5.6, "Relief styles" (p. 12) for more 3-d effect options. You can also use <code>relief=tk.SOLID</code> , which displays a solid black frame around the radiobutton.
<code>selectcolor</code>	The color of the radiobutton when it is set. Default is red.
<code>selectimage</code>	If you are using the <code>image</code> option to display a graphic instead of text when the radiobutton is cleared, you can set the <code>selectimage</code> option to a different image that will be displayed when the radiobutton is set. See Section 5.9, "Images" (p. 14).
<code>state</code>	The default is <code>state=tk.NORMAL</code> , but you can set <code>state=tk.DISABLED</code> to gray out the control and make it unresponsive. If the cursor is currently over the radiobutton, the state is <code>tk.ACTIVE</code> .
<code>takefocus</code>	By default, the input focus (see Section 53, "Focus: routing keyboard input" (p. 155)) will pass through a radiobutton. If you set <code>takefocus=0</code> , focus will not visit this radiobutton.
<code>text</code>	The label displayed next to the radiobutton. Use newlines (' <code>\n</code> ') to display multiple lines of text.
<code>textvariable</code>	If you need to change the label on a radiobutton during execution, create a <code>StringVar</code> (see Section 52, "Control variables: the values behind the widgets" (p. 153)) to manage the current value, and set this option to that control variable. Whenever the control variable's value changes, the radiobutton's annotation will automatically change to that text as well.
<code>underline</code>	With the default value of -1, none of the characters of the text label are underlined. Set this option to the index of a character in the text (counting from zero) to underline that character.
<code>value</code>	When a radiobutton is turned on by the user, its control variable is set to its current <code>value</code> option. If the control variable is an <code>IntVar</code> , give each radiobutton in the group a different integer <code>value</code> option. If the control variable is a <code>StringVar</code> , give each radiobutton a different string <code>value</code> option.
<code>variable</code>	The control variable that this radiobutton shares with the other radiobuttons in the group; see Section 52, "Control variables: the values behind the widgets" (p. 153). This can be either an <code>IntVar</code> or a <code>StringVar</code> .

<code>width</code>	The default width of a radiobutton is determined by the size of the displayed image or text. You can set this option to a number of characters (<i>not pixels</i>) and the radiobutton will always have room for that many characters.
<code>wraplength</code>	Normally, lines are not wrapped. You can set this option to a number of characters and all lines will be broken into pieces no longer than that number.

Methods on radiobutton objects include:

.deselect()

Clears (turns off) the radiobutton.

.flash()

Flashes the radiobutton a few times between its active and normal colors, but leaves it the way it started.

.invoke()

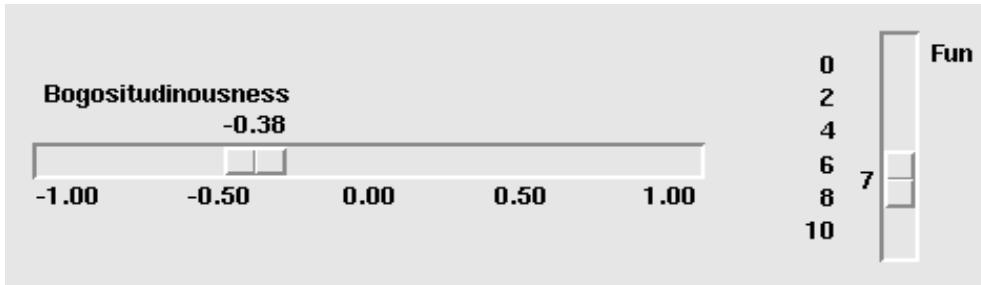
You can call this method to get the same actions that would occur if the user clicked on the radiobutton to change its state.

.select()

Sets (turns on) the radiobutton.

21. The Scale widget

The purpose of a scale widget is to allow the user to set some `int` or `float` value within a specified range. Here are two scale widgets, one horizontal and one vertical:



Each scale displays a *slider* that the user can drag along a *trough* to change the value. In the figure, the first slider is currently at -0.38 and the second at 7.

- You can drag the slider to a new value with mouse button 1.
- If you click button 1 in the trough, the slider will move one increment in that direction per click. Holding down button 1 in the trough will, after a delay, start to auto-repeat its function.
- If the scale has keyboard focus, left arrow and up arrow keystrokes will move the slider up (for vertical scales) or left (for horizontal scales). Right arrow and down arrow keystrokes will move the slider down or to the right.

To create a new scale widget as the child of a root window or frame named `parent`:

```
w = tk.Scale(parent, option, ...)
```

The constructor returns the new `Scale` widget. Options:

Table 30. Scale widget options

<code>activebackground</code>	The color of the slider when the mouse is over it. See Section 5.3, “Colors” (p. 10).
<code>bg</code> or <code>background</code>	The background color of the parts of the widget that are outside the trough.
<code>bd</code> or <code>borderwidth</code>	Width of the 3-d border around the trough and slider. Default is two pixels. For acceptable values, see Section 5.1, “Dimensions” (p. 9).
<code>command</code>	A procedure to be called every time the slider is moved. This procedure will be passed one argument, the new scale value. If the slider is moved rapidly, you may not get a callback for every possible position, but you'll certainly get a callback when it settles.
<code>cursor</code>	The cursor that appears when the mouse is over the scale. See Section 5.8, “Cursors” (p. 13).
<code>digits</code>	The way your program reads the current value shown in a scale widget is through a control variable; see Section 52, “Control variables: the values behind the widgets” (p. 153). The control variable for a scale can be an <code>IntVar</code> , a <code>DoubleVar</code> (for type <code>float</code>), or a <code>StringVar</code> . If it is a string variable, the <code>digits</code> option controls how many digits to use when the numeric scale value is converted to a string.
<code>font</code>	The font used for the label and annotations. See Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	The color of the text used for the label and annotations.
<code>from_</code>	A <code>float</code> value that defines one end of the scale's range. For vertical scales, this is the top end; for horizontal scales, the left end. The underbar (<code>_</code>) is not a typo: because <code>from</code> is a reserved word in Python, this option is spelled <code>from_</code> . The default is 0.0. See the <code>to</code> option, below, for the other end of the range.
<code>highlightbackground</code>	The color of the focus highlight when the scale does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	The color of the focus highlight when the scale has the focus.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set <code>highlightthickness=0</code> to suppress display of the focus highlight.
<code>label</code>	You can display a label within the scale widget by setting this option to the label's text. The label appears in the top left corner if the scale is horizontal, or the top right corner if vertical. The default is no label.
<code>length</code>	The length of the scale widget. This is the <code>x</code> dimension if the scale is horizontal, or the <code>y</code> dimension if vertical. The default is 100 pixels. For allowable values, see Section 5.1, “Dimensions” (p. 9).
<code>orient</code>	Set <code>orient=tk.HORIZONTAL</code> if you want the scale to run along the <code>x</code> dimension, or <code>orient=tk.VERTICAL</code> to run parallel to the <code>y</code> -axis. Default is vertical.
<code>relief</code>	With the default <code>relief=tk.FLAT</code> , the scale does not stand out from its background. You may also use <code>relief=tk.SOLID</code> to get a solid black frame around the scale, or any of the other relief types described in Section 5.6, “Relief styles” (p. 12).

<code>repeatdelay</code>	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is <code>repeatdelay=300</code> , and the units are milliseconds.
<code>repeatinterval</code>	This option controls how often the slider jumps once button 1 has been held down in the trough for at least <code>repeatdelay</code> milliseconds. For example, <code>repeatinterval=100</code> would jump the slider every 100 milliseconds.
<code>resolution</code>	Normally, the user will only be able to change the scale in whole units. Set this option to some other value to change the smallest increment of the scale's value. For example, if <code>from_=-1.0</code> and <code>to=1.0</code> , and you set <code>resolution=0.5</code> , the scale will have 5 possible values: -1.0, -0.5, 0.0, +0.5, and +1.0. All smaller movements will be ignored. Use <code>resolution=-1</code> to disable any rounding of values.
<code>showvalue</code>	Normally, the current value of the scale is displayed in text form by the slider (above it for horizontal scales, to the left for vertical scales). Set this option to 0 to suppress that label.
<code>sliderlength</code>	Normally the slider is 30 pixels along the length of the scale. You can change that length by setting the <code>sliderlength</code> option to your desired length; see Section 5.1, "Dimensions" (p. 9).
<code>sliderrelief</code>	By default, the slider is displayed with a <code>tk.RAISED</code> relief style. For other relief styles, set this option to any of the values described in Section 5.6, "Relief styles" (p. 12).
<code>state</code>	Normally, scale widgets respond to mouse events, and when they have the focus, also keyboard events. Set <code>state=tk.DISABLED</code> to make the widget unresponsive.
<code>takefocus</code>	Normally, the focus will cycle through scale widgets. Set this option to 0 if you don't want this behavior. See Section 53, "Focus: routing keyboard input" (p. 155).
<code>tickinterval</code>	Normally, no "ticks" are displayed along the scale. To display periodic scale values, set this option to a number, and ticks will be displayed on multiples of that value. For example, if <code>from_=0.0</code> , <code>to=1.0</code> , and <code>tickinterval=0.25</code> , labels will be displayed along the scale at values 0.0, 0.25, 0.50, 0.75, and 1.00. These labels appear below the scale if horizontal, to its left if vertical. Default is 0, which suppresses display of ticks.
<code>to</code>	A <code>float</code> value that defines one end of the scale's range; the other end is defined by the <code>from_</code> option, discussed above. The <code>to</code> value can be either greater than or less than the <code>from_</code> value. For vertical scales, the <code>to</code> value defines the bottom of the scale; for horizontal scales, the right end. The default value is 100.0.
<code>troughcolor</code>	The color of the trough.
<code>variable</code>	The control variable for this scale, if any; see Section 52, "Control variables: the values behind the widgets" (p. 153). Control variables may be from class <code>IntVar</code> , <code>DoubleVar</code> (for type <code>float</code>), or <code>StringVar</code> . In the latter case, the numerical value will be converted to a string. See the the <code>digits</code> option, above, for more information on this conversion.
<code>width</code>	The width of the trough part of the widget. This is the x dimension for vertical scales and the y dimension if the scale has <code>orient=tk.HORIZONTAL</code> . Default is 15 pixels.

Scale objects have these methods:

.coords(value=None)

Returns the coordinates, relative to the upper left corner of the widget, corresponding to a given value of the scale. For `value=None`, you get the coordinates of the center of the slider at its current position. To find where the slider would be if the scale's value were set to some value `x`, use `value=x`.

.get()

This method returns the current value of the scale.

.identify(x, y)

Given a pair of coordinates (`x, y`) relative to the top left corner of the widget, this method returns a string identifying what functional part of the widget is at that location. The return value may be any of these:

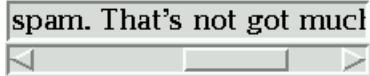
'slider'	The slider.
'trough1'	For horizontal scales, to the left of the slider; for vertical scales, above the slider.
'trough2'	For horizontal scales, to the right of the slider; for vertical scales, below the slider.
''	Position (<code>x, y</code>) is not on any of the above parts.

.set(value)

Sets the scale's value.

22. The Scrollbar widget

A number of widgets, such as listboxes and canvases, can act like sliding windows into a larger virtual area. You can connect scrollbar widgets to them to give the user a way to slide the view around relative to the contents. Here's a screen shot of an entry widget with an associated scrollbar widget:



- Scrollbars can be horizontal, like the one shown above, or vertical. A widget that has two scrollable dimensions, such as a canvas or listbox, can have both a horizontal and a vertical scrollbar.
- The *slider*, or *scroll thumb*, is the raised-looking rectangle that shows the current scroll position.
- The two triangular *arrowheads* at each end are used for moving the position by small steps. The one on the left or top is called `arrow1`, and the one on the right or bottom is called `arrow2`.
- The *trough* is the sunken-looking area visible behind the arrowheads and slider. The trough is divided into two areas named `trough1` (above or to the left of the slider) and `trough2` (below or to the right of the slider).
- The slider's size and position, relative to the length of the entire widget, show the size and position of the view relative to its total size. For example, if a vertical scrollbar is associated with a listbox, and its slider extends from 50% to 75% of the height of the scrollbar, that means that the visible part of the listbox shows that portion of the overall list starting at the halfway mark and ending at the three-quarter mark.
- In a horizontal scrollbar, clicking B1 (button 1) on the left arrowhead moves the view by a small amount to the left. Clicking B1 on the right arrowhead moves the view by that amount to the right. For a vertical scrollbar, clicking the upward- and downward-pointing arrowheads moves the view

small amounts up or down. Refer to the discussion of the associated widget to find out the exact amount that these actions move the view.

- The user can drag the slider with B1 or B2 (the middle button) to move the view.
- For a horizontal scrollbar, clicking B1 in the trough to the left of the slider moves the view left by a page, and clicking B1 in the trough to the right of the slider moves the view a page to the right. For a vertical scrollbar, the corresponding actions move the view a page up or down.
- Clicking B2 anywhere along the trough moves the slider so that its left or top end is at the mouse, or as close to it as possible.

The *normalized position* of the scrollbar refers to a number in the closed interval [0.0, 1.0] that defines the slider's position. For vertical scrollbars, position 0.0 is at the top and 1.0 at the bottom; for horizontal scrollbars, position 0.0 is at the left end and 1.0 at the right.

To create a new **Scrollbar** widget as the child of a root window or frame *parent*:

```
w = tk.Scrollbar(parent, option, ...)
```

The constructor returns the new **Scrollbar** widget. Options for scrollbars include:

Table 31. Scrollbar widget options

activebackground	The color of the slider and arrowheads when the mouse is over them. See Section 5.3, "Colors" (p. 10).
activerelief	By default, the slider is shown with the <code>tk.RAISED</code> relief style. To display the slider with a different relief style when the mouse is over the slider.
bg or background	The color of the slider and arrowheads when the mouse is not over them.
bd or borderwidth	The width of the 3-d borders around the entire perimeter of the trough, and also the width of the 3-d effects on the arrowheads and slider. Default is no border around the trough, and a two-pixel border around the arrowheads and slider. For possible values, see Section 5.1, "Dimensions" (p. 9).
command	A procedure to be called whenever the scrollbar is moved. For a discussion of the calling sequence, see Section 22.1, "The Scrollbar <i>command</i> callback" (p. 77).
cursor	The cursor that appears when the mouse is over the scrollbar. See Section 5.8, "Cursors" (p. 13).
elementborderwidth	The width of the borders around the arrowheads and slider. The default is <code>elementborderwidth=-1</code> , which means to use the value of the <code>borderwidth</code> option.
highlightbackground	The color of the focus highlight when the scrollbar does not have focus. See Section 53, "Focus: routing keyboard input" (p. 155).
highlightcolor	The color of the focus highlight when the scrollbar has the focus.
highlightthickness	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
jump	This option controls what happens when a user drags the slider. Normally (<code>jump=0</code>), every small drag of the slider causes the <code>command</code> callback to be called. If you set this option to 1, the callback isn't called until the user releases the mouse button.
orient	Set <code>orient=tk.HORIZONTAL</code> for a horizontal scrollbar, <code>orient=tk.VERTICAL</code> for a vertical one (the default orientation).

<code>relief</code>	Controls the relief style of the widget; the default style is <code>tk.SUNKEN</code> . This option has no effect in Windows.
<code>repeatdelay</code>	This option controls how long button 1 has to be held down in the trough before the slider starts moving in that direction repeatedly. Default is <code>repeatdelay=300</code> , and the units are milliseconds.
<code>repeatinterval</code>	This option controls how often slider movement will repeat when button 1 is held down in the trough. Default is <code>repeatinterval=100</code> , and the units are milliseconds.
<code>takefocus</code>	Normally, you can tab the focus through a scrollbar widget; see Section 53, “Focus: routing keyboard input” (p. 155). Set <code>takefocus=0</code> if you don’t want this behavior. The default key bindings for scrollbars allow the user to use the \leftarrow and \rightarrow arrow keys to move horizontal scrollbars, and they can use the \uparrow and \downarrow keys to move vertical scrollbars.
<code>troughcolor</code>	The color of the trough.
<code>width</code>	Width of the scrollbar (its y dimension if horizontal, and its x dimension if vertical). Default is 16. For possible values, see Section 5.1, “Dimensions” (p. 9).

Methods on scrollbar objects include:

.activate(element=None)

If no argument is provided, this method returns one of the strings '`arrow1`', '`arrow2`', '`slider`', or '`'`', depending on where the mouse is. For example, the method returns '`slider`' if the mouse is on the slider. The empty string is returned if the mouse is not currently on any of these three controls.

To highlight one of the controls (using its `active relief` relief style and its `activebackground` color), call this method and pass a string identifying the control you want to highlight, one of '`arrow1`', '`arrow2`', or '`slider`'.

.delta(dx, dy)

Given a mouse movement of (`dx, dy`) in pixels, this method returns the `float` value that should be added to the current slider position to achieve that same movement. The value must be in the closed interval [-1.0, 1.0].

.fraction(x, y)

Given a pixel location (`x, y`), this method returns the corresponding normalized slider position in the interval [0.0, 1.0] that is closest to that location.

.get()

Returns two numbers (`a, b`) describing the current position of the slider. The `a` value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the `b` value gives the position of the right or bottom edge. Each value is in the interval [0.0, 1.0] where 0.0 is the leftmost or top position and 1.0 is the rightmost or bottom position. For example, if the slider extends from halfway to three-quarters of the way along the trough, you might get back the tuple (0.5, 0.75).

.identify(x, y)

This method returns a string indicating which (if any) of the components of the scrollbar are under the given (`x, y`) coordinates. The return value is one of '`arrow1`', '`trough1`', '`slider`', '`trough2`', '`arrow2`', or the empty string '`'`' if that location is not on any of the scrollbar components.

.set(*first*, *last*)

To connect a scrollbar to another widget *w*, set *w*'s `xscrollcommand` or `yscrollcommand` to the scrollbar's `.set` method. The arguments have the same meaning as the values returned by the `.get()` method. Please note that moving the scrollbar's slider does *not* move the corresponding widget.

22.1. The Scrollbar `command` callback

When the user manipulates a scrollbar, the scrollbar calls its `command` callback. The arguments to this call depend on what the user does:

- When the user requests a movement of one “unit” left or up, for example by clicking button B1 on the left or top arrowhead, the arguments to the callback look like:

```
command(tk.SCROLL, -1, tk.UNITS)
```

- When the user requests a movement of one unit right or down, the arguments are:

```
command(tk.SCROLL, 1, tk.UNITS)
```

- When the user requests a movement of one page left or up:

```
command(tk.SCROLL, -1, tk.PAGES)
```

- When the user requests a movement of one page right or down:

```
command(tk.SCROLL, 1, tk.PAGES)
```

- When the user drags the slider to a value *f* in the range [0,1], where 0 means all the way left or up and 1 means all the way right or down, the call is:

```
command(tk.MOVETO, f)
```

These calling sequences match the arguments expected by the `.xview()` and `.yview()` methods of canvases, listboxes, and text widgets. The `Entry` widget does not have an `.xview()` method. See Section 10.1, “Scrolling an `Entry` widget” (p. 45).

22.2. Connecting a Scrollbar to another widget

Here is a code fragment showing the creation of a canvas with horizontal and vertical scrollbars. In this fragment, `self` is assumed to be a `Frame` widget.

```
self.canv = tk.Canvas(self, width=600, height=400,
                      scrollregion=(0, 0, 1200, 800))
self.canv.grid(row=0, column=0)

self.scrollY = tk.Scrollbar(self, orient=tk.VERTICAL,
                           command=self.canv.yview)
self.scrollY.grid(row=0, column=1, sticky=tk.N+tk.S)

self.scrollX = tk.Scrollbar(self, orient=tk.HORIZONTAL,
                           command=self.canv.xview)
self.scrollX.grid(row=1, column=0, sticky=tk.E+tk.W)
```

```
self.canv['xscrollcommand'] = self.scrollX.set  
self.canv['yscrollcommand'] = self.scrollY.set
```

Notes:

- The connection goes both ways. The canvas's `xscrollcommand` option has to be connected to the horizontal scrollbar's `.set` method, and the scrollbar's `command` option has to be connected to the canvas's `.xview` method. The vertical scrollbar and canvas must have the same mutual connection.
- The `sticky` options on the `.grid()` method calls for the scrollbars force them to stretch just enough to fit the corresponding dimension of the canvas.

23. The Spinbox widget

The Spinbox widget allows the user to select values from a given set. The values may be a range of numbers, or a fixed set of strings.



On the screen, a Spinbox has an area for displaying the current values, and a pair of arrowheads.

- The user can click the upward-pointing arrowhead to advance the value to the next higher value in sequence. If the value is already at maximum, you can set up the widget, if you wish, so that the new value will wrap around to the lowest value.
- The user can click the downward-pointing arrowhead to advance the value to the next lower value in sequence. This arrow may also be configured to wrap around, so that if the current value is the lowest, clicking on the down-arrow will display the highest value.
- The user can also enter values directly, treating the widget as if it were an `Entry`. The user can move the focus to the widget (see Section 53, “Focus: routing keyboard input” (p. 155)), either by clicking on it or by using `tab` or `shift-tab`, and then edit the displayed value.

To create a new Spinbox widget as the child of a root window or frame `parent`:

```
w = tk.Spinbox(parent, option, ...)
```

The constructor returns the new Spinbox widget. Options include:

Table 32. Spinbox widget options

<code>activebackground</code>	Background color when the cursor is over the widget; see Section 5.3, “Colors” (p. 10).
<code>bg</code> or <code>background</code>	Background color of the widget.
<code>bd</code> or <code>borderwidth</code>	Width of the border around the widget; see Section 5.1, “Dimensions” (p. 9). The default value is one pixel.
<code>buttonbackground</code>	The background color displayed on the arrowheads. The default is gray.

<code>buttoncursor</code>	The cursor to be displayed when the mouse is over the arrowheads; see Section 5.8, “Cursors” (p. 13).
<code>buttondownrelief</code>	The relief style for the downward-pointing arrowhead; see Section 5.6, “Relief styles” (p. 12). The default style is <code>tk.RAISED</code> .
<code>buttonup</code>	The relief style for the upward-pointing arrowhead; see Section 5.6, “Relief styles” (p. 12). The default style is <code>tk.RAISED</code> .
<code>command</code>	Use this option to specify a function or method to be called whenever the user clicks on one of the arrowheads. Note that the callback is <i>not</i> called when the user edits the value directly as if it were an <code>Entry</code> .
<code>cursor</code>	Selects the cursor that is displayed when the mouse is over the entry part of the widget; see Section 5.8, “Cursors” (p. 13).
<code>disabledbackground</code>	These options select the background and foreground colors displayed when the widget's <code>state</code> is <code>tk.DISABLED</code> .
<code>disabledforeground</code>	
<code>exportselection</code>	Normally, the text in the entry portion of a <code>Spinbox</code> can be cut and pasted. To prohibit this behavior, set the <code>exportselection</code> option to <code>True</code> .
<code>font</code>	Use this option to select a different typeface for the entry text; see Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	This option selects the color used to display the text in the entry part of the widget, and the color of the arrowheads.
<code>format</code>	Use this option to control the formatting of numeric values in combination with the <code>from_</code> and <code>to</code> options. For example, <code>format='%.10.4f'</code> would display the value as a ten-character field, with four digits after the decimal.
<code>from_</code>	Use this option in combination with the <code>to</code> option (described below) to constrain the values to a numeric range. For example, <code>from_=1</code> and <code>to=9</code> would allow only values between 1 and 9 inclusive. See also the <code>increment</code> option below.
<code>highlightbackground</code>	The color of the focus highlight when the <code>Spinbox</code> does not have focus. See Section 5.3, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	The color of the focus highlight when the <code>Spinbox</code> has the focus.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set to 0 to suppress display of the focus highlight.
<code>increment</code>	When you constrain the values with the <code>from_</code> and <code>to</code> options, you can use the <code>increment</code> option to specify how much the value increases or decreases when the user clicks on an arrowhead. For example, with options <code>from_=0.0</code> , <code>to=2.0</code> , and <code>increment=0.5</code> , the up-arrowhead will step through values 0.0, 0.5, 1.0, 1.5, and 2.0.
<code>insertbackground</code>	Selects the color of the insertion cursor displayed in the entry part of the widget.
<code>insertborderwidth</code>	This option controls the width of the border around the insertion cursor. Normally, the insertion cursor will have no border. If this option is set to a nonzero value, the insertion cursor will be displayed in the <code>tk.RAISED</code> relief style.
<code>insertofftime</code>	These two options control the blink cycle of the insertion cursor: the amount of time it spends off and on, respectively, in milliseconds. For example, with options <code>insertofftime=200</code> and <code>insertontime=400</code> , the cursor would blink off for 0.2 seconds and then on for 0.4 seconds.
<code>insertontime</code>	

<code>insertwidth</code>	Use this option to specify the width of the insertion cursor; for possible values, see Section 5.1, “Dimensions” (p. 9). The default width is two pixels.
<code>justify</code>	This option controls the position of the text in the entry part of the widget. Values may be <code>tk.LEFT</code> to left-justify the text; <code>tk.CENTER</code> to center it; or <code>RIGHT</code> to right-justify the text.
<code>readonlybackground</code>	This option specifies the background color that will be displayed when the widget’s <code>state</code> is ‘ <code>readonly</code> ’; see Section 5.3, “Colors” (p. 10).
<code>relief</code>	Use this option to select a relief style for the widget; see Section 5.6, “Relief styles” (p. 12). The default style is <code>tk.SUNKEN</code> .
<code>repeatdelay</code>	These options specify the auto-repeat behavior of mouse clicks on the arrowheads; values are in milliseconds. The <code>repeatdelay</code> value specifies how long the mouse button must be held down before it repeats, and <code>repeatinterval</code> specifies how often the function repeats. Default values are 400 and 100 milliseconds, respectively.
<code>selectbackground</code>	The background color to use displaying selected items.
<code>selectborderwidth</code>	The width of the border to display around selected items.
<code>selectforeground</code>	The foreground color to use displaying selected items.
<code>state</code>	Normally, a <code>Spinbox</code> widget is created in the <code>tk.NORMAL</code> state. Set this option to <code>tk.DISABLED</code> to make the widget unresponsive to mouse or keyboard actions. If you set it to ‘ <code>readonly</code> ’, the value in the entry part of the widget cannot be modified with keystrokes, but the value can still be copied to the clipboard, and the widget still responds to clicks on the arrowheads.
<code>takefocus</code>	Normally, the entry part of a <code>Spinbox</code> widget can have focus (see Section 53, “Focus: routing keyboard input” (p. 155)). To remove the widget from the focus traversal sequence, set <code>takefocus=False</code> .
<code>textvariable</code>	If you want to retrieve the current value of the widget, you can use the <code>.get()</code> method below, or you can associate a control variable with the widget by passing that control variable as the value of this option. See Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>to</code>	This option specifies the upper limit of a range values. See the <code>from_</code> option, above, and also the <code>increment</code> option.
<code>values</code>	There are two ways to specify the possible values of the widget. One way is to provide a tuple of strings as the value of the <code>values</code> option. For example, <code>values=('red', 'blue', 'green')</code> would allow only those three strings as values. To configure the widget to accept a range of numeric values, see the <code>from_</code> option above.
<code>width</code>	Use this option to specify the number of characters allowed in the entry part of the widget. The default value is 20.
<code>wrap</code>	Normally, when the widget is at its highest value, the up-arrowhead does nothing, and when the widget is at its lowest value, the down-arrowhead does nothing. If you select <code>wrap=True</code> , the up-arrowhead will advance from the highest value back to the lowest, and the down-arrowhead will advance from the lowest value back to the highest.

<code>xscrollcommand</code>	Use this option to connect a scrollbar to the entry part of the widget. For details, see Section 22.2, “Connecting a Scrollbar to another widget” (p. 77).
-----------------------------	--

These methods are available on `Spinbox` widgets:

`.bbox(index)`

This method returns the bounding box of the character at position `index` in the entry part of the widget. The result is a tuple `(x, y, w, h)`, where the values are the `x` and `y` coordinates of the upper left corner, and the character's width and height in pixels, in that order.

`.delete(first, last=None)`

This method deletes characters from the entry part of the `Spinbox`. The values of `first` and `last` are interpreted in the standard way for Python slices.

`.get()`

This method returns the value of the `Spinbox`. The value is always returned as a string, even if the widget is set up to contain a number.

`.icursor(index)`

Use this method to position the insertion cursor at the location specified by `index`, using the standard Python convention for positions.

`.identify(x, y)`

Given a position `(x, y)` within the widget, this method returns a string describing what is at that location. Values may be any of:

- 'entry' for the entry area.
- 'buttonup' for the upward-pointing arrowhead.
- 'buttontdown' for the downward-pointing arrowhead.
- '' (an empty string) if these coordinates are not within the widget.

`.index(i)`

This method returns the numerical position of an index `i`. Arguments may be any of:

- `tk.END` to get the position after the last character of the entry.
- `tk.INSERT` to get the position of the insertion cursor.
- `tk.ANCHOR` to get the position of the selection anchor.
- `tk.SEL_FIRST` to get the position of the start of the selection. If the selection is not within the widget, this method raises a `tk.TclError` exception.
- `tk.SEL_LAST` to get the position just past the end of the selection. If the selection is not within the widget, this method raises a `tk.TclError` exception.
- A string of the form "@`x`" denotes an `x`-coordinate within the widget. The return value is the position of the character containing that coordinate. If the coordinate is outside the widget altogether, the return value will be the position of the character closest to that position.

`.insert(index, text)`

This method inserts characters from the string `text` at the position specified by `index`. For the possible index values, see the `.index()` method above.

`.invoke(element)`

Call this method to get the same effect as the user clicking on an arrowhead. The `element` argument is 'buttonup' for the up-arrowhead, and 'buttontdown' for the down-arrowhead.

`.scan_dragto(x)`

This method works the same as the `.scan_dragto()` method described in Section 10, “The `Entry` widget” (p. 41).

.scan_mark(*x*)

This method works the same as the `.scan_mark()` method described in Section 10, “The Entry widget” (p. 41).

.selection('from', *index*)

Sets the selection anchor in the widget to the position specified by the *index*. For the possible values of *index*, see the `.index()` method above. The initial value of the selection anchor is 0.

.selection('to', *index*)

Selects the text between the selection anchor and the given *index*.

.selection('range', *start*, *end*)

Select the text between the *start* and *end* indices. For allowable index values, see the `.index()` method above.

.selection_clear()

Clears the selection.

.selection_get()

Returns the selected text. If there is currently no selection, this method will raise a `tk.TclError` exception.

24. The Text widget

Text widgets are a much more generalized method for handling multiple lines of text than the Label widget. Text widgets are pretty much a complete text editor in a window:

- You can mix text with different fonts, colors, and backgrounds.
- You can intersperse embedded images with text. An image is treated as a single character. See Section 24.3, “Text widget images” (p. 86).
- An *index* is a way of describing a specific position between two characters of a text widget. See Section 24.1, “Text widget indices” (p. 84).
- A text widget may contain invisible *mark* objects between character positions. See Section 24.2, “Text widget marks” (p. 86).
- Text widgets allow you to define names for regions of the text called *tags*. You can change the appearance of a tagged region, changing its font, foreground and background colors, and other option. See Section 24.5, “Text widget tags” (p. 87).
- You can bind events to a tagged region. See Section 54, “Events” (p. 157).
- You can even embed a text widget in a “window” containing any Tkinter widget—even a frame widget containing other widgets. A window is also treated as a single character. See Section 24.4, “Text widget windows” (p. 87).

To create a text widget as the child of a root window or frame named *parent*:

```
w = tk.Text(parent, option, ...)
```

The constructor returns the new Text widget. Options include:

Table 33. Text widget options

autoseparators	If the <code>undo</code> option is set, the <code>autoseparators</code> option controls whether separators are automatically added to the undo stack after each insertion or deletion (if <code>autoseparators=True</code>) or not (if <code>autoseparat-</code>
-----------------------	---

	<code>ors=False)</code> . For an overview of the undo mechanism, see Section 24.7, “The <code>Text</code> widget undo/redo stack” (p. 88).
<code>bg</code> or <code>background</code>	The default background color of the text widget. See Section 5.3, “Colors” (p. 10).
<code>bd</code> or <code>borderwidth</code>	The width of the border around the text widget; see Section 5.1, “Dimensions” (p. 9). The default is two pixels.
<code>cursor</code>	The cursor that will appear when the mouse is over the text widget. See Section 5.8, “Cursors” (p. 13).
<code>exportselection</code>	Normally, text selected within a text widget is exported to be the selection in the window manager. Set <code>exportselection=0</code> if you don't want that behavior.
<code>font</code>	The default font for text inserted into the widget. Note that you can have multiple fonts in the widgets by using tags to change the properties of some text. See Section 5.4, “Type fonts” (p. 10).
<code>fg</code> or <code>foreground</code>	The color used for text (and bitmaps) within the widget. You can change the color for tagged regions; this option is just the default.
<code>height</code>	The height of the widget in <i>lines</i> (not pixels!), measured according to the current <code>font</code> size.
<code>highlightbackground</code>	The color of the focus highlight when the text widget does not have focus. See Section 5.3, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	The color of the focus highlight when the text widget has the focus.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set <code>highlightthickness=0</code> to suppress display of the focus highlight.
<code>insertbackground</code>	The color of the insertion cursor. Default is black.
<code>insertborderwidth</code>	Size of the 3-D border around the insertion cursor. Default is 0.
<code>insertofftime</code>	The number of milliseconds the insertion cursor is off during its blink cycle. Set this option to zero to suppress blinking. Default is 300.
<code>insertontime</code>	The number of milliseconds the insertion cursor is on during its blink cycle. Default is 600.
<code>insertwidth</code>	Width of the insertion cursor (its height is determined by the tallest item in its line). Default is 2 pixels.
<code>maxundo</code>	This option sets the maximum number of operations retained on the undo stack. For an overview of the undo mechanism, see Section 24.7, “The <code>Text</code> widget undo/redo stack” (p. 88). Set this option to -1 to specify an unlimited number of entries in the undo stack.
<code>padx</code>	The size of the internal padding added to the left and right of the text area. Default is one pixel. For possible values, see Section 5.1, “Dimensions” (p. 9).
<code>pady</code>	The size of the internal padding added above and below the text area. Default is one pixel.
<code>relief</code>	The 3-D appearance of the text widget. Default is <code>relief=tk.SUNKEN</code> ; for other values, see Section 5.6, “Relief styles” (p. 12).
<code>selectbackground</code>	The background color to use displaying selected text.
<code>selectborderwidth</code>	The width of the border to use around selected text.
<code>selectforeground</code>	The foreground color to use displaying selected text.

<code>spacing1</code>	This option specifies how much extra vertical space is put above each line of text. If a line wraps, this space is added only before the first line it occupies on the display. Default is <code>0</code> .
<code>spacing2</code>	This option specifies how much extra vertical space to add between displayed lines of text when a logical line wraps. Default is <code>0</code> .
<code>spacing3</code>	This option specifies how much extra vertical space is added below each line of text. If a line wraps, this space is added only after the last line it occupies on the display. Default is <code>0</code> .
<code>state</code>	Normally, text widgets respond to keyboard and mouse events; set <code>state=tk.NORMAL</code> to get this behavior. If you set <code>state=tk.DISABLED</code> , the text widget will not respond, and you won't be able to modify its contents programmatically either.
<code>tabs</code>	This option controls how tab characters position text. See Section 24.6, "Setting tabs in a <code>Text</code> widget" (p. 87).
<code>takefocus</code>	Normally, focus will visit a text widget (see Section 53, "Focus: routing keyboard input" (p. 155)). Set <code>takefocus=0</code> if you do not want focus in the widget.
<code>undo</code>	Set this option to <code>True</code> to enable the undo mechanism, or <code>False</code> to disable it. See Section 24.7, "The <code>Text</code> widget undo/redo stack" (p. 88).
<code>width</code>	The width of the widget in <i>characters</i> (not pixels!), measured according to the current font size.
<code>wrap</code>	<p>This option controls the display of lines that are too wide.</p> <ul style="list-style-type: none"> With the default behavior, <code>wrap=tk.CHAR</code>, any line that gets too long will be broken at any character. Set <code>wrap=tk.WORD</code> and it will break the line after the last word that will fit. If you want to be able to create lines that are too long to fit in the window, set <code>wrap=tk.NONE</code> and provide a horizontal scrollbar.
<code>xscrollcommand</code>	To make the text widget horizontally scrollable, set this option to the <code>.set</code> method of the horizontal scrollbar.
<code>yscrollcommand</code>	To make the text widget vertically scrollable, set this option to the <code>.set</code> method of the vertical scrollbar.

24.1. Text widget indices

An *index* is a general method of specifying a position in the content of a text widget. An index is a string with one of these forms:

`'line.column'`

The position just before the given *column* (counting from zero) on the given *line* (counting from one). Examples: '`1.0`' is the position of the beginning of the text; '`2.3`' is the position before the fourth character of the second line.

`'line.end'`

The position just before the newline at the end of the given line (counting from one). So, for example, index '`10.end`' is the position at the end of the tenth line.

tk.INSERT

The position of the insertion cursor in the text widget. This constant is equal to the string '`insert`'.

tk.CURRENT

The position of the character closest to the mouse pointer. This constant is equal to the string '`current`'.

tk.END

The position after the last character of the text. This constant is equal to the string '`end`'.

tk.SEL_FIRST

If some of the text in the widget is currently selection (as by dragging the mouse over it), this is the position before the start of the selection. If you try to use this index and nothing is selected, a `tk.TclError` exception will be raised. This constant is equal to the string '`sel.first`'.

tk.SEL_LAST

The position after the end of the selection, if any. As with `SEL_FIRST`, you'll get a `tk.TclError` exception if you use such an index and there is no selection. This constant is equal to the string '`sel.last`'.

'markname'

You can use a mark as an index; just pass its name where an index is expected. See Section 24.2, "Text widget marks" (p. 86).

'tag.first'

The position before the first character of the region tagged with name `tag`; see Section 24.5, "Text widget tags" (p. 87).

'tag.last'

The position after the last character of a tagged region.

'@x,y'

The position before the character closest to the coordinate (x, y) .

embedded-object

If you have an image or window embedded in the text widget, you can use the `PhotoImage`, `BitmapImage`, or embedded widget as an index. See Section 24.3, "Text widget images" (p. 86) and Section 24.4, "Text widget windows" (p. 87).

In addition to the basic index options above, you can build arbitrary complex expressions by adding any of these suffixes to a basic index or index expression:

+ n chars

From the given index, move forward n characters. This operation will cross line boundaries.

For example, suppose the first line looks like this:

```
abcdef
```

The index expression "`1.0 + 5 chars`" refers to the position between `e` and `f`. You can omit blanks and abbreviate keywords in these expressions if the result is unambiguous. This example could be abbreviated "`1.0+5c`".

- n chars

Similar to the previous form, but the position moves *backwards* n characters.

+ n lines

Moves n lines past the given index. *Tkinter* tries to leave the new position in the same column as it was on the line it left, but if the line at the new position is shorter, the new position will be at the end of the line.

- *n* lines

Moves *n* lines before the given index.

linestart

Moves to the position before the first character of the given index. For example, position “`current linestart`” refers to the beginning of the line closest to the mouse pointer.

lineend

Moves to the position after the last character of the given index. For example, position “`sel.last lineend`” refers to the end of the line containing the end of the current selection.

wordstart

The position before the beginning of the word containing the given index. For example, index “`11.44 wordstart`” refers to the position before the word containing position 44 on line 11.

For the purposes of this operation, a word is either a string of consecutive letter, digit, or underbar (`_`) characters, or a single character that is none of these types.

24.2. Text widget marks

A *mark* represents a floating position somewhere in the contents of a text widget.

- You handle each mark by giving it a name. This name can be any string that doesn't include whitespace or periods.
- There are two special marks. `tk.INSERT` is the current position of the insertion cursor, and `tk.CURRENT` is the position closest to the mouse cursor.
- Marks float along with the adjacent content. If you modify text somewhere away from a mark, the mark stays at the same position relative to its immediate neighbors.
- Marks have a property called *gravity* that controls what happens when you insert text at a mark. The default gravity is `tk.RIGHT`, which means that when new text is inserted at that mark, the mark stays after the end of the new text. If you set the gravity of a mark to `tk.LEFT` (using the text widget's `.mark_gravity()` method), the mark will stay at a position just before text newly inserted at that mark.
- Deleting the text all around a mark does not remove the mark. If you want to remove a mark, use the `.mark_unset()` method on the text widget.

Refer to Section 24.8, “Methods on Text widgets” (p. 88), below, to see how to use marks.

24.3. Text widget images

You can put an image or bitmap into a text widget. It is treated as a single character whose size is the natural size of the object. See Section 5.9, “Images” (p. 14) and Section 5.7, “Bitmaps” (p. 12).

Images are placed into the text widget by calling that widget's `.image_create()` method. See below for the calling sequence and other methods for image manipulation.

Images are manipulated by passing their name to methods on the text widget. You can give *Tkinter* a name for an image, or you can just let *Tkinter* generate a default name for that image.

An image may appear any number of times within the same Text widget. Each instance will carry a unique name. This names can be used as an index.

24.4. Text widget windows

You can put any *Tkinter* widget—even a frame containing other widgets—into a text widget. For example, you can put a fully functional button or a set of radiobuttons into a text widget.

Use the `.window_create()` method on the text widget to add the embedded widget. For the calling sequence and related methods, see Section 24.8, “Methods on Text widgets” (p. 88).

24.5. Text widget tags

There are lots of ways to change both the appearance and functionality of the items in a text widget. For text, you can change the font, size, and color. Also, you can make text, widgets, or embedded images respond to keyboard or mouse actions.

To control these appearance and functional features, you associate each feature with a *tag*. You can then associate a tag with any number of pieces of text in the widget.

- The name of a tag can be any string that does not contain white space or periods.
- There is one special predefined tag called SEL. This is the region currently selected, if any.
- Since any character may be part of more than one tag, there is a *tag stack* that orders all the tags. Entries are added at the end of the tag list, and later entries have priority over earlier entries.

So, for example, if there is a character *c* that is part of two tagged regions *t*₁ and *t*₂, and *t*₁ is deeper in the tag stack than *t*₂, and *t*₁ wants the text to be green and *t*₂ wants it to be blue, *c* will be rendered in blue because *t*₂ has precedence over *t*₁.

- You can change the ordering of tags in the tag stack.

Tags are created by using the `.tag_add()` method on the text widget. See Section 24.8, “Methods on Text widgets” (p. 88), below, for information on this and related methods.

24.6. Setting tabs in a Text widget

The `tabs` option for Text widgets gives you a number of ways to set tab stops within the widget.

- The default is to place tabs every eight characters.
- To set specific tab stops, set this option to a sequence of one or more distances. For example, setting `tabs= ('3c', '5c', '12c')` would put tab stops 3, 5, and 12cm from the left side. Past the last tab you set, tabs have the same width as the distance between the last two existing tab stops. So, continuing our example, because 12c - 5c is 7 cm, if the user keeps pressing the *Tab* key, the cursor would be positioned at 19cm, 26cm, 33cm, and so on.
- Normally, text after a tab character is aligned with its left edge on the tab stop, but you can include any of the keywords `tk.LEFT`, `tk.RIGHT`, `tk.CENTER`, or `tk.NUMERIC` in the list after a distance, and that will change the positioning of the text after each tab.
 - A `tk.LEFT` tab stop has the default behavior.
 - A `tk.RIGHT` tab stop will position the text so its right edge is on the stop.
 - A `tk.CENTER` tab will center the following text on the tab stop.
 - A `tk.NUMERIC` tab stop will place following text to the left of the stop up until the first period ('.') in the text—after that, the period will be centered on the stop, and the rest of the text will be positioned to its right.

For example, setting `tabs=('0.5i', '0.8i', tk.RIGHT, '1.2i', tk.CENTER, '2i', tk.NUMERIC)` would set four tab stops: a left-aligned tab stop half an inch from the left side, a right-aligned tab stop 0.8" from the left side, a center-aligned tab stop 1.2" from the left, and a numeric-aligned tab stop 2" from the left.

24.7. The Text widget undo/redo stack

The Text widget has a built-in mechanism that allows you to implement undo and redo operations that can cancel or reinstate changes to the text within the widget.

Here is how the undo/redo stack works:

- Every change to the content is recorded by pushing entries onto the stack that describe the change, whether an insertion or a deletion. These entries record the old state of the contents as well as the new state: if a deletion, the deleted text is recorded; if an insertion, the inserted text is recorded, along with a description of the location and whether it was an insertion or a deletion.
- Your program may also push a special record called a *separator* onto the stack.
- An *undo* operation changes the contents of the widget to what they were at some previous point. It does this by reversing all the changes pushed onto the undo/redo stack until it reaches a separator or until it runs out of stack.

However, note that Tkinter also remembers how much of the stack was reversed in the undo operation, until some other editing operation changes the contents of the widget.

- A *redo* operation works only if no editing operation has occurred since the last undo operation. It re-applies all the undone operations.

For the methods used to implement the undo/redo stack, see the `.edit_redo`, `.edit_reset`, `.edit_separator`, and `.edit_undo` methods in Section 24.8, “Methods on Text widgets” (p. 88). The undo mechanism is not enabled by default; you must set the `undo` option in the widget.

24.8. Methods on Text widgets

These methods are available on all text widgets:

`.bbox(index)`

Returns the bounding box for the character at the given index, a 4-tuple (`x`, `y`, `width`, `height`). If the character is not visible, returns `None`. Note that this method may not return an accurate value unless you call the `.update_idletasks()` method (see Section 26, “Universal widget methods” (p. 97)).

`.compare(index1, op, index2)`

Compares the positions of two indices in the text widget, and returns true if the relational `op` holds between `index1` and `index2`. The `op` specifies what comparison to use, one of: '`<`', '`<=`', '`==`', '`!=`', '`>=`', or '`>`'.

For example, for a text widget `t`, `t.compare('2.0', '<=', END)` returns true if the beginning of the second line is before or at the end of the text in `t`.

`.delete(index1, index2=None)`

Deletes text starting just after `index1`. If the second argument is omitted, only one character is deleted. If a second index is given, deletion proceeds up to, but not including, the character after `index2`. Recall that indices sit *between* characters.

.dlineinfo(index)

Returns a bounding box for the line that contains the given *index*. For the form of the bounding box, and a caution about updating idle tasks, see the definition of the `.bbox` method above.

.edit_modified(arg=None)

Queries, sets, or clears the *modified flag*. This flag is used to track whether the contents of the widget have been changed. For example, if you are implementing a text editor in a `Text` widget, you might use the modified flag to determine whether the contents have changed since you last saved the contents to a file.

When called with no argument, this method returns `True` if the modified flag has been set, `False` if it is clear. You can also explicitly set the modified flag by passing a `True` value to this method, or clear it by passing a `False` value.

Any operation that inserts or deletes text, whether by program actions or user actions, or an undo or redo operation, will set the modified flag.

.edit_redo()

Performs a redo operation. For details, see Section 24.7, “The `Text` widget undo/redo stack” (p. 88).

.edit_reset()

Clears the undo stack.

.edit_separator()

Pushes a separator onto the undo stack. This separator limits the scope of a future undo operation to include only the changes pushed since the separator was pushed. For details, see Section 24.7, “The `Text` widget undo/redo stack” (p. 88).

.edit_undo()

Reverses all changes to the widget's contents made since the last separator was pushed on the undo stack, or all the way to the bottom of the stack if the stack contains no separators. For details, see Section 24.7, “The `Text` widget undo/redo stack” (p. 88). It is an error if the undo stack is empty.

.image_create(index[, option=value, ...])

This method inserts an image into the widget. The image is treated as just another character, whose size is the image's natural size.

The options for this method are shown in the table below. You may pass either a series of `option=value` arguments, or a dictionary of option names and values.

<code>align</code>	This option specifies how the image is to be aligned vertically if its height is less than the height of its containing line. Values may be <code>top</code> to align it at the top of its space; <code>center</code> to center it; <code>bottom</code> to place it at the bottom; or <code>baseline</code> to line up the bottom of the image with the text baseline.
<code>image</code>	The image to be used. See Section 5.9, “Images” (p. 14).
<code>name</code>	You can assign a name to this instance of the image. If you omit this option, <code>Tkinter</code> will generate a unique name. If you create multiple instances of an image in the same <code>Text</code> widget, <code>Tkinter</code> will generate a unique name by appending a “#” followed by a number.
<code>padx</code>	If supplied, this option is a number of pixels of extra space to be added on both sides of the image.
<code>pady</code>	If supplied, this option is a number of pixels of extra space to be added above and below the image.

.get(index1, index2=None)

Use this method to retrieve the current text from the widget. Retrieval starts at index *index1*. If the second argument is omitted, you get the character after *index1*. If you provide a second index,

you get the text between those two indices. Embedded images and windows (widgets) are ignored. If the range includes multiple lines, they are separated by newline ('\n') characters.

.image_cget(index, option)

To retrieve the current value of an option set on an embedded image, call this method with an index pointing to the image and the name of the option.

.image_configure(index, option, ...)

To set one or more options on an embedded image, call this method with an index pointing to the image as the first argument, and one or more *option=value* pairs.

If you specify no options, you will get back a dictionary defining all the options on the image, and the corresponding values.

.image_names()

This method returns a tuple of the names of all the text widget's embedded images.

.index(i)

For an index *i*, this method returns the equivalent position in the form '*line.char*'.

.insert(index, text, tags=None)

Inserts the given *text* at the given *index*.

If you omit the **tags** argument, the newly inserted text will be tagged with any tags that apply to the characters *both* before and after the insertion point.

If you want to apply one or more tags to the text you are inserting, provide as a third argument a tuple of tag strings. Any tags that apply to existing characters around the insertion point are ignored. Note: The third argument must be a tuple. If you supply a list argument, Tkinter will silently fail to apply the tags. If you supply a string, each character will be treated as a tag.

.mark_gravity(mark, gravity=None)

Changes or queries the gravity of an existing mark; see Section 24.2, "Text widget marks" (p. 86), above, for an explanation of gravity.

To set the gravity, pass in the name of the mark, followed by either `tk.LEFT` or `tk.RIGHT`. To find the gravity of an existing mark, omit the second argument and the method returns `tk.LEFT` or `tk.RIGHT`.

.mark_names()

Returns a sequence of the names of all the marks in the window, including `tk.INSERT` and `tk.CURRENT`.

.mark_next(index)

Returns the name of the mark following the given *index*; if there are no following marks, the method returns an empty string.

If the *index* is in numeric form, the method returns the first mark at that position. If the *index* is a mark, the method returns the next mark following that mark, which may be at the same numerical position.

.mark_previous(index)

Returns the name of the mark preceding the given *index*. If there are no preceding marks, the method returns an empty string.

If the *index* is in numeric form, the method returns returns the last mark at that position. If the *index* is a mark, the method returns the preceding mark, which may be at the same numerical position.

.mark_set(*mark*, *index*)

If no mark with name *mark* exists, one is created with `tk.RIGHT` gravity and placed where *index* points. If the mark already exists, it is moved to the new location.

This method may change the position of the `tk.INSERT` or `tk.CURRENT` indices.

.mark_unset(*mark*)

Removes the named mark. This method cannot be used to remove the `tk.INSERT` or `tk.CURRENT` marks.

.scan_dragto(*x*, *y*)

See `.scan_mark`, below.

.scan_mark(*x*, *y*)

This method is used to implement fast scrolling of a `Text` widget. Typically, a user presses and holds a mouse button at some position in the widget, and then moves the mouse in the desired direction, and the widget moves in that direction at a rate proportional to the distance the mouse has moved since the button was depressed. The motion may be any combination of vertical or horizontal scrolling.

To implement this feature, bind a mouse button down event to a handler that calls `.scan_mark(x, y)`, where *x* and *y* are the current mouse position. Then bind the `<Motion>` event to a handler that calls `.scan_dragto(x, y)`, where *x* and *y* are the new mouse position.

.search(*pattern*, *index*, *option*, ...)

Searches for *pattern* (which can be either a string or a regular expression) in the buffer starting at the given *index*. If it succeeds, it returns an index of the '`line.char`' form; if it fails, it returns an empty string.

The allowable options for this method are:

backwards	Set this option to <code>True</code> to search backwards from the index. Default is forwards.
count	If you set this option to an <code>IntVar</code> control variable, when there is a match you can retrieve the length of the text that matched by using the <code>.get()</code> method on that variable after the method returns.
exact	Set this option to <code>True</code> to search for text that exactly matches the <i>pattern</i> . This is the default option. Compare the <code>regexp</code> option below.
forwards	Set this option to <code>True</code> to search forwards from the index. This is the default option.
regexp	Set this option to <code>True</code> to interpret the <i>pattern</i> as a Tcl-style regular expression. The default is to look for an exact match to <i>pattern</i> . Tcl regular expressions are a subset of Python regular expressions, supporting these features: <code>.</code> <code>^</code> <code>[c1...]</code> <code>(...)</code> <code>*</code> <code>+</code> <code>?</code> <code>e1 e2</code>
nocase	Set this option to <code>1</code> to ignore case. The default is a case-sensitive search.
stopindex	To limit the search, set this option to the index beyond which the search should not go.

.see(*index*)

If the text containing the given index is not visible, scroll the text until that text is visible.

.tag_add(*tagName*, *index1*, *index2=None*)

This method associates the tag named *tagName* with a region of the contents starting just after index *index1* and extending up to index *index2*. If you omit *index2*, only the character after *index1* is tagged.

.tag_bind(*tagName*, *sequence*, *func*, *add=None*)

This method binds an event to all the text tagged with *tagName*. See Section 54, “Events” (p. 157), below, for more information on event bindings.

To create a new binding for tagged text, use the first three arguments: *sequence* identifies the event, and *func* is the function you want it to call when that event happens.

To add another binding to an existing tag, pass the same first three arguments and '+' as the fourth argument.

To find out what bindings exist for a given sequence on a tag, pass only the first two arguments; the method returns the associated function.

To find all the bindings for a given tag, pass only the first argument; the method returns a list of all the tag's *sequence* arguments.

.tag_cget(*tagName*, *option*)

Use this method to retrieve the value of the given *option* for the given *tagName*.

.tag_config(*tagName*, *option*, ...)

To change the value of options for the tag named *tagName*, pass in one or more *option=value* pairs.

If you pass only one argument, you will get back a dictionary defining all the options and their values currently in force for the named tag.

Here are the options for tag configuration:

background	The background color for text with this tag. Note that you can't use bg as an abbreviation.
bgstipple	To make the background appear grayish, set this option to one of the standard bitmap names (see Section 5.7, “Bitmaps” (p. 12)). This has no effect unless you also specify a background .
borderwidth	Width of the border around text with this tag. Default is 0 . Note that you can't use bd as an abbreviation.
fgstipple	To make the text appear grayish, set this option a bitmap name.
font	The font used to display text with this tag. See Section 5.4, “Type fonts” (p. 10).
foreground	The color used for text with this tag. Note that you can't use the fg abbreviation here.
justify	The justify option set on the first character of each line determines how that line is justified: tk.LEFT (the default), tk.CENTER , or tk.RIGHT .
lmargin1	How much to indent the first line of a chunk of text that has this tag. The default is 0 . See Section 5.1, “Dimensions” (p. 9) for allowable values.
lmargin2	How much to indent successive lines of a chunk of text that has this tag. The default is 0 .
offset	How much to raise (positive values) or lower (negative values) text with this tag relative to the baseline. Use this to get superscripts or subscripts, for example. For allowable values, see Section 5.1, “Dimensions” (p. 9).
overstrike	Set overstrike=1 to draw a horizontal line through the center of text with this tag.
relief	Which 3-D effect to use for text with this tag. The default is relief=tk.FLAT ; for other possible values see Section 5.6, “Relief styles” (p. 12).

<code>rmargin</code>	Size of the right margin for chunks of text with this tag. Default is <code>0</code> .
<code>spacing1</code>	This option specifies how much extra vertical space is put above each line of text with this tag. If a line wraps, this space is added only before the first line it occupies on the display. Default is <code>0</code> .
<code>spacing2</code>	This option specifies how much extra vertical space to add between displayed lines of text with this tag when a logical line wraps. Default is <code>0</code> .
<code>spacing3</code>	This option specifies how much extra vertical space is added below each line of text with this tag. If a line wraps, this space is added only after the last line it occupies on the display. Default is <code>0</code> .
<code>tabs</code>	How tabs are expanded on lines with this tag. See Section 24.6, “Setting tabs in a <code>Text</code> widget” (p. 87).
<code>underline</code>	Set <code>underline=1</code> to underline text with this tag.
<code>wrap</code>	How long lines are wrapped in text with this tag. See the description of the <code>wrap</code> option for text widgets, above.

.tag_delete(*tagName*, ...)

To delete one or more tags, pass their names to this method. Their options and bindings go away, and the tags are removed from all regions of text.

.tag_lower(*tagName*, *belowThis=None*)

Use this method to change the order of tags in the tag stack (see Section 24.5, “Text widget tags” (p. 87), above, for an explanation of the tag stack). If you pass two arguments, the tag with name `tagName` is moved to a position just below the tag with name `belowThis`. If you pass only one argument, that tag is moved to the bottom of the tag stack.

.tag_names(*index=None*)

If you pass an index argument, this method returns a sequence of all the tag names that are associated with the character after that index. If you pass no argument, you get a sequence of all the tag names defined in the text widget.

.tag_nextrange(*tagName*, *index1*, *index2=None*)

This method searches a given region for places where a tag named `tagName` starts. The region searched starts at index `index1` and ends at index `index2`. If the `index2` argument is omitted, the search goes all the way to the end of the text.

If there is a place in the given region where that tag starts, the method returns a sequence `[i0, i1]`, where `i0` is the index of the first tagged character and `i1` is the index of the position just after the last tagged character.

If no tag starts are found in the region, the method returns an empty string.

.tag_prevrange(*tagName*, *index1*, *index2=None*)

This method searches a given region for places where a tag named `tagName` starts. The region searched starts *before* index `index1` and ends at index `index2`. If the `index2` argument is omitted, the search goes all the way to the end of the text.

The return values are as in `.tag_nextrange()`.

.tag_raise(*tagName*, *aboveThis=None*)

Use this method to change the order of tags in the tag stack (see Section 24.5, “Text widget tags” (p. 87), above, for an explanation of the tag stack). If you pass two arguments, the tag with name `tagName` is moved to a position just above the tag with name `aboveThis`. If you pass only one argument, that tag is moved to the top of the tag stack.

.tag_ranges(*tagName*)

This method finds all the ranges of text in the widget that are tagged with name *tagName*, and returns a sequence $[s_0, e_0, s_1, e_1, \dots]$, where each s_i is the index just before the first character of the range and e_i is the index just after the last character of the range.

.tag_remove(*tagName*, *index1*, *index2=None*)

Removes the tag named *tagName* from all characters between *index1* and *index2*. If *index2* is omitted, the tag is removed from the single character after *index1*.

.tag_unbind(*tagName*, *sequence*, *funcid=None*)

Remove the event binding for the given *sequence* from the tag named *tagName*. If there are multiple handlers for this sequence and tag, you can remove only one handler by passing it as the third argument.

.window_cget(*index*, *option*)

Returns the value of the given *option* for the embedded widget at the given *index*.

.window_configure(*index*, *option*)

To change the value of options for embedded widget at the given *index*, pass in one or more *option=value* pairs.

If you pass only one argument, you will get back a dictionary defining all the options and their values currently in force for the given widget.

.window_create(*index*, *option*, ...)

This method creates a window where a widget can be embedded within a text widget. There are two ways to provide the embedded widget:

- a. you can use pass the widget to the `window` option in this method, or
- b. you can define a procedure that will create the widget and pass that procedure as a callback to the `create` option.

Options for `.window_create()` are:

<code>align</code>	Specifies how to position the embedded widget vertically in its line, if it isn't as tall as the text on the line. Values include: <code>align=tk.CENTER</code> (the default), which centers the widget vertically within the line; <code>align=tk.TOP</code> , which places the top of the image at the top of the line; <code>align=tk.BOTTOM</code> , which places the bottom of the image at the bottom of the line; and <code>align=tk.BASELINE</code> , which aligns the bottom of the image with the text baseline.
<code>create</code>	A procedure that will create the embedded widget on demand. This procedure takes no arguments and must create the widget as a child of the text widget and return the widget as its result.
<code>padx</code>	Extra space added to the left and right of the widget within the text line. Default is <code>0</code> .
<code>pady</code>	Extra space added above and below the widget within the text line. Default is <code>0</code> .
<code>stretch</code>	This option controls what happens when the line is higher than the embedded widget. Normally this option is <code>0</code> , meaning that the embedded widget is left at its natural size. If you set <code>stretch=1</code> , the widget is stretched vertically to fill the height of the line, and the <code>align</code> option is ignored.
<code>window</code>	The widget to be embedded. This widget must be a child of the text widget.

.window_names()

Returns a sequence containing the names of all embedded widgets.

.xview(*tk.MOVETO*, *fraction*)

This method scrolls the text widget horizontally, and is intended for binding to the command option of a related horizontal scrollbar.

This method can be called in two different ways. The first call positions the text at a value given by *fraction*, where 0.0 moves the text to its leftmost position and 1.0 to its rightmost position.

.xview(*tk.SCROLL*, *n*, *what*)

The second call moves the text left or right: the *what* argument specifies how much to move and can be either *tk.UNITS* or *tk.PAGES*, and *n* tells how many characters or pages to move the text to the right relative to its image (or left, if negative).

.xview_moveto(*fraction*)

This method scrolls the text in the same way as `.xview(tk.MOVETO, fraction)`.

.xview_scroll(*n*, *what*)

Same as `.xview(tk.SCROLL, n, what)`.

.yview(*tk.MOVETO*, *fraction*)

The vertical scrolling equivalent of `.xview(tk.MOVETO, ...)`.

.yview(*tk.SCROLL*, *n*, *what*)

The vertical scrolling equivalent of `.xview(tk.SCROLL, ...)`. When scrolling vertically by *tk.UNITS*, the units are lines.

.yview_moveto(*fraction*)

The vertical scrolling equivalent of `.xview_moveto()`.

.yview_scroll(*n*, *what*)

The vertical scrolling equivalent of `.xview_scroll()`.

25. Toplevel: Top-level window methods

A *top-level window* is a window that has an independent existence under the window manager. It is decorated with the window manager's decorations, and can be moved and resized independently. Your application can use any number of top-level windows.

For any widget *w*, you can get to its top-level widget using `w.winfo_toplevel()`.

To create a new top-level window:

```
w = tk.Toplevel(option, ...)
```

Options include:

Table 34. Toplevel window methods

bg or background	The background color of the window. See Section 5.3, “Colors” (p. 10).
bd or borderwidth	Border width in pixels; default is 0. For possible values, see Section 5.1, “Dimensions” (p. 9). See also the <code>relief</code> option, below.
class_	You can give a <code>Toplevel</code> window a “class” name. Such names are matched against the option database, so your application can pick up the user's configuration preferences (such as colors) by class name. For example, you might design a series of pop-ups called “screamers,” and set them all up with <code>class_='Screamer'</code> . Then you can put a line in your option database like this:

	*Screamer*background: red
	and then, if you use the <code>.option_readfile()</code> method to read your option database, all widgets with that class name will default to a red background. This option is named <code>class</code> because <code>class</code> is a reserved word in Python.
<code>cursor</code>	The cursor that appears when the mouse is in this window. See Section 5.8, “Cursors” (p. 13).
<code>height</code>	Window height; see Section 5.1, “Dimensions” (p. 9).
<code>highlightbackground</code>	The color of the focus highlight when the window does not have focus. See Section 53, “Focus: routing keyboard input” (p. 155).
<code>highlightcolor</code>	The color of the focus highlight when the window has the focus.
<code>highlightthickness</code>	The thickness of the focus highlight. Default is 1. Set <code>highlightthickness=0</code> to suppress display of the focus highlight.
<code>menu</code>	To provide this window with a top-level menubar, supply a <code>Menu</code> widget as the value of this option. Under Mac OS, this menu will appear at the top of the screen when the window is active. Under Windows or Unix, it will appear at the top of the application.
<code>padx</code>	Use this option to provide extra space on the left and right sides of the window. The value is a number of pixels.
<code>pady</code>	Use this option to provide extra space on the top and bottom sides of the window. The value is a number of pixels.
<code>relief</code>	Normally, a top-level window will have no 3-d borders around it. To get a shaded border, set the <code>bd</code> option larger than its default value of zero, and set the <code>relief</code> option to one of the constants discussed under Section 5.6, “Relief styles” (p. 12).
<code>takefocus</code>	Normally, a top-level window does not get focus. Use <code>takefocus=True</code> if you want it to be able to take focus; see Section 53, “Focus: routing keyboard input” (p. 155).
<code>width</code>	The desired width of the window; see Section 5.1, “Dimensions” (p. 9).

These methods are available for top-level windows:

.aspect(n_{min} , d_{min} , n_{max} , d_{max})

Constrain the root window's width:length ratio to the range [n_{min} / d_{min} , n_{max} / d_{max}].

.deiconify()

If this window is iconified, expand it.

.geometry(*newGeometry=None*)

Set the window geometry. For the form of the argument, see Section 5.10, “Geometry strings” (p. 15).

If the argument is omitted, the current geometry string is returned.

.iconify()

Iconify the window.

.lift(*aboveThis=None*)

To raise this window to the top of the stacking order in the window manager, call this method with no arguments. You can also raise it to a position in the stacking order just above another `Toplevel` window by passing that window as an argument.

.lower(belowThis=None)

If the argument is omitted, moves the window to the bottom of the stacking order in the window manager. You can also move the window to a position just under some other top-level window by passing that `Toplevel` widget as an argument.

.maxsize(width=None, height=None)

Set the maximum window size. If the arguments are omitted, returns the current (`width`, `height`).

.minsize(width=None, height=None)

Set the minimum window size. If the arguments are omitted, returns the current minima as a 2-tuple.

.overrideredirect(flag=None)

If called with a `True` argument, this method sets the override redirect flag, which removes all window manager decorations from the window, so that it cannot be moved, resized, iconified, or closed. If called with a `False` argument, window manager decorations are restored and the override redirect flag is cleared. If called with no argument, it returns the current state of the override redirect flag.

Be sure to call the `.update_idletasks()` method (see Section 26, “Universal widget methods” (p. 97)) before setting this flag. If you call it before entering the main loop, your window will be disabled before it ever appears.

This method may not work on some Unix and MacOS platforms.

.resizable(width=None, height=None)

If is true, allow horizontal resizing. If height is true, allow vertical resizing. If the arguments are omitted, returns the current size as a 2-tuple.

.state(newstate=None)

Returns the window's current state, one of:

- 'normal': Displayed normally.
- 'iconic': Iconified with the `.iconify()` method.
- 'withdrawn': Hidden; see the `.withdraw()` method below.

To change the window's state, pass one of the strings above as an argument to the method. For example, to iconify a `Toplevel` instance `T`, use “`T.state('iconify')`”.

.title(text=None)

Set the window title. If the argument is omitted, returns the current title.

.transient(parent=None)

Make this window a transient window for some `parent` window; the default parent window is this window's parent.

This method is useful for short-lived pop-up dialog windows. A transient window always appears in front of its parent. If the parent window is iconified, the transient is iconified as well.

.withdraw()

Hides the window. Restore it with `.deiconify()` or `.iconify()`.

26. Universal widget methods

The methods are defined below on all widgets. In the descriptions, `w` can be any widget of any type.

w.after(delay_ms, callback=None, *args)

Requests Tkinter to call function `callback` with arguments `args` after a delay of at least `delay_ms` milliseconds. There is no upper limit to how long it will actually take, but your callback won't be called sooner than you request, and it will be called only once.

This method returns an integer "after identifier" that can be passed to the `.after_cancel()` method if you want to cancel the callback.

If you do not pass a `callback` argument, this method waits `delay_ms` milliseconds, as in the `.sleep()` function of the standard Python `time` module.⁹

w.after_cancel(id)

Cancels a request for callback set up earlier `.after()`. The `id` argument is the result returned by the original `.after()` call.

w.after_idle(func, *args)

Requests that Tkinter call function `func` with arguments `args` next time the system is idle, that is, next time there are no events to be processed. The callback will be called only once. If you want your callback to be called again, you must call the `.after_idle` method again.

w.bell()

Makes a noise, usually a beep.

w.bind(sequence=None, func=None, add=None)

This method is used to attach an event binding to a widget. See Section 54, "Events" (p. 157) for the overview of event bindings.

The `sequence` argument describes what event we expect, and the `func` argument is a function to be called when that event happens to the widget. If there was already a binding for that event for this widget, normally the old callback is replaced with `func`, but you can preserve both callbacks by passing `add='+'`.

w.bind_all(sequence=None, func=None, add=None)

Like `.bind()`, but applies to all widgets in the entire application.

w.bind_class(className, sequence=None, func=None, add=None)

Like `.bind()`, but applies to all widgets named `className` (e.g., `'Button'`).

w.bindtags(tagList=None)

If you call this method, it will return the "binding tags" for the widget as a sequence of strings. A binding tag is the name of a window (starting with `'. '`) or the name of a class (e.g., `'Listbox'`).

You can change the order in which binding levels are called by passing as an argument the sequence of binding tags you want the widget to use.

See Section 54, "Events" (p. 157) for a discussion of binding levels and their relationship to tags.

w.cget(option)

Returns the current value of `option` as a string. You can also get the value of an option for widget `w` as `w[option]`.

w.clipboard_append(text)

Appends the given `text` string to the display's clipboard, where cut and pasted strings are stored for all that display's applications.

w.clipboard_clear()

Clears the display's clipboard (see `.clipboard_append()` above).

⁹ <http://docs.python.org/library/time.html>

w.column_configure()

See Section 4.2, “Other grid management methods” (p. 7).

w.config(option=value, ...)

Same as .configure().

w.configure(option=value, ...)

Set the values of one or more options. For the options whose names are Python reserved words (`class`, `from`, `in`), use a trailing underbar: `'class_'`, `'from_'`, `'in_'`.

You can also set the value of an option for widget `w` with the statement

```
w[option] = value
```

If you call the `.config()` method on a widget with no arguments, you'll get a dictionary of all the widget's current options. The keys are the option names (including aliases like `bd` for `borderwidth`). The value for each key is:

- for most entries, a five-tuple: (option name, option database key, option database class, default value, current value); or,
- for alias names (like `'fg'`), a two-tuple: (alias name, equivalent standard name).

w.destroy()

Calling `w.destroy()` on a widget `w` destroys `w` and all its children.

w.event_add(virtual, *sequences)

This method creates a virtual event whose name is given by the `virtual` string argument. Each additional argument describes one `sequence`, that is, the description of a physical event. When that event occurs, the new virtual event is triggered.

See Section 54, “Events” (p. 157) for a general description of virtual events.

w.event_delete(virtual, *sequences)

Deletes physical events from the virtual event whose name is given by the string `virtual`. If all the physical events are removed from a given virtual event, that virtual event won't happen anymore.

w.event_generate(sequence, **kw)

This method causes an event to trigger without any external stimulus. The handling of the event is the same as if it had been triggered by an external stimulus. The `sequence` argument describes the event to be triggered. You can set values for selected fields in the `Event` object by providing `keyword=value` arguments, where the `keyword` specifies the name of a field in the `Event` object.

See Section 54, “Events” (p. 157) for a full discussion of events.

w.event_info(virtual=None)

If you call this method without an argument, you'll get back a sequence of all the currently defined virtual event names.

To retrieve the physical events associated with a virtual event, pass this method the name of the virtual event and you will get back a sequence of the physical `sequence` names, or `None` if the given virtual event has never been defined.

w.focus_displayof()

Returns the name of the window that currently has input focus on the same display as the widget. If no such window has input focus, returns `None`.

See Section 53, “Focus: routing keyboard input” (p. 155) for a general description of input focus.

w.focus_force()

Force the input focus to the widget. This is impolite. It's better to wait for the window manager to give you the focus. See also `.grab_set_global()` below.

w.focus_get()

Returns the widget that has focus in this application, if any—otherwise returns `None`.

w.focus_lastfor()

This method retrieves the name of the widget that last had the input focus in the top-level window that contains `w`. If none of this top-level's widgets have ever had input focus, it returns the name of the top-level widget. If this application doesn't have the input focus, `.focus_lastfor()` will return the name of the widget that will get the focus next time it comes back to this application.

w.focus_set()

If `w`'s application has the input focus, the focus will jump to `w`. If `w`'s application doesn't have focus, Tk will remember to give it to `w` next the application gets focus.

w.grab_current()

If there is a grab in force for `w`'s display, return its identifier, otherwise return `None`. Refer to Section 54, "Events" (p. 157) for a discussion of grabs.

w.grab_release()

If `w` has a grab in force, release it.

w.grab_set()

Widget `w` grabs all events for `w`'s application. If there was another grab in force, it goes away. See Section 54, "Events" (p. 157) for a discussion of grabs.

w.grab_set_global()

Widget `w` grabs all events for the entire screen. This is considered impolite and should be used only in great need. Any other grab in force goes away. Try to use this awesome power only for the forces of good, and never for the forces of evil, okay?

w.grab_status()

If there is a local grab in force (set by `.grab_set()`), this method returns the string '`local`'. If there is a global grab in force (from `.grab_set_global()`), it returns '`global`'. If no grab is in force, it returns `None`.

w.grid_forget()

See Section 4.2, "Other grid management methods" (p. 7).

w.grid_propagate()

See Section 4.2, "Other grid management methods" (p. 7).

w.grid_remove()

See Section 4.2, "Other grid management methods" (p. 7).

w.image_names()

Returns the names of all the images in `w`'s application as a sequence of strings.

w.keys()

Returns the option names for the widget as a sequence of strings.

w.lift(aboveThis=None)

If the argument is `None`, the window containing `w` is moved to the top of the window stacking order. To move the window just above some `Toplevel` window `w`, pass `w` as an argument.

w.lower(belowThis=None)

If the argument is `None`, the window containing `w` is moved to the bottom of the window stacking order. To move the window just below some `Toplevel` window `w`, pass `w` as an argument.

w.mainloop()

This method must be called, generally after all the static widgets are created, to start processing events. You can leave the main loop with the `.quit()` method (below). You can also call this method inside an event handler to resume the main loop.

w.nametowidget(name)

This method returns the actual widget whose path name is `name`. See Section 5.11, “Window names” (p. 16). If the `name` is unknown, this method will raise `KeyError`.

w.option_add(pattern, value, priority=None)

This method adds default option values to the *Tkinter* option database. The `pattern` is a string that specifies a default `value` for options of one or more widgets. The `priority` values are one of:

20	For global default properties of widgets.
40	For default properties of specific applications.
60	For options that come from user files such as their <code>.Xdefaults</code> file.
80	For options that are set after the application starts up. This is the default priority level.

Higher-level priorities take precedence over lower-level ones. See Section 27, “Standardizing appearance” (p. 105) for an overview of the option database. The syntax of the `pattern` argument to `.option_add()` is the same as the `option-pattern` part of the resource specification line.

For example, to get the effect of this resource specification line:

```
*Button*font: times 24 bold
```

your application (`self` in this example) might include these lines:

```
self.bigFont = tkFont.Font(family='times', size=24,  
                           weight='bold')  
self.option_add('*Button*font', self.bigFont)
```

Any `Button` widgets created after executing these lines would default to bold Times 24 font (unless overridden by a `font` option to the `Button` constructor).

w.option_clear()

This method removes all options from the *Tkinter* option database. This has the effect of going back to all the default values.

w.option_get(name, classname)

Use this method to retrieve the current value of an option from the *Tkinter* option database. The first argument is the instance key and the second argument is the class key. If there are any matches, it returns the value of the option that best matches. If there are no matches, it returns ''.

Refer to Section 27, “Standardizing appearance” (p. 105) for more about how keys are matched with options.

w.option_readfile(fileName, priority=None)

As a convenience for user configuration, you can designate a named file where users can put their preferred options, using the same format as the `.Xdefaults` file. Then, when your application is initializing, you can pass that file's name to this method, and the options from that file will be added to the database. If the file doesn't exist, or its format is invalid, this method will raise `tk.TclError`.

Refer to Section 27, “Standardizing appearance” (p. 105) for an introduction to the options database and the format of option files.

w.register(*function*)

This method creates a Tcl wrapper around a Python *function*, and returns the Tcl wrapper name as a string. For an example of the usage of this method, see Section 10.2, “Adding validation to an `Entry` widget” (p. 45).

w.quit()

This method exits the main loop. See `.mainloop()`, above, for a discussion of main loops.

w.rowconfigure()

See Section 4.2, “Other grid management methods” (p. 7).

w.selection_clear()

If *w* currently has a selection (such as a highlighted segment of text in an entry widget), clear that selection.

w.selection_get()

If *w* currently has a selection, this method returns the selected text. If there is no selection, it raises `tk.TclError`.

w.selection_own()

Make *w* the owner of the selection in *w*'s display, stealing it from the previous owner, if any.

w.selection_own_get()

Returns the widget that currently owns the selection in *w*'s display. Raises `tk.TclError` if there is no such selection.

w.tk_focusFollowsMouse()

Normally, the input focus cycles through a sequence of widgets determined by their hierarchy and creation order; see Section 53, “Focus: routing keyboard input” (p. 155). You can, instead, tell *Tkinter* to force the focus to be wherever the mouse is; just call this method. There is no easy way to undo it, however.

w.tk_focusNext()

Returns the widget that follows *w* in the focus traversal sequence. Refer to Section 53, “Focus: routing keyboard input” (p. 155) for a discussion of focus traversal.

w.tk_focusPrev()

Returns the widget that precedes *w* in the focus traversal sequence.

w.unbind(*sequence*, *funcid=None*)

This method deletes bindings on *w* for the event described by *sequence*. If the second argument is a callback bound to that sequence, that callback is removed and the rest, if any, are left in place. If the second argument is omitted, all bindings are deleted.

See Section 54, “Events” (p. 157), below, for a general discussion of event bindings.

w.unbind_all(*sequence*)

Deletes all event bindings throughout the application for the event described by the given *sequence*.

w.unbind_class(*className*, *sequence*)

Like `.unbind()`, but applies to all widgets named *className* (e.g., `'Entry'` or `'Listbox'`).

w.update()

This method forces the updating of the display. It should be used only if you know what you're doing, since it can lead to unpredictable behavior or looping. It should never be called from an event callback or a function that is called from an event callback.

w.update_idletasks()

Some tasks in updating the display, such as resizing and redrawing widgets, are called *idle tasks* because they are usually deferred until the application has finished handling events and has gone back to the main loop to wait for new events.

If you want to force the display to be updated before the application next idles, call the `w.update_idletasks()` method on any widget.

w.wait_variable(v)

Waits until the value of variable `v` is set, even if the value does not change. This method enters a local wait loop, so it does not block the rest of the application.

w.wait_visibility(w)

Wait until widget `w` (typically a `Toplevel`) is visible.

w.wait_window(w)

Wait until window `w` is destroyed.

w.winfo_children()

Returns a list of all `w`'s children, in their stacking order from lowest (bottom) to highest (top).

w.winfo_class()

Returns `w`'s class name (e.g., '`Button`').

w.winfo_containing(rootX, rootY, displayof=0)

This method is used to find the window that contains point (`rootX`, `rootY`). If the `displayof` option is false, the coordinates are relative to the application's root window; if true, the coordinates are treated as relative to the top-level window that contains `w`. If the specified point is in one of the application's top-level windows, this method returns that window; otherwise it returns `None`.

w.winfo_depth()

Returns the number of bits per pixel in `w`'s display.

w.winfo_fpixels(number)

For any dimension `number` (see Section 5.1, "Dimensions" (p. 9)), this method returns that distance in pixels on `w`'s display, as a number of type `float`.

w.winfo_geometry()

Returns the geometry string describing the size and on-screen location of `w`. See Section 5.10, "Geometry strings" (p. 15).

Warning

The geometry is not accurate until the application has updated its idle tasks. In particular, all geometries are initially '`1x1+0+0`' until the widgets and geometry manager have negotiated their sizes and positions. See the `.update_idletasks()` method, above, in this section to see how to insure that the widget's geometry is up to date.

w.winfo_height()

Returns the current height of `w` in pixels. See the remarks on geometry updating under `.winfo_geometry()`, above. You may prefer to use `.winfo_reqheight()`, described below, which is always up to date.

w.winfo_id()

Returns an integer that uniquely identifies `w` within its top-level window. You will need this for the `.winfo_pathname()` method, below.

w.winfo_ismapped()

This method returns true if *w* is mapped, false otherwise. A widget is mapped if it has been gridded (or placed or packed, if you are using one of the other geometry managers) into its parent, and if its parent is mapped, and so on up to the top-level window.

w.winfo_manager()

If *w* has not been gridded (or placed via one of the other geometry managers), this method returns an empty string. If *w* has been gridded or otherwise placed, it returns a string naming the geometry manager for *w*: this value will be one of 'grid', 'pack', 'place', 'canvas', or 'text'.

w.winfo_name()

This method returns *w*'s name relative to its parent. See Section 5.11, "Window names" (p. 16). Also see `.winfo_pathname()`, below, to find out how to obtain a widget's path name.

w.winfo_parent()

Returns *w*'s parent's path name, or an empty string if *w* is a top-level window. See Section 5.11, "Window names" (p. 16) above, for more on widget path names.

w.winfo_pathname(*id*, *displayof=0*)

If the *displayof* argument is false, returns the window path name of the widget with unique identifier *id* in the application's main window. If *displayof* is true, the *id* number specifies a widget in the same top-level window as *w*. See Section 5.11, "Window names" (p. 16) for a discussion of widget path names.

w.winfo_pixels(*number*)

For any dimension *number* (see Dimensions, above), this method returns that distance in pixels on *w*'s display, as an integer.

w.winfo_pointerx()

Returns the same value as the *x* coordinate returned by `.winfo_pointerxy()`.

w.winfo_pointerxy()

Returns a tuple (*x*, *y*) containing the coordinates of the mouse pointer relative to *w*'s root window. If the mouse pointer isn't on the same screen, returns (-1, -1).

w.winfo_pointery()

Returns the same value as the *y* coordinate returned by `.winfo_pointerxy()`.

w.winfo_reqheight()

These methods return the requested height of widget *w*. This is the minimum height necessary so that all of *w*'s contents have the room they need. The actual height may be different due to negotiations with the geometry manager.

w.winfo_reqwidth()

Returns the requested width of widget *w*, the minimum width necessary to contain *w*. As with `.winfo_reqheight()`, the actual width may be different due to negotiations with the geometry manager.

w.winfo_rgb(*color*)

For any given color, this method returns the equivalent red-green-blue color specification as a 3-tuple (*r*, *g*, *b*), where each number is an integer in the range [0, 65536]. For example, if the *color* is 'green', this method returns the 3-tuple (0, 65535, 0).

For more on specifying colors, see Section 5.3, "Colors" (p. 10).

w.winfo_rootx()

Returns the *x* coordinates of the left-hand side of *w*'s root window relative to *w*'s parent.

If *w* has a border, this is the outer edge of the border.

w.winfo_rooty()

Returns the *y* coordinate of the top side of *w*'s root window relative to *w*'s parent.

If *w* has a border, this is the top edge of the border.

w.winfo_screenheight()

Returns the height of the screen in pixels.

w.winfo_screenmmheight()

Returns the height of the screen in millimeters.

w.winfo_screenmmwidth()

Returns the width of the screen in millimeters.

w.winfo_screenvisual()

Returns a string that describes the display's method of color rendition. This is usually '`truecolor`' for 16- or 24-bit displays, '`pseudocolor`' for 256-color displays.

w.winfo_screenwidth()

Returns the width of the screen in pixels.

w.winfo_toplevel()

Returns the top-level window containing *w*. That window supports all the methods on `Toplevel` widgets; see Section 25, "Toplevel: Top-level window methods" (p. 95).

w.winfo_viewable()

A predicate that returns a `True` value if *w* is viewable, that is, if it and all its ancestors in the same `Toplevel` are mapped.

w.winfo_width()

Returns the current width of *w* in pixels. See the remarks on geometry updating under `.winfo_geometry()`, above. You may prefer to use the `.winfo_reqwidth()` method, described above; it is always up to date.

w.winfo_x()

Returns the *x* coordinate of the left side of *w* relative to its parent. If *w* has a border, this is the outer edge of the border.

w.winfo_y()

Returns the *y* coordinate of the top side of *w* relative to its parent. If *w* has a border, this is the outer edge of the border.

27. Standardizing appearance and the option database

It's easy to apply colors, fonts, and other options to the widgets when you create them. However,

- if you want a lot of widgets to have the same background color or font, it's tedious to specify each option each time, and
- it's nice to let the user override your choices with their favorite color schemes, fonts, and other choices.

Accordingly, we use the idea of an *option database* to set up default option values.

- Your application can specify a file (such as the standard `.Xdefaults` file used by the X Window System) that contains the user's preferences. You can set up your application to read the file and tell `Tkinter` to use those defaults. See the section on the `.option_readfile()` method, above, in the section on Section 26, "Universal widget methods" (p. 97), for the structure of this file.

- Your application can directly specify defaults for one or many types of widgets by using the `.option_add()` method; see this method under Section 26, “Universal widget methods” (p. 97).

Before we discuss how options are set, consider the problem of customizing the appearance of GUIs in general. We could give every widget in the application a name, and then ask the user to specify every property of every name. But this is cumbersome, and would also make the application hard to reconfigure—if the designer adds new widgets, the user would have to describe every property of every new widget.

So, the option database allows the programmer and the user to specify *general patterns* describing which widgets to configure.

These patterns operate on the names of the widgets, but widgets are named using *two* parallel naming schemes:

- a. Every widget has a *class name*. By default, the class name is the same as the class constructor: '`Button`' for buttons, '`Frame`' for a frame, and so on. However, you can create new classes of widgets, usually inheriting from the `Frame` class, and give them new names of your own creation. See Section 27.1, “How to name a widget class” (p. 106) for details.
- b. You can also give any widget an *instance name*. The default name of a widget is usually a meaningless number (see Section 5.11, “Window names” (p. 16)). However, as with widget classes, you can assign a name to any widget. See the section Section 27.2, “How to name a widget instance” (p. 107) for details.

Every widget in every application therefore has two hierarchies of names—the class name hierarchy and the instance name hierarchy. For example, a button embedded in a text widget which is itself embedded in a frame would have the class hierarchy `Frame.Text.Button`. It might also have an instance hierarchy something like `.mainFrame.messageText.panicButton` if you so named all the instances. The initial dot stands for the root window; see Section 5.11, “Window names” (p. 16) for more information on window path names.

The option database mechanism can make use of either class names or instance names in defining options, so you can make options apply to whole classes (e.g., all buttons have a blue background) or to specific instances (e.g., the Panic Button has red letters on it). After we look at how to name classes and instances, in Section 27.3, “Resource specification lines” (p. 107), we’ll discuss how the options database really works.

27.1. How to name a widget class

For example, suppose that `Jukebox` is a new widget class that you have created. It’s probably best to have new widget classes inherit from the `Frame` class, so to *Tkinter* it acts like a frame, and you can arrange other widgets such as labels, entries, and buttons inside it.

You set the new widget’s class name by passing the name as the `class_` option to the parent constructor in your new class’s constructor. Here is a fragment of the code that defines the new class:

```
class Jukebox(tk.Frame):
    def __init__(self, master):
        '''Constructor for the Jukebox class
        ...
        tk.Frame.__init__(self, master, class_='Jukebox')
        self.__createWidgets()
        ...
```

27.2. How to name a widget instance

To give an instance name to a specific widget in your application, set that widget's `name` option to a string containing the name.

Here's an example of an instance name. Suppose you are creating several buttons in an application, and you want one of the buttons to have an instance name of `panicButton`. Your call to the constructor might look like this:

```
self.panic = tk.Button(self, name='panicButton', text='Panic', ...)
```

27.3. Resource specification lines

Each line in an option file specifies the value of one or more options in one or more applications and has one of these formats:

```
app option-pattern: value  
option-pattern: value
```

The first form sets options only when the name of the application matches `app`; the second form sets options for all applications.

For example, if your application is called `xparrot`, a line of the form

```
xparrot*background: LimeGreen
```

sets all `background` options in the `xparrot` application to lime green. (Use the `-name` option on the command line when launching your application to set the name to '`xparrot`'.)

The `option-pattern` part has this syntax:

```
{{*| .}name}...option
```

That is, each `option-pattern` is a list of zero or more names, each of which is preceded by an asterisk or period. The last name in the series is the name of the option you are setting. Each of the rest of the names can be either:

- the name of a widget *class* (capitalized), or
- the name of an *instance* (lowercased).

The way the option patterns work is a little complicated. Let's start with a simple example:

```
*font: times 24
```

This line says that all `font` options should default to 24-point Times. The `*` is called the *loose binding* symbol, and means that this option pattern applies to any `font` option anywhere in any application. Compare this example:

```
*Listbox.font: lucidatypewriter 14
```

The period between `Listbox` and `font` is called the *tight binding* symbol, and it means that this rule applies only to `font` options for widgets in class `Listbox`.

As another example, suppose your `xparrot` application has instances of widgets of class `Jukebox`. In order to set up a default background color for all widgets of that class `Jukebox`, you could put a line in your options file like this:

```
xparrot*Jukebox*background: PapayaWhip
```

The loose-binding (*) symbol between `Jukebox` and `background` makes this rule apply to any `background` option of any widget anywhere inside a `Jukebox`. Compare this option line:

```
xparrot*Jukebox.background: NavajoWhite
```

This rule will apply to the frame constituting the `Jukebox` widget itself, but because of the tight-binding symbol it will not apply to widgets that are inside the `Jukebox` widget.

In the next section we'll talk about how *Tkinter* figures out exactly which option value to use if there are multiple resource specification lines that apply.

27.4. Rules for resource matching

When you are creating a widget, and you don't specify a value for some option, and two or more resource specifications apply to that option, the most specific one applies.

For example, suppose your options file has these two lines:

```
*background: LimeGreen  
*Listbox*background: FloralWhite
```

Both specifications apply to the `background` option in a `Listbox` widget, but the second one is more specific, so it will win.

In general, the names in a resource specification are a sequence n_1, n_2, n_3, \dots, o where each n_i is a class or instance name. The class names are ordered from the highest to the lowest level, and o is the name of an option.

However, when *Tkinter* is creating a widget, all it has is the class name and the instance name of that widget.

Here are the precedence rules for resource specifications:

1. The name of the option must match the o part of the *option-pattern*. For example, if the rule is

```
xparrot*indicatoron: 0
```

this will match only options named `indicatoron`.

2. The tight-binding operator (.) is more specific than the loose-binding operator (*). For example, a line for `*Button.font` is more specific than a line for `*Button*font`.
3. References to instances are more specific than references to classes. For example, if you have a button whose instance name is `panicButton`, a rule for `*panicButton*font` is more specific than a rule for `*Button*font`.
4. A rule with more levels is more specific. For example, a rule for `*Button*font` is more specific than a rule for `*font`.
5. If two rules have same number of levels, names earlier in the list are more specific than later names. For example, a rule for `xparrot*font` is more specific than a rule for `*Button*font`.

28. `ttk`: Themed widgets

Starting with Tk 8.5, the `ttk` module became available. This module replaces much (but not all) of the original *Tkinter* machinery. Use this module to gain these advantages:

- *Platform-specific appearance.* In releases before Tk 8.5, one of the commonest complaints about Tk applications was that they did not conform to the style of the various platforms.

The `ttk` module allows you to write your application in a generic way, yet your application can look like a Windows application under Windows, like a MacOS app under MacOS, and so on, without any change to your program.

Each possible different appearance is represented by a named *ttk theme*. For example, the `classic` theme gives you the appearance of the original *Tkinter* widgets described in the previous sections.

- *Simplification and generalization of state-specific widget behavior.* In the basic *Tkinter* world, there are a lot of widget options that specify how the widget should look or behave depending on various conditions.

For example, the `tk.Button` widget has several different options that control the foreground (text) color.

- The `activeforeground` color option applies when the cursor is over the button.
- The `disabledforeground` color is used when the widget is disabled.
- The widget will have the `foreground` color when the other conditions don't apply.

The `ttk` module collapses a lot of these special cases into a simple two-part system:

- Every widget has a number of different states, and each state can be turned on or off independently of the others. Examples of states are: `disabled`, `active`, and `focus`.
- You can set up a *style map* that specifies that certain options will be set to certain values depending on some state or some combination of the widget's states.

To use `ttk`, you will need to know these things.

- Section 28.1, “Importing `ttk`” (p. 109): Setting up your program to use `ttk`.
- Section 28.2, “The `ttk` widget set” (p. 110): The new and replaced `ttk` widgets.
- Section 47, “Customizing and creating `ttk` themes and styles” (p. 146).

28.1. Importing `ttk`

There are different ways to import the `ttk` module.

- If you prefer that all the widgets and other features of *Tkinter* and `ttk` be in your global namespace, use this form of import:

```
from Tkinter import *
from ttk import *
```

It is important to do these two imports in this order, so that all the widget types from `ttk` replace the equivalent widgets from *Tkinter*. For example, all your `Button` widgets will come from `ttk` and not *Tkinter*.

- In more complex applications, where you are using more than one imported module, it can greatly improve the readability of your code if you practice *safe namespace hygiene*: import all your modules using the “`import modulename`” syntax. This requires just a bit more typing, but it has the great advantage that you can look at a reference to something and tell where it came from.

We recommend this form of import:

```
import ttk
```

So after this import, `ttk.Label` is the `Label` widget constructor, `ttk.Button` is a `Button`, and so on.

If you need to refer to items from the *Tkinter* module, it is available as `ttk.Tkinter`. For example, the anchor code for “northeast” is `ttk.Tkinter.NE`.

You may instead import *Tkinter* separately in this way:

```
import Tkinter as tk
```

After this form of import, the code for “northeast” is `tk.NE`.

28.2. The `ttk` widget set

The `ttk` module contains different versions of most of the standard *Tkinter* widgets and a few new ones. These widgets replace the ones from *Tkinter* of the same name:

- Section 29, “`ttk.Button`” (p. 110).
- Section 30, “`ttk.Checkbutton`” (p. 112).
- Section 32, “`ttk.Entry`” (p. 116).
- Section 33, “`ttk.Frame`” (p. 118).
- Section 34, “`ttk.Label`” (p. 119).
- Section 35, “`ttk.LabelFrame`” (p. 122).
- Section 36, “`ttk.MenuButton`” (p. 124).
- Section 38, “`ttk.PanedWindow`” (p. 129).
- Section 40, “`ttk.Radiobutton`” (p. 131).
- Section 41, “`ttk.Scale`” (p. 133).
- Section 42, “`ttk.Scrollbar`” (p. 135).

These widgets are new, and specific to `ttk`:

- Section 31, “`ttk.Combobox`” (p. 115).
- Section 37, “`ttk.Notebook`” (p. 126).
- Section 39, “`ttk.Progressbar`” (p. 130).
- Section 43, “`ttk.Separator`” (p. 137).

29. `ttk.Button`

This widget is the `ttk` version of Section 7, “The `Button` widget” (p. 18). To create a `ttk.Button` widget:

```
w = ttk.Button(parent, option=value, ...)
```

Here are the options for the `ttk.Button` widget. Compare these to the *Tkinter* version discussed in Section 7, “The `Button` widget” (p. 18).

Table 35. `ttk.Button` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>command</code>	A function to be called when the button is pressed.
<code>compound</code>	If you provide both <code>image</code> and <code>text</code> options, the <code>compound</code> option specifies the position of the image relative to the text. The value may be <code>tk.TOP</code> (image above text), <code>tk.BOTTOM</code> (image below text), <code>tk.LEFT</code> (image to the left of the text), or <code>tk.RIGHT</code> (image to the right of the text).

	When you provide both <code>image</code> and <code>text</code> options but don't specify a <code>compound</code> option, the image will appear and the text will not.
<code>cursor</code>	The cursor that will appear when the mouse is over the button; see Section 5.8, "Cursors" (p. 13).
<code>image</code>	An image to appear on the button; see Section 5.9, "Images" (p. 14).
<code>style</code>	The style to be used in rendering this button; see Section 49, "Using and customizing <code>ttk</code> styles" (p. 147).
<code>takefocus</code>	By default, a <code>ttk.Button</code> will be included in focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
<code>text</code>	The text to appear on the button, as a string.
<code>textvariable</code>	A variable that controls the text that appears on the button; see Section 52, "Control variables: the values behind the widgets" (p. 153).
<code>underline</code>	If this option has a nonnegative value n , an underline will appear under the character at position n .
<code>width</code>	If the label is text, this option specifies the absolute width of the text area on the button, as a number of characters; the actual width is that number multiplied by the average width of a character in the current font. For image labels, this option is ignored. The option may also be configured in a style.

These options of the `Tkinter Button` widget are *not* supported by the `ttk.Button` constructor:

Table 36. Tkinter Button options not in `ttk.Button`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, "ttk style maps: dynamic appearance changes" (p. 151).
<code>activeforeground</code>	Use a style map to control the <code>foreground</code> option.
<code>anchor</code>	Configure this option using a style; see Section 49, "Using and customizing <code>ttk</code> styles" (p. 147). Use this option to specify the position of the text when the <code>width</code> option allocates extra horizontal space. For example, if you specify options <code>width=20</code> and <code>compound=tk.RIGHT</code> on a button that displays both text and an image, and a style that specifies <code>anchor=tk.E</code> (east), the image will be at the right-hand end of the twenty-character space, with the text just to its left. When the button displays an image but no text, this option is ignored.
<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style. The <code>bg</code> abbreviation is not supported.
<code>bitmap</code>	Not supported.
<code>borderwidth</code> or <code>bd</code>	Configure the <code>borderwidth</code> option using a style. The <code>bd</code> abbreviation is not supported.
<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, "Cursors" (p. 13).
<code>default</code>	Not supported; see Section 50.2, "ttk style maps: dynamic appearance changes" (p. 151).
<code>disabledforeground</code>	Use a style map for the <code>foreground</code> option; see Section 50.2, "ttk style maps: dynamic appearance changes" (p. 151).

<code>font</code>	Configure this option using a style.
<code>foreground</code> or <code>fg</code>	Configure this option using a style.
<code>height</code>	Not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the button does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.
<code>justify</code>	If the <code>text</code> contains newline ('\n') characters, the text will occupy multiple lines on the button. The <code>justify</code> option controls how each line is positioned horizontally. Configure this option using a style; values may be <code>tk.LEFT</code> , <code>tk.CENTER</code> , or <code>tk.RIGHT</code> for lines that are left-aligned, centered, or right-aligned, respectively.
<code>overrelief</code>	Use a style map to control the <code>relief</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.
<code>relief</code>	Configure this option using a style; see Section 49, “Using and customizing <i>ttk</i> styles” (p. 147).
<code>repeatdelay</code>	Not supported.
<code>repeatinterval</code>	Not supported.
<code>state</code>	In <i>ttk</i> , there is no option with this name. The state mechanism has been generalized; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>wraplength</code>	If you use a style with this option set to some dimensions, the <code>text</code> will be sliced into pieces no longer than that dimension.

Methods on a `ttk.Button` include all those described in Section 46, “Methods common to all *ttk* widgets” (p. 145), plus:

`.invoke()`

Calls the button's `command` callback, and returns what that function returns. Has no effect if the button is disabled or there is no callback.

The `.flash()` method of `Tkinter.Button` is not supported by the `ttk.Button` widget.

30. `ttk.Checkbutton`

This widget is the *ttk* version of Section 9, “The Checkbutton widget” (p. 38). To create a `ttk.Checkbutton` widget as the child of a given `parent` widget:

```
w = ttk.Checkbutton(parent, option=value, ...)
```

Here are the options for the `ttk.Checkbutton` widget. Compare these to the `Tkinter` version discussed in Section 7, “The Button widget” (p. 18).

Table 37. `ttk.Checkbutton` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>command</code>	A function to be called whenever the state of this checkbutton changes.
<code>compound</code>	This option specifies the relative position of the image relative to the text when you specify both. The value may be <code>tk.TOP</code> (image above text), <code>tk.BOTTOM</code> (image below text), <code>tk.LEFT</code> (image to the left of the text), or <code>tk.RIGHT</code> (image to the right of the text). If you provide both <code>image</code> and <code>text</code> options but do not specify a value for <code>compound</code> , only the image will appear.
<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
<code>image</code>	An image to appear on the checkbutton; see Section 5.9, “Images” (p. 14).
<code>offvalue</code>	By default, when a checkbutton is in the off (unchecked) state, the value of the associated <code>variable</code> is 0. You can use the <code>offvalue</code> option to specify a different value for the off state.
<code>onvalue</code>	By default, when a checkbutton is in the on (checked) state, the value of the associated <code>variable</code> is 1. You can use the <code>onvalue</code> option to specify a different value for the on state.
<code>style</code>	The style to be used in rendering this checkbutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.Checkbutton</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
<code>text</code>	The text to appear on the checkbutton, as a string.
<code>textvariable</code>	A variable that controls the text that appears on the checkbutton; see Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>underline</code>	If this option has a nonnegative value n , an underline will appear under the <code>text</code> character at position n .
<code>variable</code>	A control variable that tracks the current state of the checkbutton; see Section 52, “Control variables: the values behind the widgets” (p. 153). Normally you will use an <code>IntVar</code> here, and the off and on values are 0 and 1, respectively. However, you may use a different control variable type, and specify the <code>offvalue</code> and <code>onvalue</code> options using values of that type.
<code>width</code>	Use this option to specify a fixed width or a minimum width. The value is specified in characters; a positive value sets a fixed width of that many average characters, while a negative width sets a minimum width. For example, if an average character in the selected font is 10 pixels wide, option <code>width=8</code> will make the text label exactly 80 pixels wide; option <code>width=-8</code> will use 80 pixels or the length of the text, whichever is larger. You may also specify a <code>width</code> value in an associated style. If values are specified both in the widget constructor call and in the style, the former takes priority.

These options of the `Tkinter Checkbutton` widget are *not* supported by the `ttk.Checkbutton` widget constructor:

Table 38. Tkinter Checkbutton options not in `ttk.Checkbutton`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>activeforeground</code>	Use a style map to control the <code>foreground</code> option.
<code>anchor</code>	Configure this option using a style; see Section 49, “Using and customizing <i>ttk</i> styles” (p. 147). Use this option to specify the position of the text when the <code>width</code> option allocates extra horizontal space. For example, if you specify options <code>width=20</code> and <code>compound=tk.TOP</code> on a checkbutton that displays both text and an image, and a style that specifies <code>anchor=tk.E</code> (east), the image will be at the right-hand end of the twenty-character space, with the text just below it. When a checkbutton displays an image but no text, this option is ignored.
<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style. The <code>bg</code> abbreviation is not supported.
<code>bitmap</code>	Not supported.
<code>borderwidth</code> or <code>bd</code>	Configure this option using a style.
<code>disabledforeground</code>	Use a style map for the <code>foreground</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>font</code>	Configure this option using a style.
<code>foreground</code> or <code>fg</code>	Configure this option using a style.
<code>height</code>	Not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the checkbutton does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.
<code>indicatoron</code>	Not supported.
<code>justify</code>	Controls how multiple lines are positioned horizontally relative to each other. Configure this option using a style; values may be <code>tk.LEFT</code> , <code>tk.CENTER</code> , or <code>tk.RIGHT</code> for left-aligned, centered, or right-aligned, respectively.
<code>offrelief</code>	Not supported.
<code>overrelief</code>	Use a style map to control the <code>relief</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.
<code>relief</code>	Use a style map to control the <code>relief</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>selectcolor</code>	Not supported.
<code>selectimage</code>	Not supported.

<code>state</code>	In <code>ttk</code> , there is no option with this name. The state mechanism has been generalized; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>wraplength</code>	If you use a style that has this option set to some dimension, the <code>text</code> will be sliced into pieces no longer than that dimension.

Methods on a `ttk.Checkbutton` include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus:

`.invoke()`

This method toggles the state of the checkbutton. If there is a `command` callback, it calls that callback, and returns whatever value the callback returned.

Not supported are the following methods of the `Tkinter Checkbutton` widget: `.deselect()`, `.flash()`, `.select()`, and `.toggle()`. To change the state of a checkbutton through program control, use the `.set()` method of the associated control `variable`.

31. `ttk.Combobox`

This widget is a combination of an `Entry` and a drop-down menu. In your application, you will see the usual text entry area, with a downward-pointing arrow. When the user clicks on the arrow, a drop-down menu appears. If the user clicks on one, that choice replaces the current contents of the entry. However, the user may still type text directly into the entry (when it has focus), or edit the current text.

To create a `ttk.Combobox` widget as the child of a given `parent` widget:

```
w = ttk.Combobox(parent, option=value, ...)
```

Options:

Table 39. `ttk.Combobox` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
<code>exportselection</code>	By default, if you select text within an <code>Entry</code> widget, it is automatically exported to the clipboard. To avoid this exportation, use <code>exportselection=0</code> .
<code>height</code>	Use this option to specify a maximum number of rows that will appear in the drop-down menu; the default is 20. If there are more <code>values</code> than this number, the drop-down menu will automatically include a vertical scrollbar.
<code>justify</code>	This option specifies how the text will be positioned within the entry area when it does not completely fill the area. Values may be <code>tk.LEFT</code> to left-justify; <code>tk.CENTER</code> to center; or <code>tk.RIGHT</code> to right-justify.
<code>postcommand</code>	You may use this option to supply a callback function that will be invoked when the user clicks on the down-arrow. This callback may change the <code>values</code> option; if so, the changes will appear in the drop-down menu.
<code>style</code>	The style to be used in rendering this checkbutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).

takefocus	By default, a <code>ttk.Checkbutton</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
textvariable	A variable that controls the text that appears in the entry area; see Section 52, “Control variables: the values behind the widgets” (p. 153).
validate	You may use this option to request dynamic validation of the widget’s text content. See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
validatecommand	You may use this option to specify a callback function that dynamically validates the widget’s text content. See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
values	The choices that will appear in the drop-down menu, as a sequence of strings.
width	This option specifies the width of the entry area as a number of characters. The actual width will be this number times the average width of a character in the effective font. The default value is 20.
xscrollcommand	If the widget has an associated horizontal scrollbar, set this option to the <code>.set</code> method of that scrollbar.

Methods on a `ttk.Combobox` include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus all the methods on the `Tkinter` widget described in Section 10, “The `Entry` widget” (p. 41), plus:

.current([index])

To select one of the elements of the `values` option, pass the index of that element as the argument to this method. If you do not supply an argument, the returned value is the index of the current entry text in the `values` list, or -1 if the current entry text is not in the `values` list.

.set(value)

Set the current text in the widget to `value`.

The states of a `ttk.Combobox` widget affect its operation. To interrogate or change states, see the `.instate()` and `.state()` methods in Section 46, “Methods common to all `ttk` widgets” (p. 145).

- If the widget is in the `disabled` state, no user action will change the contents.
- If the widget is in the `!disabled` state and also the `readonly` state, the user may change the contents by using the drop-down menu, but may not directly edit the contents.

32. `ttk.Entry`

The purpose of an `Entry` widget is to allow the user to enter or edit a single line of text. This is the `ttk` version of Section 10, “The `Entry` widget” (p. 41).

To create a `ttk.Entry` widget as the child of a given `parent` widget:

```
w = ttk.Entry(parent, option=value, ...)
```

Options:

Table 40. `ttk.Entry` options

class_	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
---------------	--

<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
<code>exportselection</code>	By default, if you select text within an <code>Entry</code> widget, it is automatically exported to the clipboard. To avoid this exportation, use <code>exportselection=0</code> .
<code>font</code>	Use this option to specify the font of the text that will appear in the widget; see Section 5.4, “Type fonts” (p. 10). For reasons that are unclear to the author, this option cannot be specified with a style.
<code>invalidcommand</code>	You may set this option to a callback function that will be called whenever validation fails (that is, when the <code>validatecommand</code> returns a 0). See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
<code>justify</code>	This option specifies how the text will be positioned within the entry area when it does not completely fill the area. Values may be <code>tk.LEFT</code> to left-justify; <code>tk.CENTER</code> to center; or <code>tk.RIGHT</code> to right-justify.
<code>show</code>	To protect fields such as passwords from being visible on the screen, set this option to a string, whose first character will be substituted for each of the actual characters in the field. For example, if the field contains “sesame” but you have specified <code>show='*'</code> , the field will appear as “*****”.
<code>style</code>	The style to be used in rendering this checkbutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.Checkbutton</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
<code>textvariable</code>	A variable that controls the text that appears in the entry area; see Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>validate</code>	You may use this option to specify a callback function that dynamically validates the widget’s text content. See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
<code>validatecommand</code>	See Section 10.2, “Adding validation to an <code>Entry</code> widget” (p. 45).
<code>width</code>	This option specifies the width of the entry area as a number of characters. The actual width will be this number times the average width of a character in the effective font. The default value is 20.
<code>xscrollcommand</code>	If the widget has an associated horizontal scrollbar, set this option to the <code>.set</code> method of that scrollbar.

These options of the `Tkinter Entry` widget are *not* supported by the `ttk.Entry` widget constructor:

Table 41. Tkinter Entry options not in `ttk.Entry`

<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style; see Section 47, “Customizing and creating <code>ttk</code> themes and styles” (p. 146). The <code>bg</code> abbreviation is not supported.
<code>borderwidth</code> or <code>bd</code>	Configure this option using a style.
<code>disabledbackground</code>	Use a style map for the <code>background</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>disabledforeground</code>	Use a style map for the <code>foreground</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>foreground</code> or <code>fg</code>	Configure this option using a style.

<code>highlightbackground</code>	To control the color of the focus highlight when the checkbutton does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.
<code>insertbackground</code>	Not supported.
<code>insertborderwidth</code>	Not supported.
<code>insertofftime</code>	Not supported.
<code>insertontime</code>	Not supported.
<code>insertwidth</code>	Not supported.
<code>readonlybackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>relief</code>	Configure this option using a style; see Section 47, “Customizing and creating <i>ttk</i> themes and styles” (p. 146).
<code>selectbackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>selectborderwidth</code>	Use a style map to control the <code>borderwidth</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>selectforeground</code>	Use a style map to control the <code>foreground</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).

Methods on a `ttk.Entry` include all those described in Section 46, “Methods common to all *ttk* widgets” (p. 145), plus all the methods on the *Tkinter* widget described in Section 10, “The `Entry` widget” (p. 41).

33. `ttk.Frame`

Like the *Tkinter Frame* widget, the `ttk.Frame` widget is a rectangular container for other widgets. To create a `Frame` widget as the child of a given `parent` widget:

```
w = ttk.Frame(parent, option=value, ...)
```

Options include:

Table 42. `ttk.Frame` options

<code>borderwidth</code>	Use this option to specify the width of the border element; the default is zero.
<code>class_</code>	You may provide a widget class name when you create this widget. This name may be used to customize the widget’s appearance; see Section 27, “Standardizing appearance” (p. 105). Once the widget is created, the widget class name cannot be changed.
<code>cursor</code>	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, “Cursors” (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.

<code>height</code>	This option is a dimension that sets the height of the frame. If you want to force the frame to have a specific height, call the <code>.grid_propagate(0)</code> on the widget; see Section 4.2, “Other grid management methods” (p. 7).
<code>padding</code>	To create an empty area inside the frame and outside of the contained widgets, set this option to the desired dimension. For example, <code>padding='0.5i'</code> would clear a half-inch-wide area inside the frame and around the outside of the widgets inside it.
<code>relief</code>	Specifies the relief style for the border; see Section 5.6, “Relief styles” (p. 12). This has no effect unless you also increase the <code>borderwidth</code> .
<code>style</code>	Use this option to specify a custom widget style name; see Section 47, “Customizing and creating <code>ttk</code> themes and styles” (p. 146).
<code>takefocus</code>	Use this option to specify whether a widget is visited during focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). Specify <code>takefocus=True</code> if you want the visit to accept focus; specify <code>takefocus=False</code> if the widget is not to accept focus. The default value is an empty string; by default, <code>ttk.Frame</code> widgets do not get focus.
<code>width</code>	This option is a dimension that sets the width of the frame. If you want to force the frame to have a specific width, call the <code>.grid_propagate(0)</code> on the widget; see Section 4.2, “Other grid management methods” (p. 7).

These options on the `Tkinter Frame` widget are *not* available as options on the `ttk.Frame` constructor:

Table 43. Tkinter Frame options not in `ttk.Frame`

<code>background</code> or <code>bg</code>	Configure this option with a style; see Section 47, “Customizing and creating <code>ttk</code> themes and styles” (p. 146).
<code>highlightbackground</code>	To control the color of the focus highlight when the frame does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.

34. `ttk.Label`

The purpose of this widget is to display text, an image, or both. Generally the content is static, but your program can change the text or the image.

To create a `ttk.Label` widget as the child of a given `parent` widget:

```
w = ttk.Label(parent, option=value, ...)
```

Options include:

Table 44. `ttk.Label` options

<code>anchor</code>	If the text and/or image are smaller than the specified <code>width</code> , you can use the <code>anchor</code> option to specify where to position them: <code>tk.W</code> , <code>tk.CENTER</code> , or <code>tk.E</code>
---------------------	--

	for left, centered, or right alignment, respectively. You may also specify this option using a style.														
background	Use this option to set the background color. You may also specify this option using a style.														
borderwidth	To add a border around the label, set this option to the width dimension. You may also specify this option using a style.														
class_	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, "Standardizing appearance" (p. 105). Once the widget is created, the widget class name cannot be changed.														
compound	If you provide both <code>text</code> and <code>image</code> options, the <code>compound</code> option specifies how to display them. <table border="1" data-bbox="507 650 1432 994"> <tr> <td>'bottom'</td><td>Display the image below the text.</td></tr> <tr> <td>'image'</td><td>Display only the image, not the text.</td></tr> <tr> <td>'left'</td><td>Display the image to the left of the text.</td></tr> <tr> <td>'none'</td><td>Display the image if there is one, otherwise display the text. This is the default value.</td></tr> <tr> <td>'right'</td><td>Display the image to the right of the text.</td></tr> <tr> <td>'text'</td><td>Display the text, not the image.</td></tr> <tr> <td>'top'</td><td>Display the image above the text.</td></tr> </table>	'bottom'	Display the image below the text.	'image'	Display only the image, not the text.	'left'	Display the image to the left of the text.	'none'	Display the image if there is one, otherwise display the text. This is the default value.	'right'	Display the image to the right of the text.	'text'	Display the text, not the image.	'top'	Display the image above the text.
'bottom'	Display the image below the text.														
'image'	Display only the image, not the text.														
'left'	Display the image to the left of the text.														
'none'	Display the image if there is one, otherwise display the text. This is the default value.														
'right'	Display the image to the right of the text.														
'text'	Display the text, not the image.														
'top'	Display the image above the text.														
cursor	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, "Cursors" (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.														
font	Use this option to specify the font style for the displayed <code>text</code> . You may also specify this option using a style.														
foreground	Use this option to specify the color of the displayed <code>text</code> . You may also specify this option using a style.														
image	This option specifies an image or images to be displayed either in addition to or instead of text. The value must be an image as specified in Section 5.9, "Images" (p. 14). See the <code>compound</code> option above for what happens when you supply both image and text. <p>You may specify multiple images that will appear on the widget depending on the state of the widget (see Section 50.2, "ttk style maps: dynamic appearance changes" (p. 151) for a discussion of widget states). To do this, supply as the value of this option a tuple $(i_0, s_1, i_1, s_2, i_2, \dots)$, where:</p> <ul style="list-style-type: none"> • i_0 is the default image to be displayed on the widget. • For each pair of values after the first, s_i specifies a state or combination of states, and i_1 specifies the image to be displayed when the widget's state matches s_i. <p>Each state specifier s_i may be a single state name, optionally preceded by '!', or a sequence of such names. The ! specifies that the widget must <i>not</i> be in that state.</p>														

	<p>For example, suppose you have three <code>PhotoImage</code> instances named <code>im1</code>, <code>im2</code>, and <code>im3</code>, and in your call to the <code>Label</code> constructor you include this option:</p> <pre>self.w = ttk.Label(self, ..., image=(im1, 'selected', im2, '!disabled', 'alternate'), im3), ...)</pre> <p>The widget will display image <code>im2</code> if it is in the <code>selected</code> state. If it is not in the <code>selected</code> state or the <code>disabled</code> state but it is in the <code>alternate</code> state, it will display image <code>im3</code>. Otherwise it will display image <code>im1</code>.</p>
<code>justify</code>	If the <code>text</code> you provide contains newline ('\n') characters, this option specifies how each line will be positioned horizontally: <code>tk.LEFT</code> to left-justify; <code>tk.CENTER</code> to center; or <code>tk.RIGHT</code> to right-justify each line. You may also specify this option using a style.
<code>padding</code>	To add more space around all four sides of the text and/or image, set this option to the desired dimension. You may also specify this option using a style.
<code>relief</code>	Set this option to a relief style to create a 3-d effect. You will need to increase the <code>borderwidth</code> to make this effect appear. You may also specify this option using a style.
<code>style</code>	Use this option to specify a custom widget style name; see Section 47, “Customizing and creating <code>ttk</code> themes and styles” (p. 146).
<code>takefocus</code>	<p>Use this option to specify whether the widget is visited during focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). Specify <code>takefocus=True</code> if you want the visit to accept focus; specify <code>takefocus=False</code> if the widget is not to accept focus.</p> <p>The default value is an empty string; by default, <code>ttk.Label</code> widgets do not get focus.</p>
<code>text</code>	A string of text to be displayed in the widget.
<code>textvariable</code>	A <code>StringVar</code> instance (see Section 52, “Control variables: the values behind the widgets” (p. 153)); the text displayed on the widget will be its value. If both <code>text</code> and <code>textvariable</code> are specified, the <code>text</code> option will be ignored.
<code>underline</code>	<p>You can request that one of the letters in the text string be underline by setting this option to the position of that letter. For example, the options <code>text='Quit'</code> and <code>underline=0</code> would underline the Q.</p> <p>Using this option doesn't change anything functionally. If you want the application to react to the Q key or some variation like control-shift-Q, you'll need to set up the bindings using the event system.</p>
<code>width</code>	<p>To specify a fixed width, set this option to the number of characters. To specify a minimum width, set this option to minus the number of characters. If you don't specify this option, the size of the label area will be just enough to accommodate the current text and/or image.</p> <p>For text displayed in a proportional font, the actual width of the widget will be based on the average width of a character in the font, and not a specific number of characters.</p> <p>This option may also be specified through a style.</p>

<code>wraplength</code>	If you set this option to some dimension, all the text will be chopped into lines no longer than this dimension. This option may also be specified through a style.
-------------------------	---

The following options of the *Tkinter* version of `Label` are *not* supported by the `ttk.Label` constructor.

Table 45. Tkinter Label options not in `ttk.Label`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>activeforeground</code>	Use a style map to control the <code>foreground</code> option.
<code>bitmap</code>	Not supported.
<code>disabledforeground</code>	Use a style map for the <code>foreground</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>height</code>	Not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the label does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.

35. `ttk.LabelFrame`

This is the *ttk* version of the basic *Tkinter* widget described in Section 13, “The `LabelFrame` widget” (p. 50).

To create a new `ttk.LabelFrame` widget as a child of a given `parent` widget:

```
w = ttk.LabelFrame(parent, option=value, ...)
```

Options include:

Table 46. `ttk.LabelFrame` options

<code>borderwidth</code>	Use this option to set the width of the border around the widget to a given dimension. This option may also be configured using a style.
<code>class_</code>	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, “Standardizing appearance” (p. 105). Once the widget is created, the widget class name cannot be changed.
<code>cursor</code>	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, “Cursors” (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.
<code>height</code>	This option can be set to some dimension to specify the height of the frame. If you don't call the <code>.grid_propagate(0)</code> method, this option will be ignored; see Section 4.2, “Other grid management methods” (p. 7).

<code>labelanchor</code>	Use this option to specify the position of the label on the widget's border. The default position is ' <code>nw</code> ', which places the label at the left end of the top border. For the nine possible label positions, refer to Section 13, "The <code>LabelFrame</code> widget" (p. 50).
<code>labelwidget</code>	Instead of a text label, you can use any widget as the label in a <code>ttk.LabelFrame</code> . Create some widget <code>w</code> but do not register it with the <code>.grid()</code> method. Then create the <code>LabelFrame</code> with <code>labelwidget=w</code> . If you specify this option as well as the <code>text</code> option, the latter is ignored. For example, if you don't like the rather small and plain default font used for the label, you can use this option to display a <code>Label</code> widget with the font and other appearance of your choice.
<code>padding</code>	To add extra clear area around the contents of this widget, set this option to a dimension. This option may also be specified by a style.
<code>relief</code>	Use this option to specify a 3-d border style; see Section 5.6, "Relief styles" (p. 12). You will need to specify a nonzero <code>borderwidth</code> for this effect to appear. This option may also be specified by a style.
<code>style</code>	Use this option to specify a custom widget style name; see Section 47, "Customizing and creating <code>ttk</code> themes and styles" (p. 146).
<code>takefocus</code>	Use this option to specify whether the widget is visited during focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). Specify <code>takefocus=True</code> if you want the visit to accept focus; specify <code>takefocus=False</code> if the widget is not to accept focus. The default value is an empty string; by default, <code>ttk.Label</code> widgets do not get focus.
<code>text</code>	The value of this option is a string that will appear as part of the border.
<code>underline</code>	You can request that one of the letters in the text string be underline by setting this option to the position of that letter. For example, if you specified <code>text='Panic'</code> and <code>underline=2</code> , an underline would appear under the ' <code>n</code> '. Using this option doesn't change anything functionally. If you want the application to react to the <code>Q</code> key or some variation like control-shift- <code>Q</code> , you'll need to set up the bindings using the event system.
<code>width</code>	This option can be set to some dimension to specify the width of the frame. If you don't call the <code>.grid_propagate(0)</code> method, this option will be ignored; see Section 4.2, "Other grid management methods" (p. 7).

The following options available for the `Tkinter LabelFrame` widget are *not* available as constructor arguments.

Table 47. Tkinter LabelFrame options not in `ttk.LabelFrame`

<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style; see Section 47, "Customizing and creating <code>ttk</code> themes and styles" (p. 146). The <code>bg</code> abbreviation is not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the <code>LabelFrame</code> does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, " <code>ttk</code> style maps: dynamic appearance changes" (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style. This option may not work in all themes.

The `ttk.LabelFrame` widget supports all the methods described in Section 46, “Methods common to all `ttk` widgets” (p. 145).

36. `ttk.Menubutton`

A `Menubutton` widget is the part of a drop-down menu that is always visible. It is always used in combination with a `Menu` widget that controls what appears when the user clicks on the `Menubutton`.

There is no `ttk` version of the `Menu` widget. Use the regular `Tkinter` widget described in Section 15, “The `Menu` widget” (p. 56).

To create a new `ttk.Menubutton` widget as the child of some `parent` widget, use this constructor:

```
w = ttk.Menubutton(parent, option=value, ...)
```

Options include:

Table 48. `ttk.Menubutton` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).										
<code>compound</code>	If you provide both <code>image</code> and <code>text</code> options, the <code>compound</code> option specifies the position of the image relative to the text. The value may be <code>tk.TOP</code> (image above text), <code>tk.BOTTOM</code> (image below text), <code>tk.LEFT</code> (image to the left of the text), or <code>tk.RIGHT</code> (image to the right of the text). When you provide both <code>image</code> and <code>text</code> options but don't specify a <code>compound</code> option, the image will appear and the text will not.										
<code>cursor</code>	The cursor that will appear when the mouse is over the button; see Section 5.8, “Cursors” (p. 13).										
<code>direction</code>	This option specifies the position where the drop-down menu appears, relative to the menubutton. <table border="1"><tbody><tr><td><code>above</code></td><td>The menu will appear just above the menubutton.</td></tr><tr><td><code>below</code></td><td>The menu will appear just below the menubutton.</td></tr><tr><td><code>flush</code></td><td>The menu will appear over the menubutton, so that the menu's northwest corner coincides with the menubutton's northwest corner.</td></tr><tr><td><code>left</code></td><td>The menu will appear just to the left of the menubutton.</td></tr><tr><td><code>right</code></td><td>The menu will appear just to the right of the menubutton.</td></tr></tbody></table>	<code>above</code>	The menu will appear just above the menubutton.	<code>below</code>	The menu will appear just below the menubutton.	<code>flush</code>	The menu will appear over the menubutton, so that the menu's northwest corner coincides with the menubutton's northwest corner.	<code>left</code>	The menu will appear just to the left of the menubutton.	<code>right</code>	The menu will appear just to the right of the menubutton.
<code>above</code>	The menu will appear just above the menubutton.										
<code>below</code>	The menu will appear just below the menubutton.										
<code>flush</code>	The menu will appear over the menubutton, so that the menu's northwest corner coincides with the menubutton's northwest corner.										
<code>left</code>	The menu will appear just to the left of the menubutton.										
<code>right</code>	The menu will appear just to the right of the menubutton.										
<code>image</code>	An image to appear on the menubutton; see Section 5.9, “Images” (p. 14).										
<code>menu</code>	The related <code>Menu</code> widget. See Section 15, “The <code>Menu</code> widget” (p. 56) for the procedure used to establish this mutual connection.										
<code>style</code>	The style to be used in rendering this menubutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).										
<code>takefocus</code>	By default, a <code>ttk.Menubutton</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .										
<code>text</code>	The text to appear on the menubutton, as a string.										

<code>textvariable</code>	A variable that controls the text that appears on the menubutton; see Section 52, “Control variables: the values behind the widgets” (p. 153).
<code>underline</code>	If this option has a nonnegative value n , an underline will appear under the character at position n .
<code>width</code>	If the label is text, this option specifies the absolute width of the text area on the button, as a number of characters; the actual width is that number multiplied by the average width of a character in the current font. For image labels, this option is ignored. The option may also be configured in a style.

The following options of the `Tkinter Menobutton` button, described in Section 16, “The Menobutton widget” (p. 61), are *not* supported by `ttk.Menobutton`:

Table 49. Tkinter Menobutton options not in `ttk.Menobutton`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>activeforeground</code>	Use a style map to control the <code>foreground</code> option.
<code>anchor</code>	Configure this option using a style; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147). Use this option to specify the position of the text when the <code>width</code> option allocates extra horizontal space.
<code>bitmap</code>	Not supported.
<code>borderwidth</code> or <code>bd</code>	Configure the <code>borderwidth</code> option using a style. The <code>bd</code> abbreviation is not supported.
<code>buttonbackground</code>	Not supported.
<code>buttoncursor</code>	Not supported.
<code>buttondownrelief</code>	Not supported.
<code>buttonup</code>	Not supported.
<code>disabledforeground</code>	Use a style map for the <code>foreground</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>font</code>	Configure this option using a style.
<code>foreground</code> or <code>fg</code>	Configure the <code>foreground</code> option using a style.
<code>height</code>	Not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the menubutton does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style.
<code>justify</code>	If the <code>text</code> contains newline ('\n') characters, the text will occupy multiple lines on the menubutton. The <code>justify</code> option controls how each line is positioned horizontally. Configure this option using a style; values may be <code>tk.LEFT</code> , <code>tk.CENTER</code> , or <code>tk.RIGHT</code> for lines that are left-aligned, centered, or right-aligned, respectively.
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.

<code>relief</code>	Configure this option using a style; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>wraplength</code>	If you use a style with this option set to some dimension, the <code>text</code> will be sliced into pieces no longer than that dimension.

37. `ttk`.Notebook

The purpose of a `Notebook` widget is to provide an area where the user can select pages of content by clicking on tabs at the top of the area, like these:



- Each time the user clicks on one of these tabs, the widget will display the *child pane* associated with that tab. Typically, each pane will be a `Frame` widget, although a pane can be any widget.
- The tab for the child pane that is currently showing is referred to as the *selected* tab.
- You will use the `Notebook` widget's `.add()` method to attach a new tab, and its related content. Other methods let you remove or temporarily hide tabs.
- Each tab has its own set of options that control its appearance and behavior. These options are described in Table 51, “Tab options for the `ttk`.`Notebook` widget” (p. 128).
- A number of the methods of this widget use the idea of a `tabId` to refer to one of the tabs. Different values for a `tabId` may be any of:
 - Integer values refer to the position of the tab: 0 for the first tab, 1 for the second, and so forth.
 - You can always refer to a tab by using the child widget itself.
 - A string of the form " $@x,y$ " refers to the tab that currently contains the point (x,y) relative to the widget. For example, the string " $@37,0$ " would specify the tab containing a point 37 pixels from the left side of the widget, along the top edge of the tab.
 - The string "current" refers to whichever tab is currently selected.
 - In a call to the `Notebook` widget's `.index()` method, use the string "end" to determine the current number of tabs displayed.

To create a `Notebook` widget as the child of some *parent* widget, use this constructor:

```
w = ttk.Notebook(parent, option=value, ...)
```

Options include:

Table 50. `ttk`.`Notebook` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>cursor</code>	The cursor that will appear when the mouse is over the notebook; see Section 5.8, “Cursors” (p. 13).
<code>height</code>	The height in pixels to be allocated to the widget.
<code>padding</code>	To add some extra space around the outside of the widget, set this option to that amount of space as a dimension.
<code>style</code>	The style to be used in rendering this menubutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).

<code>takefocus</code>	By default, a <code>ttk.Notebook</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
<code>width</code>	The width in pixels to be allocated to the widget.

Methods on a `ttk.Notebook` widget include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus:

`.add(child, **kw)`

The `child` argument is a widget, usually a `Frame`, that wraps the content of one child pane. If `child` is not one of the `Notebook` widget’s child panes, `child` is added as the next available tab, and the keyword arguments `kw` define the tab options for the new pane. These options are defined in Table 51, “Tab options for the `ttk.Notebook` widget” (p. 128).

If `child` is a currently hidden pane, that tab will reappear in its former position.

`.enable_traversal()`

Once you call this method, a few additional key bindings will work:

- *Control-Tab* will select the tab after the one currently selected. If the last tab was currently selected, selection will rotate back to the first tab.
- *Shift-Control-Tab* does the reverse: it moves to the previous tab, wrapping around to the last tab if the first tab was selected.
- You can configure a particular hot key that directly selects a tab. To do this, use the `text` and `underline` tab options to underline one of the characters in each tab. Then the user can jump to a tab by typing *Alt-X* where *X* is the underlined character on that tab.

If you have multiple `Notebook` widgets in the same application, these features will not work correctly unless each child pane widget is created with its `Notebook` widget as the parent.

`.forget(child)`

This method permanently removes the specified `child` from the widget’s set of tabs.

`.hide(tabId)`

The tab identified by `tabId` is temporarily removed from the set of visible tabs in the `Notebook`. You may reinstate it by calling the `.add()` method again.

`.index(tabId)`

For a given `tabId`, this method returns the numeric index of the corresponding tab. There is one exception: if the argument is the string “end”, the method will return the total number of tabs.

`.insert(where, child, **kw)`

This method inserts the widget `child` at the position specified by `where`, using any keyword arguments to describe the new tab and pane. For the keyword options, see Table 51, “Tab options for the `ttk.Notebook` widget” (p. 128).

The `where` argument may be any of:

- “end” to place the new tab after all the existing tabs.
- An existing child widget; in this case the new `child` is inserted just before that existing widget.

`.select([tabId])`

If you call this method without an argument, it will return the window name of the widget whose tab is currently displayed.

To display a specific pane in the `Notebook`, call this method with a `tabId` as the argument.

.tab(tabId, option=None, **kw)

Use this method either to set tab options for the child panes described by *tagId*, or to find out what options are set for that child pane. The tab options are described in Table 51, “Tab options for the *ttk.Notebook* widget” (p. 128).

If you call the method with no keyword arguments, it will return a dictionary of the tab options currently in effect for the pane specified by *tagId*.

To find out the current value of a specific tab option *X*, call this method with the argument “*option=X*”, and the method will return the value of that tab option.

To set one or more tab options for the child described by *tagId*, call this method with keyword arguments. For example, if *self.nb* is a *Notebook*, this call would change the text displayed on the first tab:

```
self.nb.tab(0, text='Crunchy frog')
```

.tabs()

This method returns a list of the window names of the *Notebook*'s child panes, in order from first to last.

Here are the tab options used in the *.add()* and *.tab()* methods.

Table 51. Tab options for the *ttk.Notebook* widget

compound	If you supply both <i>image</i> and <i>text</i> to be displayed on the tab, the <i>compound</i> option specifies how to display them. Allowable values describe the position of the image relative to the text, and may be any of <i>tk.BOTTOM</i> , <i>tk.TOP</i> , <i>tk.LEFT</i> , <i>tk.RIGHT</i> , or <i>tk.CENTER</i> . For example, <i>compound=tk.LEFT</i> would position the image to the left of the text.
padding	Use this option to add extra space around all four sides of the panel's content. The value is a dimension. For example, <i>padding='0.1i'</i> would add a 0.1" space around each side of the panel content.
sticky	Use this option to specify where the panel content is positioned if it does not completely fill the panel area. Values are the same as for the <i>sticky</i> argument described in Section 4.1, “The <i>.grid()</i> method” (p. 6). For example, <i>sticky=tk.E+tk.S</i> would position the content in the lower right (southeast) corner.
image	To make a graphic image appear on the tab, supply an image as the value of this option. Refer to the <i>compound</i> option above to specify the relative positions of <i>image</i> and <i>text</i> when you specify both.
text	The text to appear on the tab.
underline	If this option has a nonnegative value <i>n</i> , an underline will appear under the character at position <i>n</i> of the text on the tab.

37.1. Virtual events for the *ttk.Notebook* widget

Whenever the selected tab changes in a *ttk.Notebook* widget, it generates a “*<<NotebookTabChanged>>*” virtual event; see Section 54.8, “Virtual events” (p. 165).

38. `ttk.PanedWindow`

This is the `ttk` version of Section 19, “The `PanedWindow` widget” (p. 65). To create a `ttk.PanedWindow` widget as the child of a given *parent* widget:

```
w = ttk.PanedWindow(parent, option=value, ...)
```

The options for this constructor are given in Table 52, “`ttk.PanedWindow` options” (p. 129).

Table 52. `ttk.PanedWindow` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
<code>height</code>	The height dimension of the widget.
<code>orient</code>	To stack child widgets side by side, use <code>orient=tk.HORIZONTAL</code> . To stack them top to bottom, use <code>orient=tk.VERTICAL</code> . The default option is <code>tk.VERTICAL</code> .
<code>style</code>	The style to be used in rendering this widget; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.PanedWindow</code> will not be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To add the widget to focus traversal, use <code>takefocus=True</code> .
<code>width</code>	The width dimension of the widget.

These options of the `Tkinter.PanedWindow` widget are *not* supported by the `ttk.PanedWindow` constructor:

Table 53. `Tkinter PanedWindow` options not in `ttk.PanedWindow`

<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style. The <code>bg</code> abbreviation is not supported.
<code>borderwidth</code> or <code>bd</code>	Not supported.
<code>cursor</code>	The cursor that will appear when the mouse is over the widget; see Section 5.8, “Cursors” (p. 13).
<code>handlepad</code>	Not supported.
<code>handlesize</code>	Not supported.
<code>opaqueresize</code>	Not supported.
<code>relief</code>	Not supported.
<code>sashrelief</code>	Not supported.
<code>sashwidth</code>	Not supported.
<code>showhandle</code>	Not supported.

Methods on a `ttk.PanedWindow` include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus:

.add(*w*[, *weight=N*])

Add a new pane to the window, where *w* is any widget (but typically a `Frame`). If you provide a `weight` option, it describes the size of the pane in the stacking dimension, relative to the other panes. For example, for `orient=tk.VERTICAL`, if pane 0 has `weight=1` and pane 1 has `weight=3`, initially the first pane will have 1/4 of the height and the second pane will have 3/4.

.forget(*what*)

Remove a pane. The argument may be either the index of the pane, counting from zero, or the child widget.

.insert(*where*, *w*[, *weight=N*])

Add a new pane *w* to the window at the position specified by *where*, where *where* may be either an index or the pane widget before which you want to insert the new pane.

.panes()

This method returns a list of the `PanedWindow`'s child widgets.

39. *ttk.ProgressBar*

The purpose of this widget is to reassure the user that something is happening. It can operate in one of two modes:

- In `determinate` mode, the widget shows an indicator that moves from beginning to end under program control.
- In `indeterminate` mode, the widget is animated so the user will believe that something is in progress. In this mode, the indicator bounces back and forth between the ends of the widget.

In either mode, the current position of the indicator has a numeric value. You can specify a `maximum` value, and you can set the indicator value directly. You may also specify that the indicator value moves a given amount every time a given time interval passes.

To create a new `ttk.ProgressBar` widget as the child of a given *parent* widget:

```
w = ttk.ProgressBar(parent, option=value, ...)
```

The options for this constructor are given in Table 54, “`ttk.ProgressBar` options” (p. 130).

Table 54. *ttk.ProgressBar* options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>cursor</code>	The cursor that will appear when the mouse is over the checkbutton; see Section 5.8, “Cursors” (p. 13).
<code>length</code>	The size of the widget along its long axis as a dimension.
<code>maximum</code>	The maximum value of the indicator; default is 100.
<code>mode</code>	If your program cannot accurately depict the relative progress that this widget is supposed to display, use <code>mode='indeterminate'</code> . In this mode, a rectangle bounces back and forth between the ends of the widget once you use the <code>.start()</code> method. If your program has some measure of relative progress, use <code>mode='determinate'</code> . In this mode, your program can move the indicator to a specified position along the widget's track.

<code>orient</code>	This option specifies the orientation: use <code>orient=tk.HORIZONTAL</code> or <code>orient=tk.VERTICAL</code> .
<code>style</code>	The style to be used in rendering this widget; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.ProgressBar</code> will not be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To add the widget to focus traversal, use <code>takefocus=True</code> .
<code>variable</code>	Use this option to link a control variable to the widget so that you can get or set the current value of the indicator.

Methods on a `ttk.ProgressBar` include those described in Section 46, “Methods common to all `ttk` widgets” (p. 145) plus:

.start([interval])

Start moving the indicator every `interval` milliseconds; the default is 50ms. Each time, the indicator is moved as if you called the `.step()` method.

.step([delta])

This method increases the indicator value by `delta`; the default increment is 1.0. In determinate mode, the indicator will never exceed the value of the `maximum` option. In indeterminate mode, the indicator will reverse direction and count down once it reaches the maximum value.

.stop()

This method stops the automatic progress that was started by calling the `.start()` method.

40. `ttk.Radiobutton`

This widget is the `ttk` version of Section 20, “The Radiobutton widget” (p. 68). To create a `ttk.Radiobutton` widget as the child of a given `parent` widget, where the `option` values are given in Table 55, “`ttk.Radiobutton` options” (p. 131):

```
w = ttk.Radiobutton(parent, option=value, ...)
```

Table 55. `ttk.Radiobutton` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>command</code>	A function to be called whenever the state of this radiobutton changes.
<code>compound</code>	This option specifies the relative position of the image relative to the text when you specify both. The value may be <code>tk.TOP</code> (image above text), <code>tk.BOTTOM</code> (image below text), <code>tk.LEFT</code> (image to the left of the text), or <code>tk.RIGHT</code> (image to the right of the text). If you provide both <code>image</code> and <code>text</code> options but do not specify a value for <code>compound</code> , only the image will appear.
<code>cursor</code>	The cursor that will appear when the mouse is over the radiobutton; see Section 5.8, “Cursors” (p. 13).
<code>image</code>	An image to appear on the radiobutton; see Section 5.9, “Images” (p. 14).
<code>style</code>	The style to be used in rendering this radiobutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).

takefocus	By default, a <code>ttk.Radiobutton</code> will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
text	The text to appear next to the radiobutton, as a string.
textvariable	A variable that controls the text that appears on the radiobutton; see Section 52, “Control variables: the values behind the widgets” (p. 153).
underline	If this option has a nonnegative value n , an underline will appear under the <code>text</code> character at position n .
value	The value associated with this radiobutton. When the radiobutton is the one selected in the group, the value of this option will be stored in the control variable for the group.
variable	A control variable that is shared by the other radiobuttons in the group; see Section 52, “Control variables: the values behind the widgets” (p. 153). The type of this variable will be the same as the type you specify for the <code>value</code> options for the radiobuttons in the group.
width	<p>Use this option to specify a fixed width or a minimum width. The value is specified in characters; a positive value sets a fixed width of that many average characters, while a negative width sets a minimum width.</p> <p>You may also specify a <code>width</code> value in an associated style. If values are specified both in the widget constructor call and in the style, the former takes priority.</p>

These options of the `Tkinter Radiobutton` widget are *not* supported by the `ttk.Radiobutton` constructor:

Table 56. `ttk Radiobutton` options not in `ttk.Radiobutton`

activebackground	Use a style map to control the <code>background</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
activeforeground	Use a style map to control the <code>foreground</code> option.
anchor	<p>Configure this option using a style; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147). Use this option to specify the position of the text when the <code>width</code> option allocates extra horizontal space.</p> <p>For example, if you specify options <code>width=30</code> and <code>compound=tk.BOTTOM</code> on a radiobutton that displays both text and an image, and a style that specifies <code>anchor=tk.W</code> (west), the image will be at the left-hand end of the thirty-character space, with the text just above it.</p> <p>When a radiobutton displays an image but no text, this option is ignored.</p>
background or bg	Configure the <code>background</code> option using a style. The <code>bg</code> abbreviation is not supported.
bitmap	Not supported.
borderwidth or bd	Configure this option using a style.
disabledforeground	Use a style map for the <code>foreground</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
font	Configure this option using a style.
foreground or fg	Configure the <code>foreground</code> option using a style. The <code>fg</code> abbreviation is not supported.

<code>height</code>	Not supported.
<code>highlightbackground</code>	To control the color of the focus highlight when the menubutton does not have focus, use a style map to control the <code>highlightcolor</code> option; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>highlightcolor</code>	You may specify the default focus highlight color by setting this option in a style. You may also control the focus highlight color using a style map.
<code>highlightthickness</code>	Configure this option using a style.
<code>indicatoron</code>	Not supported.
<code>justify</code>	Controls how multiple lines are positioned horizontally relative to each other. Configure this option using a style; values may be <code>tk.LEFT</code> , <code>tk.CENTER</code> , or <code>tk.RIGHT</code> for left-aligned, centered, or right-aligned, respectively.
<code>offrelief</code>	Not supported.
<code>overrelief</code>	Not supported.
<code>padx</code>	Not supported.
<code>pady</code>	Not supported.
<code>relief</code>	Configure this option using a style.
<code>selectcolor</code>	Not supported.
<code>selectimage</code>	Not supported.
<code>state</code>	In <i>ttk</i> , there is no option with this name. The state mechanism has been generalized; see Section 50.2, “ <i>ttk</i> style maps: dynamic appearance changes” (p. 151).
<code>wraplength</code>	If you use a style that has this option set to some dimension, the <code>text</code> will be sliced into pieces no longer than that dimension.

Methods on a `ttk.Radiobutton` include all those described in Section 46, “Methods common to all *ttk* widgets” (p. 145), plus:

`.invoke()`

When you call this method on a `ttk.Radiobutton`, the result is the same as if the user clicked on it: the indicator is set and the associated `variable` is set to the radiobutton’s `value`. If there is a command associated with this radiobutton, that function is called, and the `.invoke()` method returns whatever the function returned; otherwise it returns `None`.

41. `ttk.Scale`

This is the *ttk* version of Section 21, “The `Scale` widget” (p. 71). To create a `ttk.Scale` widget as the child of a given `parent` widget, where the `option` values are given in Table 57, “*ttk.Scale* options” (p. 133):

```
w = ttk.Scale(parent, option=value, ...)
```

Table 57. `ttk.Scale` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
---------------------	--

<code>command</code>	A function to be called whenever the state of this widget changes. This function will receive one argument, the new value shown on the widget, as a <code>float</code> .
<code>cursor</code>	The cursor that will appear when the mouse is over the scale; see Section 5.8, “Cursors” (p. 13).
<code>from_</code>	Use this option in combination with the <code>to</code> option (described below) to constrain the values to a numeric range. For example, <code>from_=-10</code> and <code>to=10</code> would allow only values between -10 and 20 inclusive. See also the <code>increment</code> option below.
<code>length</code>	The length of the scale widget. This is the <code>x</code> dimension if the scale is horizontal, or the <code>y</code> dimension if vertical. The default is 100 pixels. For allowable values, see Section 5.1, “Dimensions” (p. 9).
<code>orient</code>	Set <code>orient=tk.HORIZONTAL</code> if you want the scale to run along the <code>x</code> dimension, or <code>orient=tk.VERTICAL</code> to run parallel to the <code>y</code> -axis. Default is vertical.
<code>style</code>	The style to be used in rendering this radiobutton; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.Scale</code> widget will be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To remove the widget from focus traversal, use <code>takefocus=False</code> .
<code>to</code>	A <code>float</code> value that defines one end of the scale's range; the other end is defined by the <code>from_</code> option, discussed above. The <code>to</code> value can be either greater than or less than the <code>from_</code> value. For vertical scales, the <code>to</code> value defines the bottom of the scale; for horizontal scales, the right end. The default value is 100.
<code>value</code>	Use this option to set the initial value of the widget's <code>variable</code> ; the default is 0.0.
<code>variable</code>	Use this option to associate a control variable with the widget. Typically this will be a <code>tk.DoubleVar</code> instance, which holds a value of type <code>float</code> . You may instead use a <code>tk.IntVar</code> instance, but values stored in it will be truncated as type <code>int</code> .

These options of the `Tkinter Scale` widget are *not* supported by the `ttk.Scale` widget constructor:

Table 58. Tkinter Scale options not in `ttk.Scale`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style; this option controls the color of the slider. The <code>bg</code> abbreviation is not supported.
<code>borderwidth</code> or <code>bd</code>	Configure this option using a style.
<code>digits</code>	Not supported.
<code>font</code>	Not supported.
<code>foreground</code> or <code>fg</code>	Not supported.
<code>highlightbackground</code>	Not supported.
<code>highlightcolor</code>	Not supported.
<code>highlightthickness</code>	Not supported.
<code>label</code>	Not supported.
<code>relief</code>	Not supported.
<code>repeatdelay</code>	Not supported.
<code>repeatinterval</code>	Not supported.

<code>resolution</code>	Not supported.
<code>showvalue</code>	Not supported.
<code>sliderlength</code>	Configure this option using a style.
<code>sliderrelief</code>	Configure this option using a style.
<code>state</code>	In <code>ttk</code> , there is no option with this name. The state mechanism has been generalized; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>tickinterval</code>	Not supported.
<code>troughcolor</code>	Configure this option using a style.
<code>width</code>	Configure this option using the <code>sliderthickness</code> option in a style.

Methods on a `ttk.Scale` include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus:

.get()

Returns the current value shown on the widget.

.set(*newValue*)

Change the widget's current value to *newValue*.

42. `ttk.Scrollbar`

This is the `ttk` version of Section 22, “The `Scrollbar` widget” (p. 74). To create a `ttk.Scrollbar` as the child of a given `parent` widget, where the `option` values are given in Table 59, “`ttk.Scrollbar` options” (p. 135):

```
w = ttk.Scrollbar(parent, option=value, ...)
```

Table 59. `ttk.Scrollbar` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>command</code>	A procedure to be called whenever the scrollbar is moved. For a discussion of the calling sequence, see Section 22.1, “The <code>Scrollbar</code> <code>command</code> callback” (p. 77).
<code>cursor</code>	The cursor that will appear when the mouse is over the scrollbar; see Section 5.8, “Cursors” (p. 13).
<code>orient</code>	Set <code>orient=tk.HORIZONTAL</code> for a horizontal scrollbar, <code>orient=tk.VERTICAL</code> for a vertical one (the default orientation).
<code>style</code>	The style to be used in rendering this scrollbar; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147).
<code>takefocus</code>	By default, a <code>ttk.Scrollbar</code> will not be included in focus traversal; see Section 53, “Focus: routing keyboard input” (p. 155). To add the widget to focus traversal, use <code>takefocus=True</code> .

These options of a `Tkinter` `Scrollbar` widget are *not* supported by the `ttk.Scrollbar` constructor:

Table 60. Tkinter options not in `ttk.Scrollbar`

<code>activebackground</code>	Use a style map to control the <code>background</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>activerelief</code>	Use a style map to control the <code>relief</code> option; see Section 50.2, “ <code>ttk</code> style maps: dynamic appearance changes” (p. 151).
<code>background</code> or <code>bg</code>	Configure the <code>background</code> option using a style; this option controls the color of the slider. The <code>bg</code> abbreviation is not supported.
<code>borderwidth</code> or <code>bd</code>	Configure the <code>borderwidth</code> option using a style. The <code>bd</code> abbreviation is not supported.
<code>elementborderwidth</code>	Not supported.
<code>highlightbackground</code>	Not supported.
<code>highlightcolor</code>	Not supported.
<code>highlightthickness</code>	Not supported.
<code>jump</code>	Not supported.
<code>relief</code>	Configure this option using a style.
<code>repeatdelay</code>	Not supported.
<code>repeatinterval</code>	Not supported.
<code>troughcolor</code>	Configure this option using a style.
<code>width</code>	Configure this option using a style. You may find that configuring <code>arrowsize</code> is a better choice; in some themes, increasing the <code>width</code> may not increase the size of the arrowheads.

Methods on a `ttk.Scrollbar` include all those described in Section 46, “Methods common to all `ttk` widgets” (p. 145), plus:

.delta(*dx*, *dy*)

Given a mouse movement of (*dx*, *dy*) in pixels, this method returns the `float` value that should be added to the current slider position to achieve that same movement. The value must be in the closed interval [-1.0, 1.0].

.fraction(*x*, *y*)

Given a pixel location (*x*, *y*), this method returns the corresponding normalized slider position in the interval [0.0, 1.0] that is closest to that location.

.get()

Returns two numbers (*a*, *b*) describing the current position of the slider. The *a* value gives the position of the left or top edge of the slider, for horizontal and vertical scrollbars respectively; the *b* value gives the position of the right or bottom edge. Each value is in the interval [0.0, 1.0] where 0.0 is the leftmost or top position and 1.0 is the rightmost or bottom position. For example, if the slider extends from halfway to three-quarters of the way along the trough, you might get back the tuple (0.5, 0.75).

.set(*first*, *last*)

To connect a scrollbar to another widget *w*, set *w*'s `xscrollcommand` or `yscrollcommand` to the scrollbar's `.set` method. The arguments have the same meaning as the values returned by the `.get()` method. Please note that moving the scrollbar's slider does *not* move the corresponding widget.

43. `ttk.Separator`

Use this widget to place a horizontal or vertical bar that separates other widgets. The widget is rendered as a 2-pixel wide line. Be sure to use the `sticky` options to the `.grid()` method to stretch the widget, or it will appear as only a single pixel.

To create a `ttk.Separator` as the child of a given `parent` widget, where the `option` values are given in Table 61, “`ttk.Separator` options” (p. 137):

```
w = ttk.Separator(parent, option=value, ...)
```

Table 61. `ttk.Separator` options

<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>orient</code>	Set <code>orient=tk.HORIZONTAL</code> for a horizontal separator, <code>orient=tk.VERTICAL</code> for a vertical one (the default orientation).
<code>style</code>	The style to be used in rendering this scrollbar; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147). The only style feature you can configure is <code>background</code> , which specifies the color of the separator bar; the default color is a dark gray.

The only methods available on a `ttk.Separator` widgets are the ones listed in Section 46, “Methods common to all `ttk` widgets” (p. 145).

44. `ttk.Sizegrip`

Use this widget to provide a widget that the user can use to resize the entire application window. Typically this widget will be located in the bottom right corner of the application. Be sure to make the entire application resizable using the techniques described in Section 4.3, “Configuring column and row sizes” (p. 7) and Section 4.4, “Making the root window resizable” (p. 8).

To create a `ttk.Sizegrip` as the child of a given `parent` widget, where the `option` values are given in Table 62, “`ttk.Sizegrip` options” (p. 137):

```
w = ttk.Sizegrip(parent, option=value, ...)
```

Table 62. `ttk.Sizegrip` options

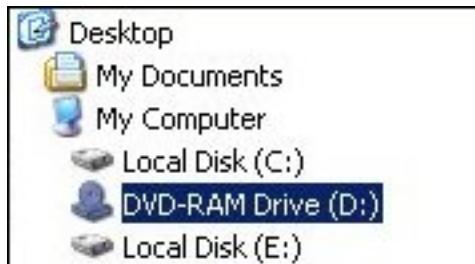
<code>class_</code>	The widget class name. This may be specified when the widget is created, but cannot be changed later. For an explanation of widget classes, see Section 27, “Standardizing appearance” (p. 105).
<code>style</code>	The style to be used in rendering this widget; see Section 49, “Using and customizing <code>ttk</code> styles” (p. 147). The only style feature you can configure is <code>background</code> , which specifies the color of the widget.

45. `ttk.Treeview`

The purpose of the `ttk.Treeview` widget is to present a hierarchical structure so that the user can use mouse actions to reveal or hide any part of the structure.

The association with the term “tree” is due to programming practice: tree structures are a commonplace in program design. Strictly speaking, the hierarchy shown in a `Treeview` widget is a forest: there is no one root, just a collection of top-level *nodes*, each of which may contain second-level nodes, each of which may contain third-level nodes, and so on.

You may have encountered this particular presentation as a way of browsing a directory or folder hierarchy. The entire hierarchy is displayed like an indented outline, where each directory is on a separate line, and the subdirectories of each directory are displayed underneath that line, indented:



The user can click on the icon for a directory to *collapse* (close) it, hiding all of the items in it. They can also click again on the icon to *expand* (open) it, so that the items in the directory or folder are shown.

The `Treeview` widget generalizes this concept so that you can use it to display any hierarchical structure, and the reader can collapse or expand subtrees of this structure with the mouse.

First, some definitions:

item

One of the entities being displayed in the widget. For a file browser, an item might be either a directory or a file.

Each item is associated with a textual label, and may also be associated with an image.

iid

Every item in the tree has a unique identifier string called the *iid*. You can supply the iid values yourself, or you can let *ttk* generate them.

child

The items directly below a given item in a hierarchy. A directory, for example, may have two kinds of children: files and subdirectories.

parent

For a given item, if it is at the top of the hierarchy it is said to have no parent; if it is not at the top level, the parent is the item that contains it.

ancestor

The ancestors of an item include its parent, its parent's parent, and so on up to the top level of the tree.

visible

Top-level items are always visible. Otherwise, an item is visible only if all its ancestors are expanded.

descendant

The descendants of an item include its children, its childrens' children, and so on. Another way of saying this is that the subtree of an item includes all its descendants.

tag

Your program can associate one or more *tag* strings with each item. You can use these tags to control the appearance of an item. For example, you could tag directories with the tag '*d*' and files with the tag '*f*', and then specify that items with tag '*d*' use a boldface font.

You may also associate events with tags, so that certain events will cause certain handlers to be called for all items that have that tag. For example, you could set up a file browser so that when a user clicks on a directory, the browser updated its contents to reflect the current file structure.

Your `Treeview` widget will be structured with multiple columns. The first column, which we'll call the *icon column*, displays the icons that collapse or expand items. In the remaining columns, you may display whatever information you like.

For example, a simple file browser widget might use two columns, with the directory icons in the first column and the directory or file name in the second columns. Or you might wish to display file sizes, permissions, and other related data in additional columns.

The operations of the `Treeview` widget even allow you to use it as a tree editor. Your program can remove an entire subtree from its location in the main tree and then attach it later at an entirely different point.

Here is the general procedure for setting up a `Treeview` widget.

1. Create the widget with the `ttk.Treeview` constructor. Use the `columns` keyword argument to specify the number of columns to be displayed and to assign symbolic names to each column.
2. Use the `.column()` and `.heading()` methods to set up column headings (if you want them) and configure column properties such as size and stretchability.
3. Starting with the top-level entries, use the `.insert()` method to populate the tree. Each call to this method adds one item to the tree. Use the `open` keyword argument of this method to specify whether the item is initially expanded or collapsed.

If you want to supply the `iid` value for this item, use the `iid` keyword argument. If you omit this argument, `ttk` will make one up and return it as the result of the `.insert()` method call.

Use the `values` keyword argument of this method to specify what should appear in each column of this item when it is visible.

To create a `Treeview` widget within a given `parent` widget:

```
w = ttk.Treeview(parent, option=value, ...)
```

The constructor returns the new `Treeview` widget. Its options include:

<code>class_</code>	You may provide a widget class name when you create this widget. This name may be used to customize the widget's appearance; see Section 27, "Standardizing appearance" (p. 105). Once the widget is created, the widget class name cannot be changed.
<code>columns</code>	A sequence of column identifier strings. These strings are used internally to identify the columns within the widget. The icon column, whose identifier is always ' <code>#0</code> ', contains the collapse/expand icons and is always the first column. The columns you specify with the <code>columns</code> argument are in addition to the icon column. For example, if you specified <code>columns=('Name', 'Size')</code> , three columns would appear in the widget: first the icon column, then two more columns whose internal identifiers are ' <code>Name</code> ' and ' <code>Size</code> '.
<code>cursor</code>	Use this option to specify the appearance of the mouse cursor when it is over the widget; see Section 5.8, "Cursors" (p. 13). The default value (an empty string) specifies that the cursor is inherited from the parent widget.

<code>displaycolumns</code>	Selects which columns are actually displayed and determines the order of their presentation. Values may be: <ul style="list-style-type: none"> '<code>#all</code>' to select all columns and display them in the order defined by the <code>columns</code> argument. A list of column numbers (integer positions, counting from 0) or column identifiers from the <code>columns</code> argument. For example, suppose you specify <code>columns=('Name', 'Size', 'Date')</code> . This means each call to the <code>.insert()</code> method will require an argument <code>values=(name, size, date)</code> to supply the values that will be displayed. Let's call this sequence the <i>logical column sequence</i> . Further suppose that in the constructor you specify <code>columns=(2, 0)</code> . The <i>physical column sequence</i> , the columns that will actually appear in the widget, will be three: the icon column will be first, followed by the date column (index 2 in the logical column sequence), followed by the name column (logical column index 0). The size column will not appear. You could get the same effect by specifying column identifiers instead of logical column positions: <code>columns=('Date', 'Name')</code> .																									
<code>height</code>	The desired height of the widget, in rows.																									
<code>padding</code>	Use this argument to place extra space around the contents inside the widget. You may provide either a single dimension or a sequence of up to four dimensions, interpreted according to this table: <table border="1"> <thead> <tr> <th>Values given</th><th>Left</th><th>Top</th><th>Right</th><th>Bottom</th></tr> </thead> <tbody> <tr> <td><code>a</code></td><td><code>a</code></td><td><code>a</code></td><td><code>a</code></td><td><code>a</code></td></tr> <tr> <td><code>a b</code></td><td><code>a</code></td><td><code>b</code></td><td><code>a</code></td><td><code>b</code></td></tr> <tr> <td><code>a b c</code></td><td><code>a</code></td><td><code>c</code></td><td><code>b</code></td><td><code>c</code></td></tr> <tr> <td><code>a b c d</code></td><td><code>a</code></td><td><code>b</code></td><td><code>c</code></td><td><code>d</code></td></tr> </tbody> </table>	Values given	Left	Top	Right	Bottom	<code>a</code>	<code>a</code>	<code>a</code>	<code>a</code>	<code>a</code>	<code>a b</code>	<code>a</code>	<code>b</code>	<code>a</code>	<code>b</code>	<code>a b c</code>	<code>a</code>	<code>c</code>	<code>b</code>	<code>c</code>	<code>a b c d</code>	<code>a</code>	<code>b</code>	<code>c</code>	<code>d</code>
Values given	Left	Top	Right	Bottom																						
<code>a</code>	<code>a</code>	<code>a</code>	<code>a</code>	<code>a</code>																						
<code>a b</code>	<code>a</code>	<code>b</code>	<code>a</code>	<code>b</code>																						
<code>a b c</code>	<code>a</code>	<code>c</code>	<code>b</code>	<code>c</code>																						
<code>a b c d</code>	<code>a</code>	<code>b</code>	<code>c</code>	<code>d</code>																						
<code>selectmode</code>	This option controls what the user is allowed to select with the mouse. Values can be: <table border="1"> <tr> <td><code>selectmode='browse'</code></td><td>The user may select only one item at a time.</td></tr> <tr> <td><code>selectmode='extended'</code></td><td>The user may select multiple items at once.</td></tr> <tr> <td><code>selectmode='none'</code></td><td>The user cannot select items with the mouse.</td></tr> </table>	<code>selectmode='browse'</code>	The user may select only one item at a time.	<code>selectmode='extended'</code>	The user may select multiple items at once.	<code>selectmode='none'</code>	The user cannot select items with the mouse.																			
<code>selectmode='browse'</code>	The user may select only one item at a time.																									
<code>selectmode='extended'</code>	The user may select multiple items at once.																									
<code>selectmode='none'</code>	The user cannot select items with the mouse.																									
<code>show</code>	To suppress the labels at the top of each column, specify <code>show='tree'</code> . The default is to show the column labels.																									
<code>style</code>	Use this option to specify a custom widget style name; see Section 47, "Customizing and creating <code>ttk</code> themes and styles" (p. 146).																									
<code>takefocus</code>	Use this option to specify whether a widget is visited during focus traversal; see Section 53, "Focus: routing keyboard input" (p. 155). Specify <code>takefocus=True</code> if you want the visit to accept focus; specify <code>takefocus=False</code> if the widget is not to accept focus. The default value is an empty string; by default, <code>ttk.Treeview</code> widgets do get focus.																									

Here are the methods available on a `Treeview` widget.

.bbox(*item*, *column=None*)

For the item with iid *item*, if the item is currently visible, this method returns a tuple (*x*, *y*, *w*, *h*), where (*x*, *y*) are the coordinates of the upper left corner of that item relative to the widget, and *w* and *h* are the width and height of the item in pixels. If the item is not visible, the method returns an empty string.

If the optional *column* argument is omitted, you get the bounding box of the entire row. To get the bounding box of one specific column of the item's row, use *column=C* where *C* is either the integer index of the column or its column identifier.

.column(*cid*, *option=None*, *kw*)**

This method configures the appearance of the logical column specified by *cid*, which may be either a column index or a column identifier. To configure the icon column, use a *cid* value of '#0'.

Each column in a Treeview widget has its own set of options from this table:

anchor	The anchor that specifies where to position the content of the column. The default value is 'w'.
id	The column name. This option is read-only and set when the constructor is called.
minwidth	Minimum width of the column in pixels; the default value is 20.
stretch	If this option is True , the column's width will be adjusted when the widget is resized. The default setting is 1.
width	Initial width of the column in pixels; the default is 200.

- If no *option* value or any other keyword argument is supplied, the method returns a dictionary of the column options for the specified column.
- To interrogate the current value of an option named *X*, use an argument *option=X*.
- To set one or more column options, you may pass keyword arguments using the option names shown above, e.g., *anchor=tk.CENTER* to center the column contents.

.delete(items*)**

The arguments are iid values. All the items in the widget that have matching iid values are destroyed, along with all their descendants.

.detach(items*)**

The arguments are iid values. All the items in the widget that have matching iid values are removed from the visible widget, along with all their descendants.

The items are not destroyed. You may reattach them to the visible tree using the *.move()* method described below.

.exists(*iid*)

Returns **True** if there exists an item in the widget with the given *iid*, or **False** otherwise. If an item is not currently visible because it was removed with the *.detach()* method, it is still considered to exist for the purposes of the *.exists()* method.

.focus([*iid*])

If you don't provide an argument to this method, you get back either the iid of the item that currently has focus, or '' if no item has focus.

You can give focus to an item by passing its iid as the argument to this method.

.get_children([*item*])

Returns a tuple of the iid values of the children of the item specified by the *item* argument. If the argument is omitted, you get a tuple containing the iid values of the top-level items.

.heading(*cid*, *option=None*, *kw*)**

Use this method to configure the column heading that appears at the top of the widget for the column specified by *cid*, which may be either a column index or a column identifier. Use a *cid* argument value of '#0' to configure the heading over the icon column.

Each heading has its own set of options with these names and values:

anchor	An anchor that specifies how the heading is aligned within the column; see Section 5.5, "Anchors" (p. 12). The default value is <code>tk.W</code> .
command	A procedure to be called when the user clicks on this column heading.
image	To present a graphic in the column heading (either with or instead of a text heading), set this option to an image, as specified in Section 5.9, "Images" (p. 14).
text	The text that you want to appear in the column heading.

- If you supply no keyword arguments, the method will return a dictionary showing the current settings of the column heading options.
- To interrogate the current value of some heading option *X*, use an argument of the form `option=X`; the method will return the current value of that option.
- You can set one or more heading options by supplying them as keyword arguments such as "`anchor=tk.CENTER`".

.identify_column(*x*)

Given an *x* coordinate, this method returns a string of the form '#*n*' that identifies the column that contains that *x* coordinate.

Assuming that the icon column is displayed, the value of *n* is 0 for the icon column; 1 for the second physical column; 2 for the third physical column; and so on. Recall that the physical column number may be different from the logical column number in cases where you have rearranged them using the `displaycolumns` argument to the `Treeview` constructor.

If the icon column is not displayed, the value of *n* is 1 for the first physical column, 2 for the second, and so on.

.identify_element(*x*, *y*)

Returns the name of the element at location (*x*, *y*) relative to the widget, or '' if no element appears at that position. Element names are discussed in Section 50, "The `ttk` element layer" (p. 149).

.identify_region(*x*, *y*)

Given the coordinates of a point relative to the widget, this method returns a string indicating what part of the widget contains that point. Return values may include:

'nothing'	The point is not within a functional part of the widget.
'heading'	The point is within one of the column headings.
'separator'	The point is located within the column headings row, but on the separator between columns. Use the <code>.identify_column()</code> method to determine which column is located just to the left of this separator.
'tree'	The point is located within the icon column.
'cell'	The point is located within an item row but not within the icon column.

.identify_row(*y*)

If *y*-coordinate *y* is within one of the items, this method returns the iid of that item. If that vertical coordinate is not within an item, this method returns an empty string.

.index(*iid*)

This method returns the index of the item with the specified *iid* relative to its parent, counting from zero.

.set_children(*item*, **newChildren*)

Use this method to change the set of children of the item whose iid is *item*. The *newChildren* argument is a sequence of iid strings. Any current children of *item* that are not in *newChildren* are removed.

.insert(*parent*, *index*, *iid=None*, *kw*)**

This method adds a new item to the tree, and returns the item's iid value. Arguments:

<i>parent</i>	To insert a new top-level item, make this argument an empty string. To insert a new item as a child of an existing item, make this argument the parent item's iid.
<i>index</i>	This argument specifies the position among this parent's children where you want the new item to be added. For example, to insert the item as the new first child, use a value of zero; to insert it after the parent's first child, use a value of 1; and so on. To add the new item as the last child of the parent, make this argument's value 'end'.
<i>iid</i>	You may supply an iid for the item as a string value. If you don't supply an iid, one will be generated automatically and returned by the method.

You may also specify a number of item options as keyword arguments to this method.

<i>image</i>	You may display an image just to the right of the icon for this item's row by providing an <i>image=I</i> argument, where <i>I</i> is an image as specified in Section 5.9, "Images" (p. 14).
<i>open</i>	This option specifies whether this item will be open initially. If you supply <i>open=False</i> , this item will be closed. If you supply <i>open=True</i> , the item's children will be visible whenever the item itself is visible. The default value is <i>False</i> .
<i>tags</i>	You may supply one or more tag strings to be associated with this item. The value may be either a single string or a sequence of strings.
<i>text</i>	You may supply text to be displayed within the icon column of this item. If given, this text will appear just to the right of the icon, and also to the right of the image if provided.
<i>values</i>	This argument supplies the data items to be displayed in each column of the item. The values are supplied in logical column order. If too few values are supplied, the remaining columns will be blank in this item; if too many values are supplied, the extras will be discarded.

.item(*iid*[, *option*[, *kw*]])**

Use this method to set or retrieve the options within the item specified by *iid*. Refer to the `.insert()` method above for the names of the item options.

With no arguments, it returns a dictionary whose keys are the option names and the corresponding values are the settings of those options. To retrieve the value of a given option, pass the option's name as its second argument. To set one or more options, pass them as keyword arguments to the method.

.move(*iid*, *parent*, *index*)

Move the item specified by *iid* to the values under the item specified by *parent* at position *index*. The *parent* and *index* arguments work the same as those arguments to the `.index()` method.

.next(*iid*)

If the item specified by *iid* is not the last child of its parent, this method returns the iid of the following child; if it is the last child of its parent, this method returns an empty string. If the specified

item is a top-level item, the method returns the iid of the next top-level item, or an empty string if the specified item is the last top-level item.

.parent(*iid*)

If the item specified by *iid* is a top-level item, this method returns an empty string; otherwise it returns the iid of that item's parent.

.prev(*iid*)

If the item specified by *iid* is not the first child of its parent, this method returns the iid of the previous child; otherwise it returns an empty string. If the specified item is a top-level item, this method returns the iid of the previous top-level item, or an empty string if it is the first top-level item.

.see(*iid*)

This method ensures that the item specified by *iid* is visible. Any of its ancestors that are closed are opened. The widget is scrolled, if necessary, so that the item appears.

.selection_add(*items*)

In addition to any items already selected, add the specified *items*. The argument may be either a single iid or a sequence of iids.

.selection_remove(*items*)

Unselect any items specified by the argument, which may be a single iid or a sequence of iids.

.selection_set(*items*)

Only the specified *items* will be selected; if any other items were selected before, they will become unselected.

.selection_toggle(*items*)

The argument may be a single iid or a sequence of iids. For each item specified by the argument, if it was selected, unselect it; if it was unselected, select it.

.set(*iid*, *column=None*, *value=None*)

Use this method to retrieve or set the column values of the item specified by *iid*. With one argument, the method returns a dictionary: the keys are the column identifiers, and each related value is the text in the corresponding column.

With two arguments, the method returns the data value from the column of the selected item whose column identifier is the *column* argument. With three arguments, the item's value for the specified column is set to the third argument.

.tag_bind(*tagName*, *sequence=None*, *callback=None*)

This method binds the event handler specified by the *callback* argument to all items that have tag *tagName*. The *sequence* and *callback* arguments work the same as the *sequence* and *func* arguments of the *.bind()* method described in Section 26, “Universal widget methods” (p. 97).

.tag_configure(*tagName*, *option=None*, *kw*)**

This method can either interrogate or set options that affect the appearance of all the items that have tag *tagName*. Tag options include:

'background'	The background color.
'font'	The text font.
'foreground'	The foreground color.
'image'	An image to be displayed in items with the given tag.

When called with one argument, it returns a dictionary of the current tag options. To return the value of a specific option *X*, use *X* as the second argument.

To set one or more options, use keyword arguments such as `foreground='red'`.

.tag_has(tagName[, iid])

Called with one argument, this method returns a list of the iid values for all items that carry tag `tagName`. If you provide an iid as the second argument, the method returns `True` if the item with that iid has tag `tagName`, `False` otherwise.

.xview(*args)

This is the usual method for connecting a horizontal scrollbar to a scrollable widget. For details, see Section 22.1, “The Scrollbar `command` callback” (p. 77).

.yview(*args)

This is the usual method for connecting a vertical scrollbar to a scrollable widget. For details, see Section 22.1, “The Scrollbar `command` callback” (p. 77).

45.1. Virtual events for the `ttk.Treeview` widget

Certain state changes within a `Treeview` widget generate virtual events that you can use to respond to these changes; see Section 54.8, “Virtual events” (p. 165).

- Whenever there is a change in the selection, either by items becoming selected or becoming unselected, the widget generates a “`<<TreeviewSelect>>`” event.
- Whenever an item is opened, the widget generates a “`<<TreeviewOpen>>`” event.
- Whenever an item is closed, the widget generates a “`<<TreeviewClose>>`” event.

46. Methods common to all `ttk` widgets

The methods shown here are available on all the `ttk` widgets.

.cget(option)

This method returns the value for the specified `option`.

.configure(option=value, ...)

To set one or more widget options, use keyword arguments of the form `option=value`. For example, to set a widget's `font`, you might use an argument such as “`font=('serif', 12)`”.

If you provide no arguments, the method will return a dictionary of all the widget's current option values. In this dictionary, the keys will be the option names, and each related value will be a tuple `(name, dbName, dbClass, default, current)`:

<code>name</code>	The option name.
<code>dbName</code>	The database name of the option.
<code>dbClass</code>	The database class of the option.
<code>default</code>	The default value of the option.
<code>current</code>	The current value of the option.

.identify(x, y)

Use this to determine what element is at a given location within the widget. If the point `(x, y)` relative to the widget is somewhere within the widget, this method returns the name of the element at that position; otherwise it returns an empty string.

.instate(stateSpec, callback=None, *args, **kw)

The purpose of this to determine whether the widget is in a specified state or combination of states.

If you provide a callable value as the `callback` argument, and the widget matches the state or combination of states specified by the `stateSpec` argument, that callable will be called with positional arguments `*args` and keyword arguments `**kw`. If the widget's state does not match `stateSpec`, the `callback` will not be called.

If you don't provide a `callback` argument, the method will return `True` if the widget's state matches `stateSpec`, `False` otherwise.

For the structure of the `stateSpec` argument, see Section 46.1, “Specifying widget states in `ttk`” (p. 146).

.state(stateSpec=None)

Use this item either to query a widget to determine its current states, or to set or clear one state.

If you provide a `stateSpec` argument of the form described in Section 46.1, “Specifying widget states in `ttk`” (p. 146), the method will set or clear states in the widget according to that argument.

For example, for a widget `w`, the method `w.state(['!disabled', 'selected'])` would clear the widget's '`disabled`' set and set its '`selected`' state.

46.1. Specifying widget states in `ttk`

Several methods within `ttk` require a `stateSpec` argument that specifies a particular widget state or combination of states. This argument may be any of the following:

- A single state name such as '`pressed`'. A `ttk.Button` widget is in this state, for example, when the mouse cursor is over the button and mouse button 1 is down.
- A single state name preceded with an exclamation point (!); this matches the widget state only when that state is `off`.

For example, a `stateSpec` argument '`!pressed`' specifies a widget that is not currently being pressed.

- A sequence of state names, or state names preceded by an '!'. Such a `stateSpec` matches only when all of its components match. For example, a `stateSpec` value of (`'!disabled', 'focus'`) matches a widget only when that widget is not disabled and it has focus.

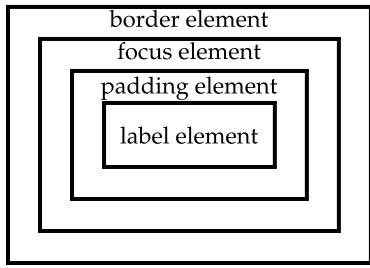
47. Customizing and creating `ttk` themes and styles

Designing with `ttk` widgets involves three levels of abstraction:

- A *theme* is a complete “look and feel,” customizing the appearance of all the widgets.
- A *style* is the description of the appearance of one kind of widget. Each theme comes with a predefined set of styles, but you can customize the built-in styles or create your own new styles.

The phrase “kind of widget” in the previous paragraph technically refers to the “class” of a widget. However, in the `ttk` world, this is different from Python classes. Within `ttk`, the class of a widget is a character string. For example, the `ttk` class of a stock `Button` widget is the string '`TButton`'.

- Each style is composed of one or more *elements*. For example, the style of a typical button has four elements: a border around the outside; a focus element that changes color when the widget has input focus; a padding element; and the button's label (text, image, or both).



We will discuss the discovery, use and customization of each of these layers in separate sections.

- Section 48, “Finding and using *ttk* themes” (p. 147).
- Section 49, “Using and customizing *ttk* styles” (p. 147).
- Section 50, “The *ttk* element layer” (p. 149).

48. Finding and using *ttk* themes

A number of operations related to themes require that you have available an instance of the `ttk.Style()` class (in the Python sense of class). For example, to obtain a list of the available themes in your installation:

```
>>> import ttk
>>> s=ttk.Style()
>>> s.theme_names()
('clam', 'alt', 'default', 'classic')
```

The `.theme_names()` method returns a tuple containing the names of the available styles. The ‘classic’ theme gives you the original, pre-*ttk* appearance.

To determine which theme you get by default, use the `.theme_use()` method with no arguments. To change the current theme, call this same method with the desired theme name as the argument:

```
>>> s.theme_use()
'default'
>>> s.theme_use('alt')
>>> s.theme_use()
'alt'
```

49. Using and customizing *ttk* styles

Within a given theme, every widget has a default *widget class*; we use this term to distinguish *ttk* classes from Python classes.

Each widget also has a *style*. The default style for a widget is determined by its *widget class*, but you may specify a different style.

In *ttk*, *widget classes* and *styles* are specified as strings. In all but one case, the default style name of a widget is ‘T’ prefixed to the widget name; for example, the default button *widget class* is ‘TButton’. There are some exceptions:

Table 63. Style names for *ttk* widget classes

Widget class	Style name
Button	TButton
Checkbutton	TCheckbutton
Combobox	TCombobox
Entry	TEntry
Frame	TFrame
Label	TLabel
LabelFrame	TLabelFrame
Menubutton	TMenubutton
Notebook	TNotebook
PanedWindow	TPanedwindow (<i>not</i> TPanedWindow!)
Progressbar	Horizontal.TProgressbar or Vertical.TProgressbar, depending on the orient option.
Radiobutton	TRadiobutton
Scale	Horizontal.TScale or Vertical.TScale, depending on the orient option.
Scrollbar	Horizontal.TScrollbar or Vertical.TScrollbar, depending on the orient option.
Separator	TSeparator
Sizegrip	TSizegrip
Treeview	Treeview (<i>not</i> TTreview!)

At runtime, you can retrieve a widget's widget class by calling its `.winfo_class()` method.

```
>>> b=ttk.Button(None)
>>> b.winfo_class()
'TButton'
>>> t=ttk.Treeview(None)
>>> t.winfo_class()
'Treeview'
>>> b.__class__      # Here, we are asking for the Python class
<class 'ttk.Button' at 0x21c76d0>
```

The name of a style may have one of two forms.

- The built-in styles are all a single word: 'TFrame' or 'TRadiobutton', for example.
- To create a new style derived from one of the built-in styles, use a style name of the form '*newName.oldName*'. For example, to create a new style of Entry widget to hold a date, you might call it 'Date.TEntry'.

Every style has a corresponding set of *options* that define its appearance. For example, buttons have a **foreground** option that changes the color of the button's text.

To change the appearance of a style, use its `.configure()` method. The first argument of this method is the name of the style you want to configure, followed by keyword arguments specifying the option names and values you want to change. For example, to make all your buttons use green text, where `s` is in instance of the `ttk.Style` class:

```
s.configure('TButton', foreground='green')
```

To create a new style based on some style *oldName*, first create an instance of `ttk.Style`, then call its `.configure()` method using a name of the form '*newName.oldName*'. For example, suppose you don't want to use maroon text on all your buttons, but you do want to create a new style that does use maroon text, and you want to call the new style '`Kim.TButton`':

```
s = ttk.Style()
s.configure('Kim.TButton', foreground='maroon')
```

Then to create a button in the new class you might use something like this:

```
self.b = ttk.Button(self, text='Friday', style='Kim.TButton',
                    command=self._fridayHandler)
```

You can even build entire hierarchies of styles. For example, if you configure a style named '`Panic.Kim.TButton`', that style will inherit all the options from the '`Kim.TButton`' style, that is, any option you don't set in the '`Panic.Kim.TButton`' style will be the same as that option in the '`Kim.TButton`' style.

When `ttk` determines what value to use for an option, it looks first in the '`Panic.Kim.TButton`' style; if there is no value for that option in that style, it looks in the '`Kim.TButton`' style; and if that style doesn't define the option, it looks in the '`TButton`' style.

There is a *root style* whose name is '`.`'. To change some feature's default appearance for every widget, you can configure this style. For example, let's suppose that you want all text to be 12-point Helvetica (unless overridden by another style or `font` option). This configuration would do it:

```
s = ttk.Style()
s.configure('.', font=('Helvetica', 12))
```

50. The `ttk` element layer

A `ttk` element is one of the pieces that make up a widget. In order to understand how elements are assembled into styles, read these sections.

- Section 50.1, “`ttk` layouts: Structuring a style” (p. 149): the static structure of elements within a widget.
- Section 50.2, “`ttk` style maps: dynamic appearance changes” (p. 151): states of a widget and how those states affect its appearance.

50.1. `ttk` layouts: Structuring a style

In general, the pieces of a widget are assembled using the idea of a *cavity*, an empty space that is to be filled with elements.

For example, in the `classic` theme, a button has four concentric elements. From the outside in, they are the *focus highlight*, *border*, *padding*, and *label* elements.

Each of these elements has a '`sticky`' attribute that specifies how many of the four sides of the cavity it “sticks” to. For example, if an element has a `sticky='ew'` attribute, that means it must stretch in order to stick to the left (west) and right (east) sides of its cavity, but it does not have to stretch vertically.

Most of the built-in `ttk` styles use the idea of a *layout* to organize the different layers that make up a widget. Assuming that `S` is an instance of `ttk.Style`, to retrieve that style's layout use a method call of this form, where `widgetClass` is the name of the widget class.

`S.layout(widgetClass)`

Some widget classes don't have a layout; in those cases, this method call will raise a `tk.TclError` exception.

For the widget classes that have a layout, the returned value is a list of tuples (`eltName, d`). Within each tuple, `eltName` is the name of an element and `d` is a dictionary that describes the element.

This dictionary may have values for the following keys:

`'sticky'`

A string that defines how this element is to be positioned within its parent. This string may contain zero or more of the characters '`n`', '`s`', '`e`', and '`w`', referring to the sides of the box with the same conventions as for anchors. For example, the value `sticky='nswe'` would stretch this element to adhere to the north, south, and west sides of the cavity within its parent element.

`'side'`

For elements with multiple children, this value defines how the element's children will be positioned inside it. Values may be `'left'`, `'right'`, `'top'`, or `'bottom'`.

`'children'`

If there are elements inside this element, this entry in the dictionary is the layout of the child elements using the same format as the top-level layout, that is, a list of two-element tuples (`eltName, d`).

Let's dissect the layout of the stock `Button` widget of the `'classic'` theme in this conversational example.

```
>>> import ttk
>>> s = ttk.Style()
>>> s.theme_use('classic')
>>> b = ttk.Button(None, text='Yo')
>>> bClass = b.winfo_class()
>>> bClass
'TButton'
>>> layout = s.layout('TButton')
>>> layout
[(['Button.highlight', {'children': [(['Button.border', {'border': '1', 'children': [(['Button.padding', {'children': [(['Button.label', {'sticky': 'nswe'}], 'sticky': 'nswe'}], 'sticky': 'nswe'}], 'sticky': 'nswe'}]}]]]
```

All those parentheses, brackets, and braces make that structure a bit hard to understand. Here it is in outline form:

- The outermost element is the focus highlight; it has style `'Button.highlight'`. Its `'sticky'` attribute is `'nswe'`, meaning it should expand in all four directions to fill its cavity.
- The only child of the focus highlight is the border element, with style `'Button.border'`. It has a `'border'` width of 1 pixel, and its `'sticky'` attribute also specifies that it adheres to all four sides of its cavity, which is defined by the inside of the highlight element.
- Inside the border is a layer of padding, with style `'Button.padding'`. Its `sticky` attribute also specifies that it fills its cavity.
- Inside the padding layer is the text (or image, or both) that appears on the button. Its style is `'Button.label'`, with the usual `sticky='nswe'` attribute.

Each element has a dictionary of *element options* that affect the appearance of that element. The names of these options are all regular `Tkinter` options such as `'anchor'`, `'justify'`, `'background'`, or `'highlightthickness'`.

To obtain the list of option names, use a method call of this form, where `S` is an instance of class `ttk.Style`:

```
S.element_options(styleName)
```

The result is a sequence of option strings, each preceded by a hyphen. Continuing our conversational above, where `S` is an instance of `ttk.Style`:

```
>>> d = S.element_options('Button.highlight')
>>> d
('-highlightcolor', '-highlightthickness')
```

To find out what attributes are associated with an element option, use a method call of this form:

```
s.lookup(layoutName, optName)
```

Continuing our example:

```
>>> s.lookup('Button.highlight', 'highlightthickness')
1
>>> s.lookup('Button.highlight', 'highlightcolor')
'#d9d9d9'
>>> print s.element_options('Button.label')
('-compound', '-space', '-text', '-font', '-foreground', '-underline',
'-width', '-anchor', '-justify', '-wraplength', '-embossed', '-image',
'-stipple', '-background')
>>> s.lookup('Button.label', 'foreground')
'black'
```

50.2. `ttk` style maps: dynamic appearance changes

The `ttk` widgets can change their appearance during the execution of the program. For example, when a widget is *disabled*, it will not respond to mouse or keyboard actions. Typically a disabled widget presents a different appearance so that the user might realize that the widget will not respond to the mouse.

In general, every `ttk` widget has a set of *state flags* that you can use to make the appearance of a widget change during execution. Each state may be set (turned on) or reset (turned off) independently of the other states. The states and their meanings:

<code>active</code>	The mouse is currently within the widget.
<code>alternate</code>	This state is reserved for application use.
<code>background</code>	Under Windows or MacOS, the widget is located in a window that is not the foreground window.
<code>disabled</code>	The widget will not respond to user actions.
<code>focus</code>	The widget currently has focus.
<code>invalid</code>	The contents of the widget are not currently valid.
<code>pressed</code>	The widget is currently being pressed (e.g., a button that is being clicked).
<code>readonly</code>	The widget will not allow any user actions to change its current value. For example, a read-only <code>Entry</code> widget will not allow editing of its content.

selected	The widget is selected. Examples are checkbuttons and radiobuttons that are in the “on” state.
-----------------	--

Some states will change in response to user actions, for example, the **pressed** state of a **Button**. Your program can interrogate, clear, or set any state by using functions described in Section 46, “Methods common to all *ttk* widgets” (p. 145).

The logic that changes the appearance of a widget is tied to one of its elements. To interrogate or set up dynamic behavior for a specific style, given an instance *s* of *ttk.Style*, use this method, where *styleName* is the element’s name, e.g., ‘*Button.label*’ or ‘*border*’.

```
s.map(styleName, *p, **kw)
```

To determine the dynamic behavior of one option of a given style element, pass the option name as the second positional argument, and the method will return a list of state change specifications.

Each state change specification is a sequence (s_0, s_1, n). This sequence means that when the widget’s current state matches all the s_i parts, set the option to the value n . Each item s_i is either a state name, or a state name preceded by a “!”. To match, the widget must be in all the states described by items that don’t start with “!”, and it must *not* be in any of the states that start with “!”.

For example, suppose you have an instance *s* of class *ttk.Style*, and you call it like this:

```
changes = s.map('TCheckbutton', 'indicatorcolor')
```

Further suppose that the return value is:

```
[('pressed', '#ececce'), ('selected', '#4a6984')]
```

This means that when a checkbutton is in the **pressed** state, its **indicatorcolor** option should be set to the color ‘#ececce’, and when the checkbutton is in the **selected** state, its **indicatorcolor** option should be set to ‘#4a6984’.

You may also change the dynamic behavior of an element by passing one or more keyword arguments to the **.map()** method. For example, to get the behavior of the above example, use this method call:

```
s.map('TCheckbutton',
      indicatoron=[('pressed', '#ececce'), ('selected', '#4a6984')])
```

Here’s a more complex example. Suppose you want to create a custom button style based on the standard **TButton** class. We’ll name our style **Wild.TButton**; because our name ends with “.TButton”, it automatically inherits the standard style features. Here’s how to set up this new style:

```
s = ttk.Style()
s.configure('Wild.TButton',
            background='black',
            foreground='white',
            highlightthickness=20,
            font=('Helvetica', 18, 'bold'))
s.map('Wild.TButton',
      foreground=[('disabled', 'yellow'),
                  ('pressed', 'red'),
                  ('active', 'blue')],
      background=[('disabled', 'magenta'),
                  ('pressed', '!focus', 'cyan'),
                  ('active', 'green')],
      highlightcolor=[('focus', 'green')],
```

```
( '!focus', 'red')],  
relief=[('pressed', 'groove'),  
       ('!pressed', 'ridge')])
```

- This button will initially show white text on a black background, with a 20-pixel-wide focus highlight.
- If the button is in '`disabled`' state, it will show yellow text on a magenta background.
- If the button is currently being pressed, the text will be red; provided the button does *not* have focus, the background will be cyan. The tuple `('pressed', '!focus', 'cyan')` is an example of how you can make an attribute dependent on a combination of states.
- If the button is active (under the cursor), the text will be blue on a green background.
- The focus highlight will be green when the button has focus and red when it does not.
- The button will show ridge relief when it is not being pressed, and groove relief when it is being pressed.

51. Connecting your application logic to the widgets

The preceding sections talked about how to arrange and configure the widgets—the front panel of the application.

Next, we'll talk about how to connect up the widgets to the logic that carries out the actions that the user requests.

- To make your application respond to events such as mouse clicks or keyboard inputs, there are two methods:
 - Some controls such as buttons have a `command` attribute that lets you specify a procedure, called a *handler*, that will be called whenever the user clicks that control.

The sequence of events for using a `Button` widget is very specific, though. The user must move the mouse pointer onto the widget with mouse button 1 up, then press mouse button 1, and then release mouse button 1 while still on the widget. No other sequence of events will “press” a `Button` widget.

- There is a much more general mechanism that can let your application react to many more kinds of inputs: the press or release of any keyboard key or mouse button; movement of the mouse into, around, or out of a widget; and many other events. As with `command` handlers, in this mechanism you write handler procedures that will be called whenever certain types of events occur. This mechanism is discussed under Section 54, “Events” (p. 157).
- Many widgets require you to use *control variables*, special objects that connect widgets together and to your program, so that you can read and set properties of the widgets. Control variables will be discussed in the next section.

52. Control variables: the values behind the widgets

A *Tkinter control variable* is a special object that acts like a regular Python variable in that it is a container for a value, such as a number or string.

One special quality of a control variable is that it can be shared by a number of different widgets, and the control variable can remember all the widgets that are currently sharing it. This means, in particular, that if your program stores a value *v* into a control variable *c* with its *c.set(v)* method, any widgets that are linked to that control variable are automatically updated on the screen.

Tkinter uses control variables for a number of important functions, for example:

- Checkbuttons use a control variable to hold the current state of the checkbutton (on or off).
- A single control variable is shared by a group of radiobuttons and can be used to tell which one of them is currently set. When the user clicks on one radiobutton in a group, the sharing of this control variable is the mechanism by which *Tkinter* groups radiobuttons so that when you set one, any other set radiobutton in the group is cleared.
- Control variables hold text string for several applications. Normally the text displayed in an **Entry** widget is linked to a control variable. In several other controls, it is possible to use a string-valued control variable to hold text such as the labels of checkbuttons and radiobuttons and the content of **Label** widgets.

For example, you could link an **Entry** widget to a **Label** widget so that when the user changes the text in the entry and presses the *Enter* key, the label is automatically updated to show that same text.

To get a control variable, use one of these class constructors, depending on what type of values you need to store in it:

```
v = tk.DoubleVar()    # Holds a float; default value 0.0
v = tk.IntVar()       # Holds an int; default value 0
v = tk.StringVar()    # Holds a string; default value ''
```

All control variables have these two methods:

.get()

Returns the current value of the variable.

.set(value)

Changes the current value of the variable. If any widget options are slaved to this variable, those widgets will be updated when the main loop next idles; see `.update_idletasks()` in Section 26, “Universal widget methods” (p. 97) for more information on controlling this update cycle.

Here are some comments on how control variables are used with specific widgets:

Button

You can set its `textvariable` to a `StringVar`. Anytime that variable is changed, the text on the button will be updated to display the new value. This is not necessary unless the button's text is actually going to change: use the `text` attribute if the button's label is static.

Checkbutton

Normally, you will set the widget's `variable` option to an `IntVar`, and that variable will be set to 1 when the checkbutton is turned on and to 0 when it is turned off. However, you can pick different values for those two states with the `onvalue` and `offvalue` options, respectively.

You can even use a `StringVar` as the checkbutton's variable, and supply string values for the `offvalue` and `onvalue`. Here's an example:

```
self.spamVar = tk.StringVar()
self.spamCB = tk.Checkbutton(self, text='Spam?',
                            variable=self.spamVar, onvalue='yes', offvalue='no')
```

If this checkbutton is on, `self.spamVar.get()` will return the string 'yes'; if the checkbutton is off, that same call will return the string 'no'. Furthermore, your program can turn the checkbutton on by calling `.set('yes')`.

You can also the `textvariable` option of a checkbutton to a `StringVar`. Then you can change the text label on that checkbutton using the `.set()` method on that variable.

Entry

Set its `textvariable` option to a `StringVar`. Use that variable's `.get()` method to retrieve the text currently displayed in the widget. You can also use the variable's `.set()` method to change the text displayed in the widget.

Label

You can set its `textvariable` option to a `StringVar`. Then any call to the variable's `.set()` method will change the text displayed on the label. This is not necessary if the label's text is static; use the `text` attribute for labels that don't change while the application is running.

Menubutton

If you want to be able to change the text displayed on the menu button, set its `textvariable` option to a `StringVar` and use that variable's `.set()` method to change the displayed text.

Radiobutton

The `variable` option must be set to a control variable, either an `IntVar` or a `StringVar`. All the radiobuttons in a functional group must share the same control variable.

Set the `value` option of each radiobutton in the group to a different value. Whenever the user sets a radiobutton, the variable will be set to the `value` option of that radiobutton, and all the other radiobuttons that share the group will be cleared.

You might wonder, what state is a group of radiobuttons in when the control variable has never been set and the user has never clicked on them? Each control variable has a default value: `0` for an `IntVar`, `0.0` for a `DoubleVar`, and `' '` for a `StringVar`. If one of the radiobuttons has that `value`, that radiobutton will be set initially. If no radiobutton's `value` option matches the value of the variable, the radiobuttons will all appear to be cleared.

If you want to change the text label on a radiobutton during the execution of your application, set its `textvariable` option to a `StringVar`. Then your program can change the text label by passing the new label text to the variable's `.set()` method.

Scale

For a scale widget, set its `variable` option to a control variable of any class, and set its `from_` and `to` options to the limiting values for the opposite ends of the scale.

For example, you could use an `IntVar` and set the scale's `from_=0` and `to=100`. Then every user change to the widget would change the variable's value to some value between 0 and 100 inclusive.

Your program can also move the slider by using the `.set()` method on the control variable. To continue the above example, `.set(75)` would move the slider to a position three-fourths of the way along its trough.

To set up a `Scale` widget for `float` values, use a `DoubleVar`.

You can use a `StringVar` as the control variable of a `Scale` widget. You will still need to provide numeric `from_` and `to` values, but the numeric value of the widget will be converted to a string for storage in the `StringVar`. Use the scale's `digits` option to control the precision of this conversion.

53. Focus: routing keyboard input

To say a widget has *focus* means that keyboard input is currently directed to that widget.

- By *focus traversal*, we mean the sequence of widgets that will be visited as the user moves from widget to widget with the `tab` key. See below for the rules for this sequence.
- You can traverse backwards using `shift-tab`.

- The `Entry` and `Text` widgets are intended to accept keyboard input, and if an entry or text widget currently has the focus, any characters you type into it will be added to its text. The usual editing characters such as `←` and `→` will have their usual effects.
- Because `Text` widgets can contain tab characters, you must use the special key sequence `control-tab` to move the focus past a text widget.
- Most of the other types of widgets will normally be visited by focus traversal, and when they have focus:
 - `Button` widgets can be “pressed” by pressing the spacebar.
 - `Checkbutton` widgets can be toggled between set and cleared states using the spacebar.
 - In `Listbox` widgets, the `↑` and `↓` keys scroll up or down one line; the `PageUp` and `PageDown` keys scroll by pages; and the spacebar selects the current line, or de-selects it if it was already selected.
 - You can set a `Radiobutton` widget by pressing the spacebar.
 - Horizontal `Scale` widgets respond to the `←` and `→` keys, and vertical ones respond to `↑` and `↓`.
 - In a `Scrollbar` widget, the `PageUp` and `PageDown` keys move the scrollbar by pageloads. The `↑` and `↓` keys will move vertical scrollbars by units, and the `←` and `→` keys will move horizontal scrollbars by units.
- Many widgets are provided with an outline called the *focus highlight* that shows the user which widget has the highlight. This is normally a thin black frame located just outside the widget's border (if any). For widgets that don't normally have a focus highlight (specifically, frames, labels, and menus), you can set the `highlightthickness` option to a nonzero value to make the focus highlight visible.
- You can also change the color of the focus highlight using the `highlightcolor` option.
- Widgets of class `Frame`, `Label`, and `Menu` are not normally visited by the focus. However, you can set their `takefocus` options to `1` to get them included in focus traversal. You can also take any widget out of focus traversal by setting its `takefocus` option to `0`.

The order in which the `tab` key traverses the widgets is:

- For widgets that are children of the same parent, focus goes in the same order the widgets were created.
- For parent widgets that contain other widgets (such as frames), focus visits the parent widget first (unless its `takefocus` option is `0`), then it visits the child widgets, recursively, in the order they were created.

To sum up: to set up the focus traversal order of your widgets, create them in that order. Remove widgets from the traversal order by setting their `takefocus` options to `0`, and for those whose default `takefocus` option is `0`, set it to `1` if you want to add them to the order.

The above describes the default functioning of input focus in *Tkinter*. There is another, completely different way to handle it—let the focus go wherever the mouse goes. Under Section 26, “Universal widget methods” (p. 97), refer to the `.tk_focusFollowsMouse()` method.

You can also add, change or delete the way any key on the keyboard functions inside any widget by using event bindings. See Section 54, “Events” (p. 157) for the details.

53.1. Focus in `ttk` widgets

If you create a `ttk` widget and do not specify its `takefocus` option, by default, all `ttk` widgets get focus except for `Frame`, `Label`, `LabelFrame`, `PanedWindow`, `Progressbar`, `Scrollbar`, `Separator`, and `Sizegrip`.

54. Events: responding to stimuli

An *event* is something that happens to your application—for example, the user presses a key or clicks or drags the mouse—to which the application needs to react.

The widgets normally have a lot of built-in behaviors. For example, a button will react to a mouse click by calling its `command` callback. For another example, if you move the focus to an entry widget and press a letter, that letter gets added to the content of the widget.

However, the event binding capability of *Tkinter* allows you to add, change, or delete behaviors.

First, some definitions:

- An *event* is some occurrence that your application needs to know about.
- An *event handler* is a function in your application that gets called when an event occurs.
- We call it *binding* when your application sets up an event handler that gets called when an event happens to a widget.

54.1. Levels of binding

You can bind a handler to an event at any of three levels:

1. Instance binding: You can bind an event to one specific widget. For example, you might bind the `PageUp` key in a canvas widget to a handler that makes the canvas scroll up one page. To bind an event of a widget, call the `.bind()` method on that widget (see Section 26, “Universal widget methods” (p. 97)).

For example, suppose you have a canvas widget named `self.canv` and you want to draw an orange blob on the canvas whenever the user clicks the mouse button 2 (the middle button). To implement this behavior:

```
self.canv.bind('<Button-2>', self.__drawOrangeBlob)
```

The first argument is a *sequence descriptor* that tells *Tkinter* that whenever the middle mouse button goes down, it is to call the *event handler* named `self.__drawOrangeBlob`. (See Section 54.6, “Writing your handler: The `Event` class” (p. 162), below, for an overview of how to write handlers such as `__drawOrangeBlob()`). Note that you omit the parentheses after the handler name, so that Python will pass in a reference the handler instead of trying to call it right away.

2. Class binding: You can bind an event to all widgets of a class. For example, you might set up all `Button` widgets to respond to middle mouse button clicks by changing back and forth between English and Japanese labels. To bind an event to all widgets of a class, call the `.bind_class()` method on any widget (see Section 26, “Universal widget methods” (p. 97), above).

For example, suppose you have several canvases, and you want to set up mouse button 2 to draw an orange blob in any of them. Rather than having to call `.bind()` for every one of them, you can set them all up with one call something like this:

```
self.bind_class('Canvas', '<Button-2>',  
               self.__drawOrangeBlob)
```

3. Application binding: You can set up a binding so that a certain event calls a handler no matter what widget has the focus or is under the mouse. For example, you might bind the `PrintScrn` key to all the widgets of an application, so that it prints the screen no matter what widget gets that key. To bind

an event at the application level, call the `.bind_all()` method on any widget (see Section 26, “Universal widget methods” (p. 97)).

Here's how you might bind the `PrintScrn` key, whose “key name” is '`Print`':

```
self.bind_all('<Key-Print>', self.__printScreen)
```

54.2. Event sequences

Tkinter has a powerful and general method for allowing you to define exactly which events, both specific and general, you want to bind to handlers.

In general, an event sequence is a string containing one or more *event patterns*. Each event pattern describes one thing that can happen. If there is more than one event pattern in a sequence, the handler will be called only when all the patterns happen in that same sequence.

The general form of an event pattern is:

```
<[modifier-]... type[-detail]>
```

- The entire pattern is enclosed inside `<...>`.
- The *event type* describes the general kind of event, such as a key press or mouse click. See Section 54.3, “Event types” (p. 158).
- You can add optional *modifier* items before the type to specify combinations such as the *shift* or *control* keys being depressed during other key presses or mouse clicks. Section 54.4, “Event modifiers” (p. 160)
- You can add optional *detail* items to describe what key or mouse button you're looking for. For mouse buttons, this is 1 for button 1, 2 for button 2, or 3 for button 3.
 - The usual setup has button 1 on the left and button 3 on the right, but left-handers can swap these positions.
 - For keys on the keyboard, this is either the key's character (for single-character keys like the `A` or `*` key) or the key's name; see Section 54.5, “Key names” (p. 160) for a list of all key names.

Here are some examples to give you the flavor of event patterns:

<code><Button-1></code>	The user pressed the first mouse button.
<code><KeyPress-H></code>	The user pressed the H key.
<code><Control-Shift-KeyPress-H></code>	The user pressed <i>control-shift-H</i> .

54.3. Event types

The full set of event types is rather large, but a lot of them are not commonly used. Here are most of the ones you'll need:

Type	Name	Description
36	<code>Activate</code>	A widget is changing from being inactive to being active. This refers to changes in the <code>state</code> option of a widget such as a button changing from inactive (grayed out) to active.

Type	Name	Description
4	Button	The user pressed one of the mouse buttons. The <code>detail</code> part specifies which button. For mouse wheel support under Linux, use <code>Button-4</code> (scroll up) and <code>Button-5</code> (scroll down). Under Linux, your handler for mouse wheel bindings will distinguish between scroll-up and scroll-down by examining the <code>.num</code> field of the <code>Event</code> instance; see Section 54.6, “Writing your handler: The <code>Event</code> class” (p. 162).
5	ButtonRelease	The user let up on a mouse button. This is probably a better choice in most cases than the <code>Button</code> event, because if the user accidentally presses the button, they can move the mouse off the widget to avoid setting off the event.
22	Configure	The user changed the size of a widget, for example by dragging a corner or side of the window.
37	Deactivate	A widget is changing from being active to being inactive. This refers to changes in the <code>state</code> option of a widget such as a radiobutton changing from active to inactive (grayed out).
17	Destroy	A widget is being destroyed.
7	Enter	The user moved the mouse pointer into a visible part of a widget. (This is different than the <code>enter</code> key, which is a <code>KeyPress</code> event for a key whose name is actually ' <code>return</code> '.)
12	Expose	This event occurs whenever at least some part of your application or widget becomes visible after having been covered up by another window.
9	FocusIn	A widget got the input focus (see Section 53, “Focus: routing keyboard input” (p. 155) for a general introduction to input focus.) This can happen either in response to a user event (like using the <code>tab</code> key to move focus between widgets) or programmatically (for example, your program calls the <code>.focus_set()</code> on a widget).
10	FocusOut	The input focus was moved out of a widget. As with <code>FocusIn</code> , the user can cause this event, or your program can cause it.
2	KeyPress	The user pressed a key on the keyboard. The <code>detail</code> part specifies which key. This keyword may be abbreviated <code>Key</code> .
3	KeyRelease	The user let up on a key.
8	Leave	The user moved the mouse pointer out of a widget.
19	Map	A widget is being mapped, that is, made visible in the application. This will happen, for example, when you call the widget's <code>.grid()</code> method.
6	Motion	The user moved the mouse pointer entirely within a widget.
38	MouseWheel	The user moved the mouse wheel up or down. At present, this binding works on Windows and MacOS, but not under Linux. For Windows and MacOS, see the discussion of the <code>.delta</code> field of the <code>Event</code> instance in Section 54.6, “Writing your handler: The <code>Event</code> class” (p. 162). For Linux, see the note above under <code>Button</code> .
18	Unmap	A widget is being unmapped and is no longer visible. This happens, for example, when you use the widget's <code>.grid_remove()</code> method.
15	Visibility	Happens when at least some part of the application window becomes visible on the screen.

54.4. Event modifiers

The modifier names that you can use in event sequences include:

Alt	True when the user is holding the <i>alt</i> key down.
Any	This modifier generalizes an event type. For example, the event pattern ' <code><Any-KeyPress></code> ' applies to the pressing of any key.
Control	True when the user is holding the <i>control</i> key down.
Double	Specifies two events happening close together in time. For example, <code><Double-Button-1></code> describes two presses of button 1 in rapid succession.
Lock	True when the user has pressed <i>shift lock</i> .
Shift	True when the user is holding down the <i>shift</i> key.
Triple	Like Double, but specifies three events in rapid succession.

You can use shorter forms of the events. Here are some examples:

- '`<1>`' is the same as '`<Button-1>`'.
- '`x`' is the same as '`<KeyPress-x>`'.

Note that you can leave out the enclosing '`<...>`' for most single-character keypresses, but you can't do that for the space character (whose name is '`<space>`') or the less-than (<) character (whose name is '`<less>`').

54.5. Key names

The detail part of an event pattern for a `KeyPress` or `KeyRelease` event specifies which key you're binding. (See the `Any` modifier, above, if you want to get all keypresses or key releases).

The table below shows several different ways to name keys. See Section 54.6, “Writing your handler: The `Event` class” (p. 162), below, for more information on `Event` objects, whose attributes will describe keys in these same ways.

- The `.keysym` column shows the “key symbol”, a string name for the key. This corresponds to the `.keysym` attribute of the `Event` object.
- The `.keycode` column is the “key code.” This identifies which key was pressed, but the code does not reflect the state of various modifiers like the shift and control keys and the `NumLock` key. So, for example, both `a` and `A` have the same key code.
- The `.keysym_num` column shows a numeric code equivalent to the key symbol. Unlike `.keycode`, these codes are different for different modifiers. For example, the digit 2 on the numeric keypad (key symbol `KP_2`) and the down arrow on the numeric keypad (key symbol `KP_Down`) have the same key code (88), but different `.keysym_num` values (65433 and 65458, respectively).
- The “Key” column shows the text you will usually find on the physical key, such as `tab`.

There are many more key names for international character sets. This table shows only the “Latin-1” set for the usual USA-type 101-key keyboard. For the currently supported set, see the manual page for Tk `keysym` values¹⁰.

¹⁰ <http://www.tcl.tk/man/tcl8.4/TkCmd/keysyms.htm>

.keysym	.keycode	.keysym_num	Key
Alt_L	64	65513	The left-hand <i>alt</i> key
Alt_R	113	65514	The right-hand <i>alt</i> key
BackSpace	22	65288	<i>backspace</i>
Cancel	110	65387	<i>break</i>
Caps_Lock	66	65549	<i>CapsLock</i>
Control_L	37	65507	The left-hand <i>control</i> key
Control_R	109	65508	The right-hand <i>control</i> key
Delete	107	65535	<i>Delete</i>
Down	104	65364	↓
End	103	65367	<i>end</i>
Escape	9	65307	<i>esc</i>
Execute	111	65378	<i>SysReq</i>
F1	67	65470	Function key <i>F1</i>
F2	68	65471	Function key <i>F2</i>
F _i	66+i	65469+i	Function key <i>F_i</i>
F12	96	65481	Function key <i>F12</i>
Home	97	65360	<i>home</i>
Insert	106	65379	<i>insert</i>
Left	100	65361	←
Linefeed	54	106	Linefeed (<i>control-J</i>)
KP_0	90	65438	0 on the keypad
KP_1	87	65436	1 on the keypad
KP_2	88	65433	2 on the keypad
KP_3	89	65435	3 on the keypad
KP_4	83	65430	4 on the keypad
KP_5	84	65437	5 on the keypad
KP_6	85	65432	6 on the keypad
KP_7	79	65429	7 on the keypad
KP_8	80	65431	8 on the keypad
KP_9	81	65434	9 on the keypad
KP_Add	86	65451	+ on the keypad
KP_Begin	84	65437	The center key (same key as 5) on the keypad
KP.Decimal	91	65439	Decimal (.) on the keypad
KP_Delete	91	65439	<i>delete</i> on the keypad
KP_Divide	112	65455	/ on the keypad
KP_Down	88	65433	↓ on the keypad
KP_End	87	65436	<i>end</i> on the keypad

.keysym	.keycode	.keysym_num	Key
KP_Enter	108	65421	<i>enter</i> on the keypad
KP_Home	79	65429	<i>home</i> on the keypad
KP_Insert	90	65438	<i>insert</i> on the keypad
KP_Left	83	65430	← on the keypad
KP_Multiply	63	65450	× on the keypad
KP_Next	89	65435	<i>PageDown</i> on the keypad
KP_Prior	81	65434	<i>PageUp</i> on the keypad
KP_Right	85	65432	→ on the keypad
KP_Subtract	82	65453	- on the keypad
KP_Up	80	65431	↑ on the keypad
Next	105	65366	<i>PageDown</i>
Num_Lock	77	65407	<i>NumLock</i>
Pause	110	65299	<i>pause</i>
Print	111	65377	<i>PrintScrn</i>
Prior	99	65365	<i>PageUp</i>
Return	36	65293	The <i>enter</i> key (<i>control-M</i>). The name Enter refers to a mouse-related event, not a keypress; see Section 54, “Events” (p. 157)
Right	102	65363	→
Scroll_Lock	78	65300	<i>ScrollLock</i>
Shift_L	50	65505	The left-hand <i>shift</i> key
Shift_R	62	65506	The right-hand <i>shift</i> key
Tab	23	65289	The <i>tab</i> key
Up	98	65362	↑

54.6. Writing your handler: The Event class

The sections above tell you how to describe what events you want to handle, and how to bind them. Now let us turn to the writing of the handler that will be called when the event actually happens.

The handler will be passed an **Event** object that describes what happened. The handler can be either a function or a method. Here is the calling sequence for a regular function:

```
def handlerName(event):
```

And as a method:

```
def handlerName(self, event):
```

The attributes of the **Event** object passed to the handler are described below. Some of these attributes are always set, but some are set only for certain types of events.

.char	If the event was related to a KeyPress or KeyRelease for a key that produces a regular ASCII character, this string will be set to that character. (For special keys like <i>delete</i> , see the .keysym attribute, below.)
.delta	For MouseWheel events, this attribute contains an integer whose sign is positive to scroll up, negative to scroll down. Under Windows, this value will be a multiple of 120; for example, 120 means scroll up one step, and -240 means scroll down two steps. Under MacOS, it will be a multiple of 1, so 1 means scroll up one step, and -2 means scroll down two steps. For Linux mouse wheel support, see the note on the Button event binding in Section 54.3, “Event types” (p. 158).
.height	If the event was a Configure , this attribute is set to the widget's new height in pixels.
.keycode	For KeyPress or KeyRelease events, this attribute is set to a numeric code that identifies the key. However, it does not identify which of the characters on that key were produced, so that “x” and “X” have the same .keyCode value. For the possible values of this field, see Section 54.5, “Key names” (p. 160).
.keysym	For KeyPress or KeyRelease events involving a special key, this attribute is set to the key's string name, e.g., 'Prior' for the <i>PageUp</i> key. See Section 54.5, “Key names” (p. 160) for a complete list of .keysym names.
.keysym_num	For KeyPress or KeyRelease events, this is set to a numeric version of the .keysym field. For regular keys that produce a single character, this field is set to the integer value of the key's ASCII code. For special keys, refer to Section 54.5, “Key names” (p. 160).
.num	If the event was related to a mouse button, this attribute is set to the button number (1, 2, or 3). For mouse wheel support under Linux, bind Button-4 and Button-5 events; when the mouse wheel is scrolled up, this field will be 4, or 5 when scrolled down.
.serial	An integer serial number that is incremented every time the server processes a client request. You can use .serial values to find the exact time sequence of events: those with lower values happened sooner.
.state	An integer describing the state of all the modifier keys. See the table of modifier masks below for the interpretation of this value.
.time	This attribute is set to an integer which has no absolute meaning, but is incremented every millisecond. This allows your application to determine, for example, the length of time between two mouse clicks.
.type	A numeric code describing the type of event. For the interpretation of this code, see Section 54.3, “Event types” (p. 158).
.widget	Always set to the widget that caused the event. For example, if the event was a mouse click that happened on a canvas, this attribute will be the actual Canvas widget.
.width	If the event was a Configure , this attribute is set to the widget's new width in pixels.
.x	The x coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
.y	The y coordinate of the mouse at the time of the event, relative to the upper left corner of the widget.
.x_root	The x coordinate of the mouse at the time of the event, relative to the upper left corner of the screen.
.y_root	The y coordinate of the mouse at the time of the event, relative to the upper left corner of the screen.

Use these masks to test the bits of the `.state` value to see what modifier keys and buttons were pressed during the event:

Mask	Modifier
<code>0x0001</code>	<i>Shift.</i>
<code>0x0002</code>	<i>Caps Lock.</i>
<code>0x0004</code>	<i>Control.</i>
<code>0x0008</code>	Left-hand <i>Alt.</i>
<code>0x0010</code>	<i>Num Lock.</i>
<code>0x0080</code>	Right-hand <i>Alt.</i>
<code>0x0100</code>	Mouse button 1.
<code>0x0200</code>	Mouse button 2.
<code>0x0400</code>	Mouse button 3.

Here's an example of an event handler. Under Section 54.1, "Levels of binding" (p. 157), above, there is an example showing how to bind mouse button 2 clicks on a canvas named `self.canv` to a handler called `self.__drawOrangeBlob()`. Here is that handler:

```
def __drawOrangeBlob(self, event):
    '''Draws an orange blob in self.canv where the mouse is.
    ...
    r = 5    # Blob radius
    self.canv.create_oval(event.x-r, event.y-r,
                         event.x+r, event.y+r, fill='orange')
```

When this handler is called, the current mouse position is `(event.x, event.y)`. The `.create_oval()` method draws a circle whose bounding box is square and centered on that position and has sides of length $2 \times r$.

54.7. The extra arguments trick

Sometimes you would like to pass other arguments to a handler besides the event.

Here is an example. Suppose your application has an array of ten checkbuttons whose widgets are stored in a list `self.cbList`, indexed by the checkbutton number in `range(10)`.

Suppose further that you want to write one handler named `__cbHandler` for `<Button-1>` events in all ten of these checkbuttons. The handler can get the actual `Checkbutton` widget that triggered it by referring to the `.widget` attribute of the `Event` object that gets passed in, but how does it find out that checkbutton's index in `self.cbList`?

It would be nice to write our handler with an extra argument for the checkbutton number, something like this:

```
def __cbHandler(self, event, cbNumber):
```

But event handlers are passed only one argument, the event. So we can't use the function above because of a mismatch in the number of arguments.

Fortunately, Python's ability to provide default values for function arguments gives us a way out. Have a look at this code:

```

def __createWidgets(self):
    ...
    self.cbList = []      # Create the checkbutton list
    for i in range(10):
        cb = tk.Checkbutton(self, ...)
        self.cbList.append(cb)
        cb.grid( row=1, column=i)
    def handler(event, self=self, i=i):  1
        return self.__cbHandler(event, i)
    cb.bind('<Button-1>', handler)
    ...
    def __cbHandler(self, event, cbNumber):
        ...

```

- 1** These lines define a new function `handler` that expects three arguments. The first argument is the `Event` object passed to all event handlers, and the second and third arguments will be set to their default values—the extra arguments we need to pass it.

This technique can be extended to supply any number of additional arguments to handlers.

54.8. Virtual events

You can create your own new kinds of events called *virtual events*. You can give them any name you want so long as it is enclosed in double pairs of <><>.

For example, suppose you want to create a new event called <><>panic<><>, that is triggered either by mouse button 3 or by the *pause* key. To create this event, call this method on any widget *w*:

```
w.event_add('<><>panic<><>', '<Button-3>',
            '<KeyPress-Pause>')
```

You can then use '<><>panic<><>' in any event sequence. For example, if you use this call:

```
w.bind('<><>panic<><>', h)
```

any mouse button 3 or *pause* keypress in widget *w* will trigger the handler *h*.

See `.event_add()`, `.event_delete()`, and `.event_info()` under Section 26, “Universal widget methods” (p. 97) for more information about creating and managing virtual events.

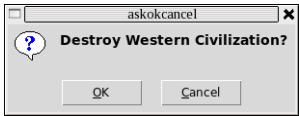
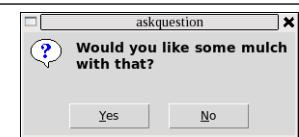
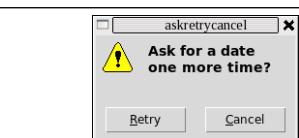
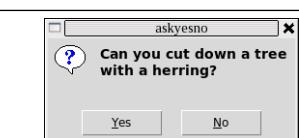
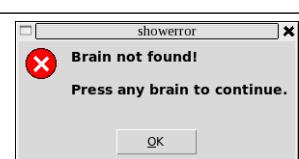
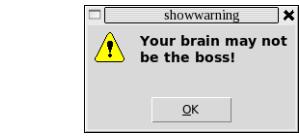
55. Pop-up dialogs

Tkinter provides three modules that can create pop-up dialog windows for you:

- Section 55.1, “The `tkMessageBox` dialogs module” (p. 165), provides an assortment of common pop-ups for simple tasks.
- Section 55.2, “The `tkFileDialog` module” (p. 167), allows the user to browse for files.
- Section 55.3, “The `tkColorChooser` module” (p. 168), allows the user to select a color.

55.1. The `tkMessageBox` dialogs module

Once you import the `tkMessageBox` module, you can create any of these seven common types of pop-up menu by calling functions from this table.

	.askokcancel(<i>title</i> , <i>message</i> , <i>options</i>)
	.askquestion(<i>title</i> , <i>message</i> , <i>options</i>)
	.askretrycancel(<i>title</i> , <i>message</i> , <i>options</i>)
	.askyesno(<i>title</i> , <i>message</i> , <i>options</i>)
	.showerror(<i>title</i> , <i>message</i> , <i>options</i>)
	.showinfo(<i>title</i> , <i>message</i> , <i>options</i>)
	.showwarning(<i>title</i> , <i>message</i> , <i>options</i>)

In each case, the *title* is a string to be displayed in the top of the window decoration. The *message* argument is a string that appears in the body of the pop-up window; within this string, lines are broken at newline ('\n') characters.

The *option* arguments may be any of these choices.

default

Which button should be the default choice? If you do not specify this option, the first button ("OK", "Yes", or "Retry") will be the default choice.

To specify which button is the default choice, use `default=C`, where *C* is one of these constants defined in `tkMessageBox`: CANCEL, IGNORE, OK, NO, RETRY, or YES.

icon

Selects which icon appears in the pop-up. Use an argument of the form `icon=I` where *I* is one of these constants defined in `tkMessageBox`: ERROR, INFO, QUESTION, or WARNING.

parent

If you don't specify this option, the pop-up appears above your root window. To make the pop-up appear above some child window *W*, use the argument `parent=W`.

Each of the "ask..." pop-up functions returns a value that depends on which button the user pushed to remove the pop-up.

- `askokcancel`, `askretrycancel`, and `askyesno` all return a `bool` value: `True` for “OK” or “Yes” choices, `False` for “No” or “Cancel” choices.
- `askquestion` returns `u'yes'` for “Yes”, or `u'no'` for “No”.

55.2. The `tkFileDialog` module

The `tkFileDialog` module provides two different pop-up windows you can use to give the user the ability to find existing files or create new files.

`.askopenfilename(option=value, ...)`

Intended for cases where the user wants to select an existing file. If the user selects a nonexistent file, a popup will appear informing them that the selected file does not exist.

`.asksaveasfilename(option=value, ...)`

Intended for cases where the user wants to create a new file or replace an existing file. If the user selects an existing file, a pop-up will appear informing that the file already exists, and asking if they really want to replace it.

The arguments to both functions are the same:

`defaultextension=s`

The default file extension, a string starting with a period ('.')). If the user's reply contains a period, this argument has no effect. It is appended to the user's reply in case there are no periods.

For example, if you supply a `defaultextension=' . jpg'` argument and the user enters 'gojiro', the returned file name will be 'gojiro.jpg'.

`filetypes=[(label1, pattern1), (label2, pattern2), ...]`

A list of two-element tuples containing file type names and patterns that will select what appears in the file listing. In the screen picture below, note the pull-down menu labeled “Files of type:”. The `filetypes` argument you supply will populate this pull-down list. Each `pattern` is a file type name (“PNG” in the example) and a pattern that selects files of a given type (“*.png” in the example).

`initialdir=D`

The path name of the directory to be displayed initially. The default directory is the current working directory.

`initialfile=F`

The file name to be displayed initially in the “File name:” field, if any.

`parent=W`

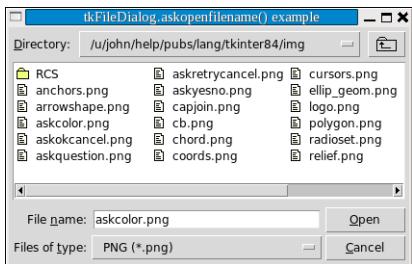
To make the pop-up appear over some window `W`, supply this argument. The default behavior is that the pop-up will appear over your application's root window.

`title=T`

If specified, `T` is a string to be displayed as the pop-up window's title.

If the user selects a file, the returned value is the complete path name of the selected file. If the user uses the *Cancel* button, the function returns an empty string.

Here is an example:



55.3. The `tkColor Chooser` module

To give your application's user a popup they can use to select a color, import the `tkColor Chooser` module and call this function:

```
result = tkColor Chooser.askcolor(color, option=value, ...)
```

Arguments are:

color

The initial color to be displayed. The default initial color is a light gray.

title=text

The specified *text* appears in the pop-up window's title area. The default title is "Color".

parent=W

Make the popup appear over window *W*. The default behavior is that it appears over your root window.

If the user clicks the *OK* button on the pop-up, the returned value will be a tuple (*tuple*, *color*), where *tuple* is a tuple (*R*, *G*, *B*) containing red, green, and blue values in the range [0,255] respectively, and *color* is the selected color as a regular *Tkinter* color object.

If the users clicks *Cancel*, this function will return (*None*, *None*).

Here's what the popup looks like on the author's system:

