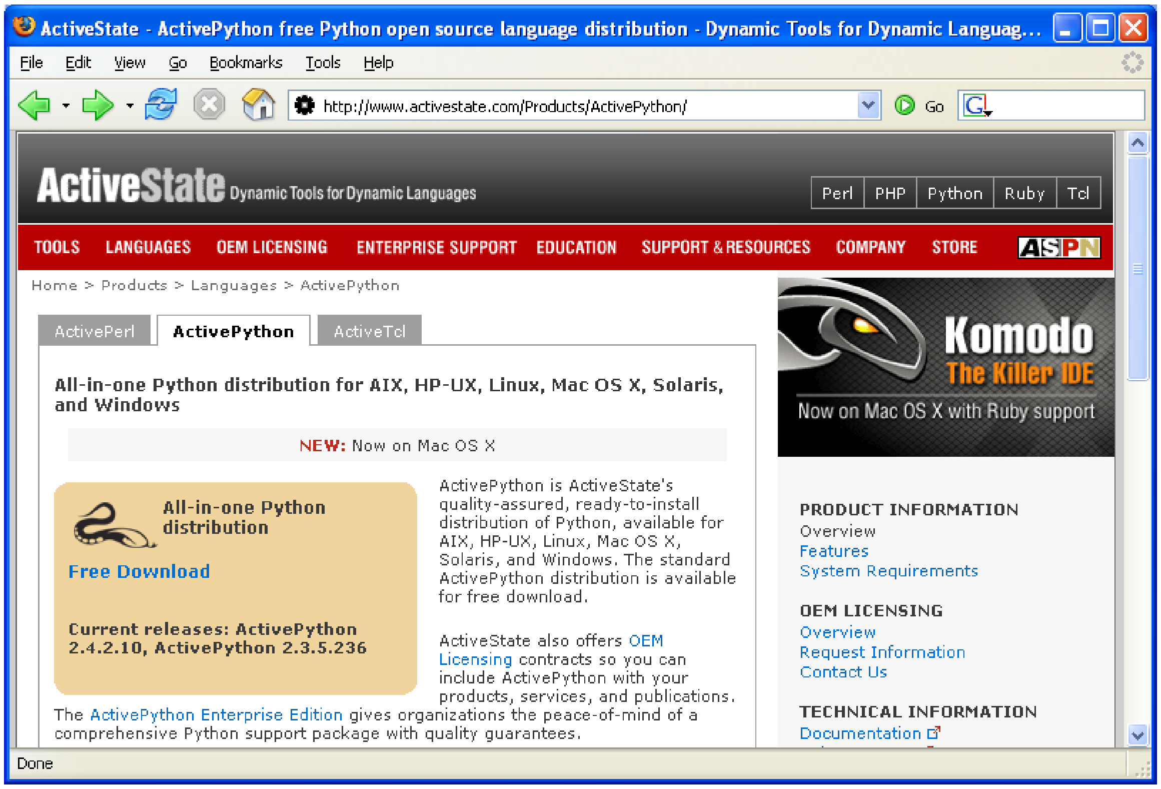
1 Installing and Running Python

Python is available for free for most computer platforms including Microsoft Windows, Macintosh, Unix and Linux machines. You can get further information about the different versions of Python at http://www.python.org/download/. One of the frequent stumbling blocks for beginner programmers of any language is successfully installing the programming language and learning the basic mechanics of writing and running programs. Because of this I am going to provide you with a step-by-step Python installation guide. Since most people use Microsoft Windows machines, I'm going to focus on installing and running Python on Windows. Keep in mind that this set of tutorials is not specifically directed at Python for Windows. You can use Python (and this set of tutorials) on any machine. If you are using Python in a school lab that already has Python installed on its computers you can skip section 1.1.

1.1 Installing Python on your own Windows machine

You can download a free version of Python from
http://www.activestate.com/Products/ActivePython/

When you go to that site, you'll see the introductory page shown below, which has a "Free Download" link in a box on the left. When you click on the download link it will take you to a registration page. If you want you can enter your name and email address or you can leave it blank.


ActiveState - ActivePython free Python open source language distribution - Dynamic Tools for Dynamic Languag...
File Edit View Go Bookmarks Tools Help
http://www.activestate.com/Products/ActivePython/
Go
ActiveState Dynamic Tools for Dynamic Languages
Perl PHP Python Ruby Tcl
TOOLS LANGUAGES OEM LICENSING ENTERPRISE SUPPORT EDUCATION SUPPORT & RESOURCES COMPANY STORE ASPN
Komodo
The Killer IDE
Now on Mac OS X with Ruby support
Done

Home > Products > Languages > ActivePython

ActivePerl | ActivePython | ActiveTcl

All-in-one Python distribution for AIX, HP-UX, Linux, Mac OS X, Solaris, and Windows

NEW: Now on Mac OS X

All-in-one Python distribution

Free Download

Current releases: ActivePython 2.4.2.10, ActivePython 2.3.5.236

ActivePython is ActiveState's quality-assured, ready-to-install distribution of Python, available for AIX, HP-UX, Linux, Mac OS X, Solaris, and Windows. The standard ActivePython distribution is available for free download.

ActiveState also offers OEM Licensing contracts so you can include ActivePython with your products, services, and publications.

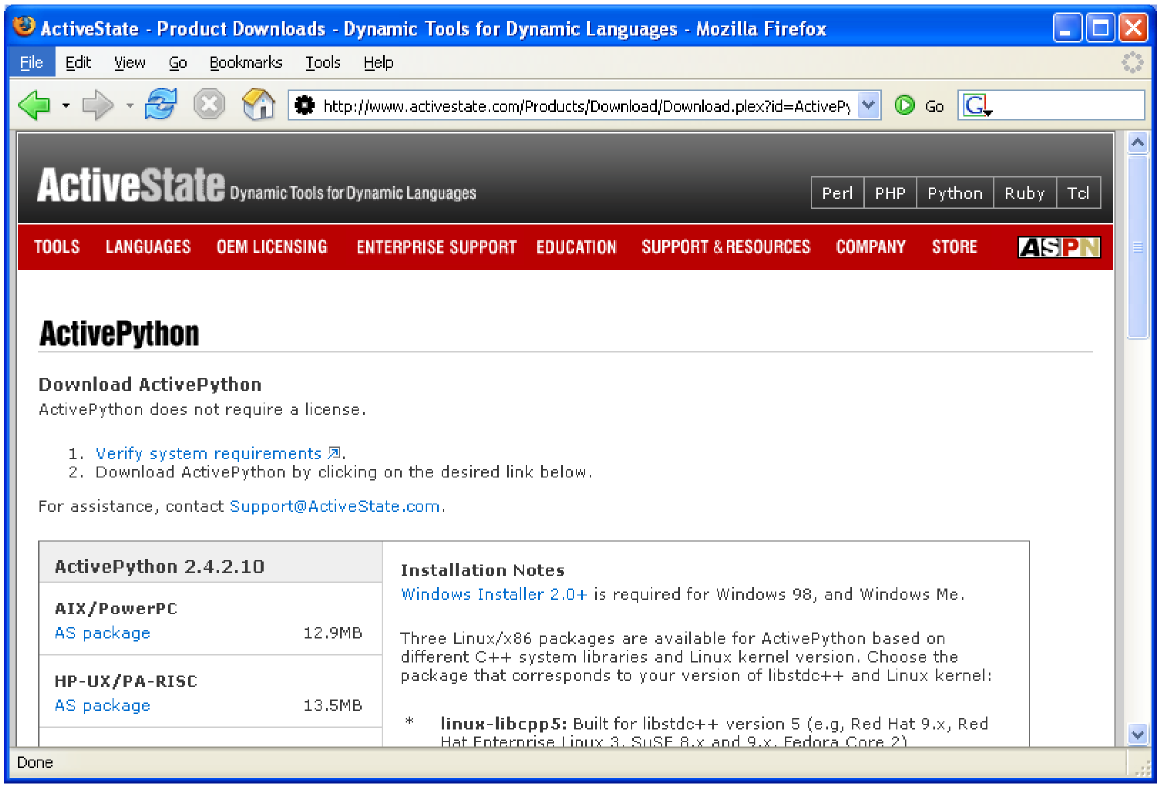
The ActivePython Enterprise Edition gives organizations the peace-of-mind of a comprehensive Python support package with quality guarantees.

PRODUCT INFORMATION
Overview
Features
System Requirements

OEM LICENSING
Overview
Request Information
Contact Us

TECHNICAL INFORMATION
Documentation

When you press the next button, you will see a web page that looks something like this:


ActiveState - Product Downloads - Dynamic Tools for Dynamic Languages - Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://www.activestate.com/Products/Download/Download.plex?id=ActivePy
Go
ActiveState Dynamic Tools for Dynamic Languages
Perl PHP Python Ruby Tcl
TOOLS LANGUAGES OEM LICENSING ENTERPRISE SUPPORT EDUCATION SUPPORT & RESOURCES COMPANY STORE ASPN
Done

ActivePython

Download ActivePython

ActivePython does not require a license.

1. Verify system requirements.
2. Download ActivePython by clicking on the desired link below.

For assistance, contact Support@ActiveState.com.

ActivePython 2.4.2.10

AIX/PowerPC
AS package 12.9MB

HP-UX/PA-RISC
AS package 13.5MB

Installation Notes

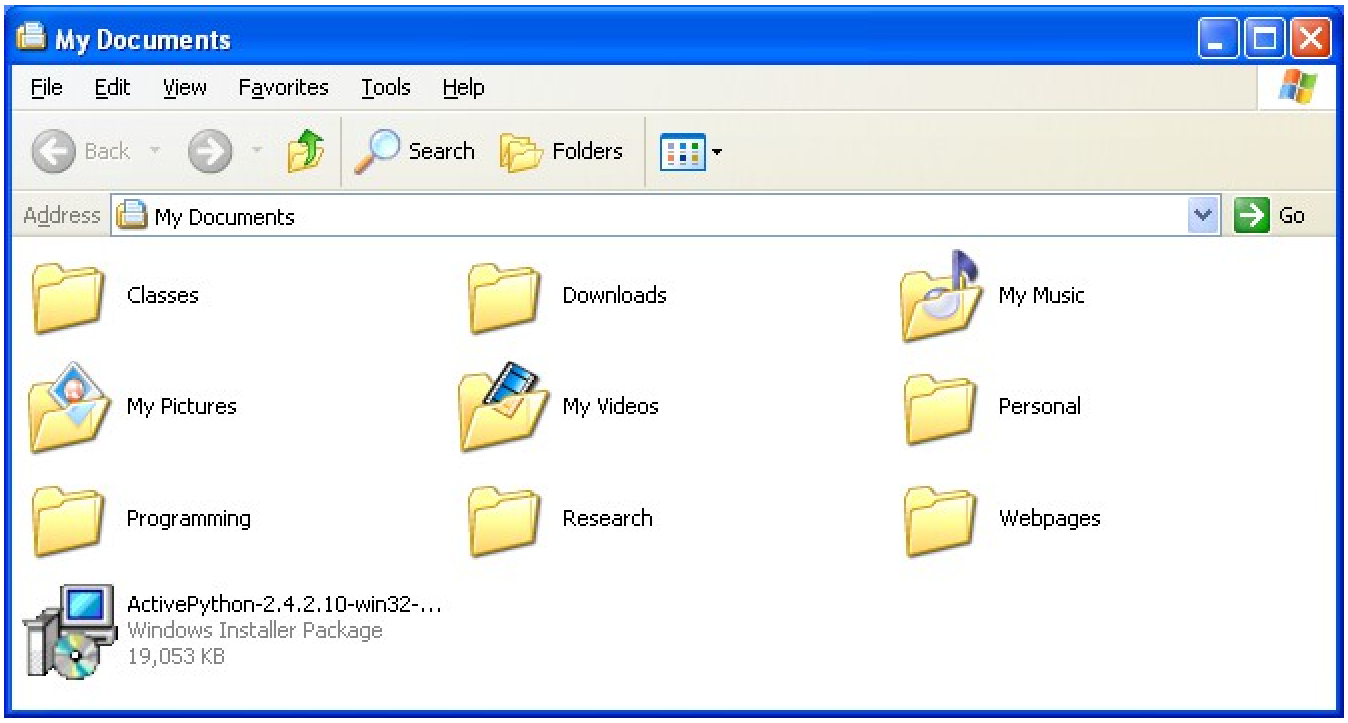
Windows Installer 2.0+ is required for Windows 98, and Windows Me.

Three Linux/x86 packages are available for ActivePython based on different C++ system libraries and Linux kernel version. Choose the package that corresponds to your version of libstdc++ and Linux kernel:

* linux-libcpp5: Built for libstdc++ version 5 (e.g, Red Hat 9.x, Red Hat Enterprise Linux 3, SuSE 8.x and 9.x, Fedora Core 2)

You will want to download the Windows/x86 MSI version of the latest build. To do so, scroll down until you see Windows/x86, and click on the MSI link just below it.[1] You will then be prompted for where to save the file and you can choose a convenient location on you computer. I selected "My Documents."

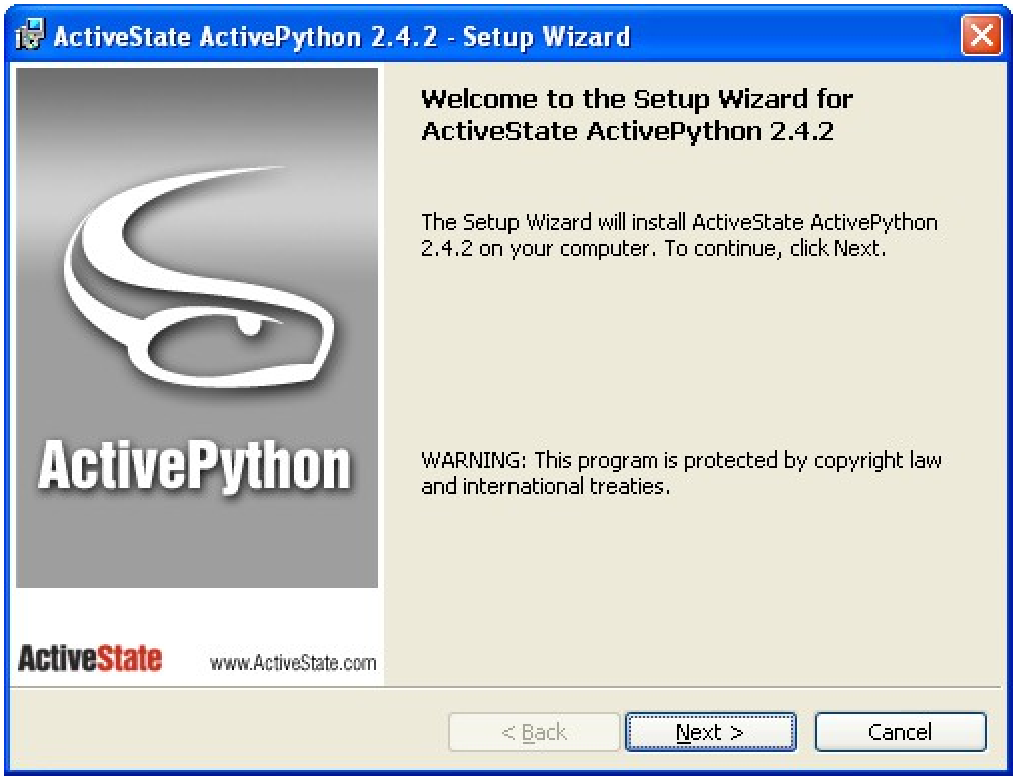
After you download this package you will need to install it by going to the location where you saved the file:



and double-clicking on the ActivePython icon:

After you double click you will see the first of several installation dialog boxes:

[1] Once you click on the MSI link, in addition to automatically downloading a file, you will see a web page titled 'Now downloading'. This page has information about the Microsoft Windows Installer package, which you will need to download if you have Windows 95, 98 or NT.

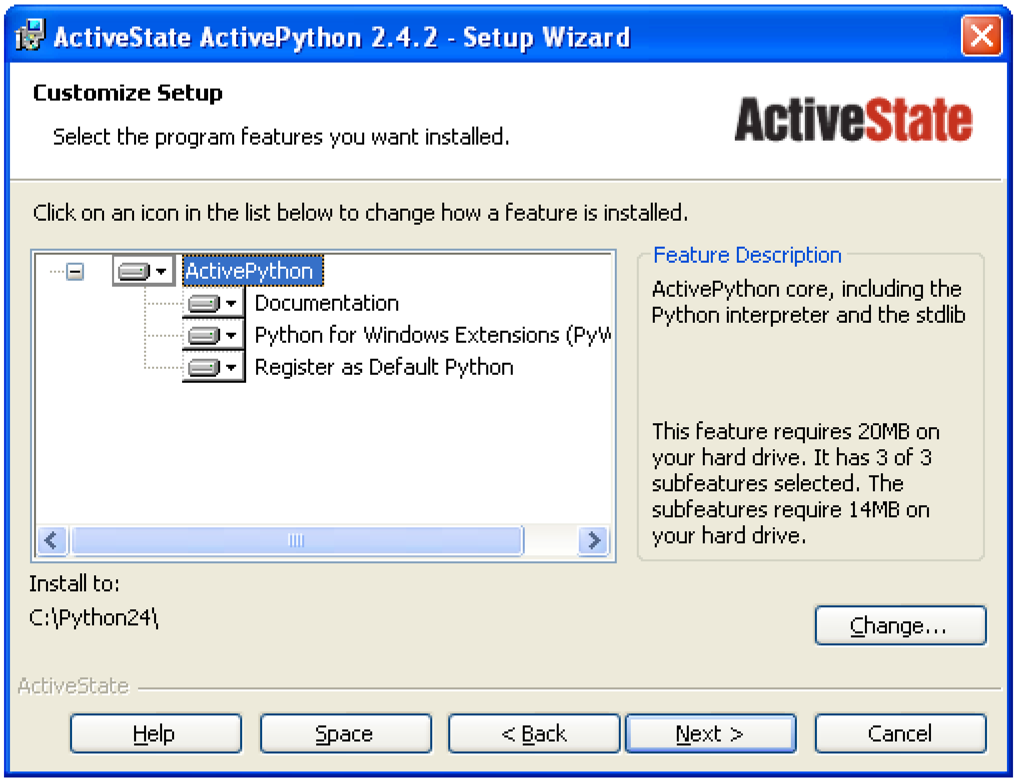


Welcome to the Setup Wizard for ActiveState ActivePython 2.4.2

The Setup Wizard will install ActiveState ActivePython 2.4.2 on your computer. To continue, click Next.

WARNING: This program is protected by copyright law and international treaties.

Click *next* to proceed to the next dialog. Then select the radio button labeled "I accept the terms of the license agreement' and press the *next* button. The next screen (shown below) allows you to change some installation options, but you can probably just click 'next.'


ActiveState ActivePython 2.4.2 - Setup Wizard
ActiveState
ActivePython
Documentation
Python for Windows Extensions (PyW
Register as Default Python
Feature Description
Change...
ActiveState
Help
Space
< Back
Next >
Cancel

Customize Setup

Select the program features you want installed.

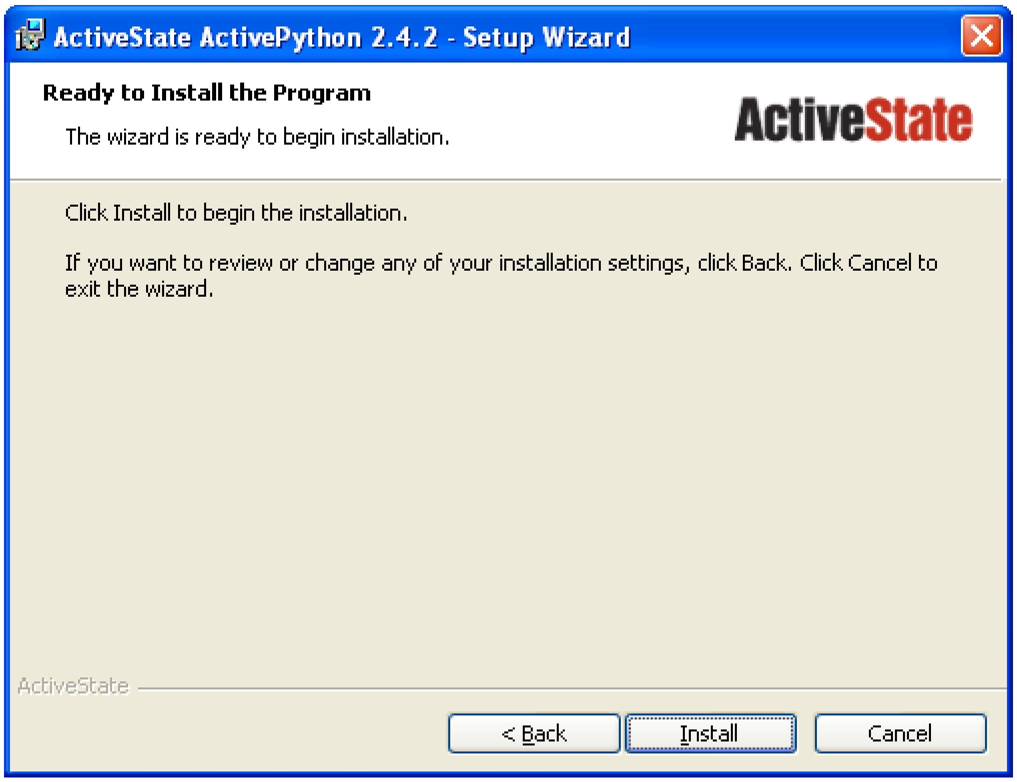
Click on an icon in the list below to change how a feature is installed.

ActivePython core, including the Python interpreter and the stdlib

This feature requires 20MB on your hard drive. It has 3 of 3 subfeatures selected. The subfeatures require 14MB on your hard drive.

Install to:
C:\Python24\

The next screen indicates that the installer is ready to begin. Click "Install".



ActiveState ActivePython 2.4.2 - Setup Wizard

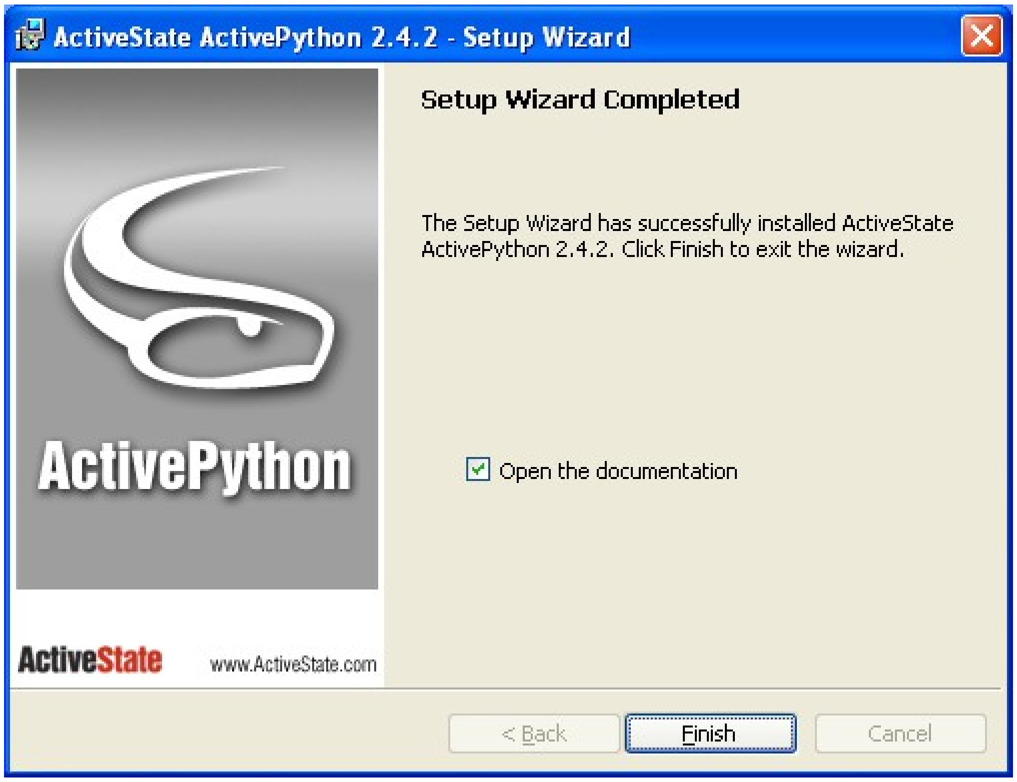
Ready to Install the Program

The wizard is ready to begin installation.

Click Install to begin the installation.

If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.

The setup wizard will then install ActivePython. This process may take several minutes. After it is complete you will see:



ActiveState ActivePython 2.4.2 - Setup Wizard

Setup Wizard Completed

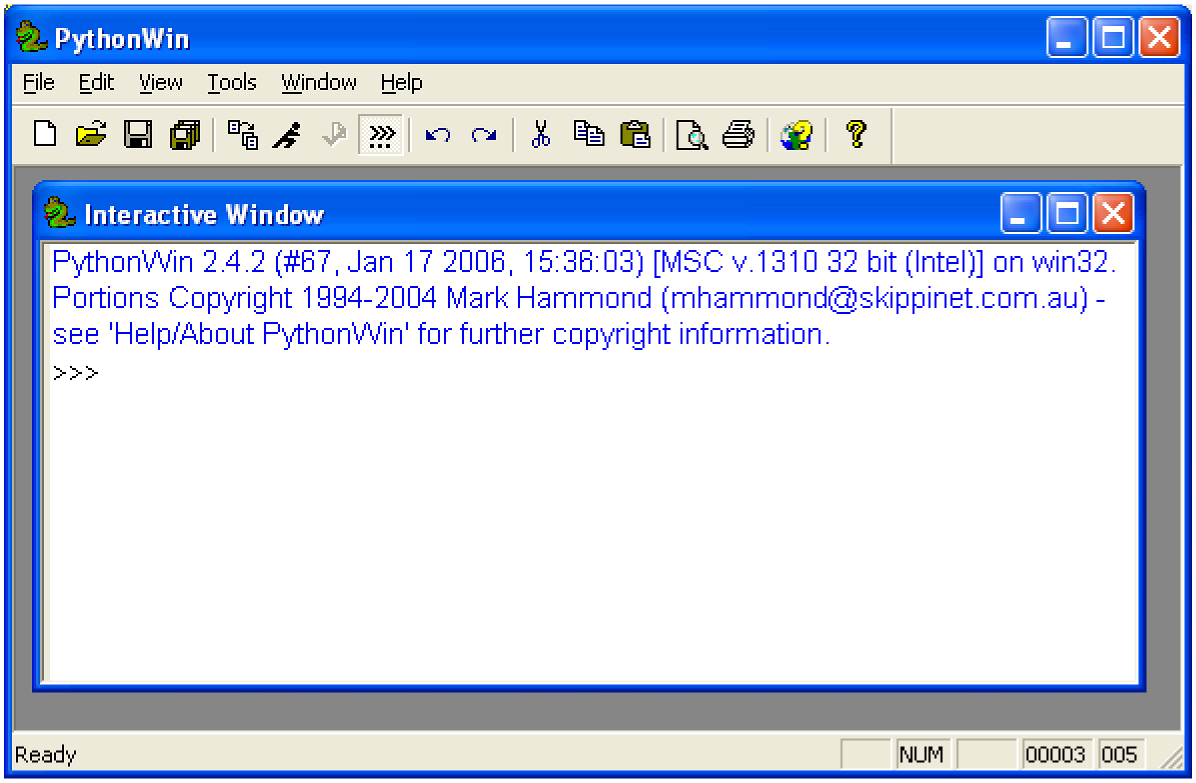
The Setup Wizard has successfully installed ActiveState ActivePython 2.4.2. Click Finish to exit the wizard.

[✓] Open the documentation

Press the ‘finish’ button. Congratulations! You just installed Python on your computer.

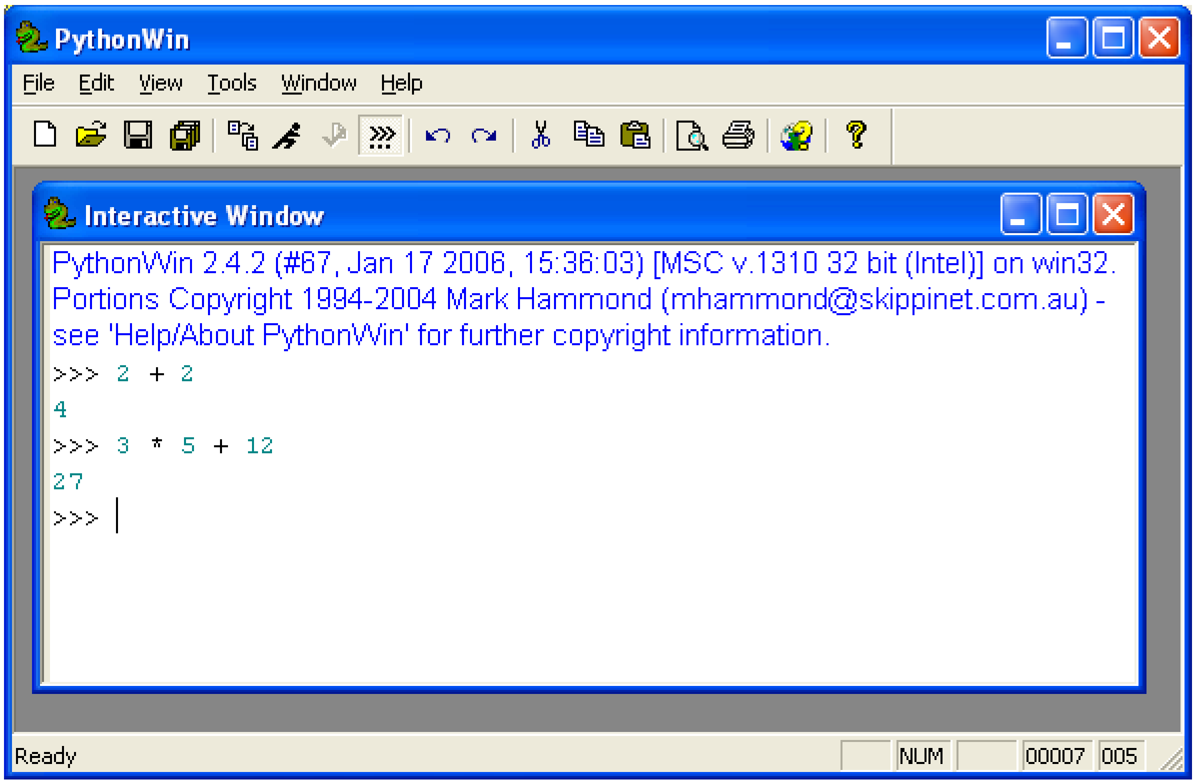
1.2 Writing and running Python programs.

There are several ways of writing and running Python programs using ActivePython. The method I prefer is to run the program, *PythonWin IDE*. This program can be started—if you did the default installation—from the start menu, under Programs → ActiveState ActivePython 2.4 → Pythonwin IDE. When you start the program you'll see something like:



```
PythonWin 2.4.2 (#67, Jan 17 2006, 15:36:03) [MSC v.1310 32 bit (Intel)] on win32.
Portions Copyright 1994-2004 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>>
```

You can type Python statements in the window labeled "Interactive Window" and immediately see the results:



```
PythonWin 2.4.2 (#67, Jan 17 2006, 15:36:03) [MSC v.1310 32 bit (Intel)] on win32.
Portions Copyright 1994-2004 Mark Hammond (mhammond@skippinet.com.au) -
see 'Help/About PythonWin' for further copyright information.
>>> 2 + 2
4
>>> 3 * 5 + 12
27
>>>
```

In the above example, I typed in 2 + 2 and Python responded by displaying '4'. Instead of showing you all the interactions with Python by using screen captures, I'll use the following notation:

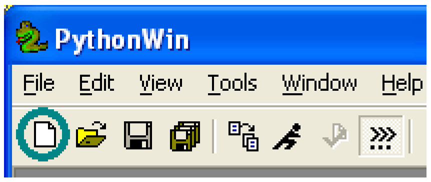
```
>>> 2 + 2
4
>>> 3 * 5 + 12
27
```

Here, as throughout the book, I'm showing Python code in the courier font. When I show interactive sessions, I'm putting what you type in bold and the system response in plain text.

The >>> is the Python prompt—it indicates that Python is waiting for input. As you just saw, Python can act as a simple calculator. Here are some additional examples:

```
>>> 12 * 5000                    (The asterisk means 'multiply.')
60000
>>> (12 * 5000) / 52
1153
>>> print "Hi there"
Hi there
>>>
```

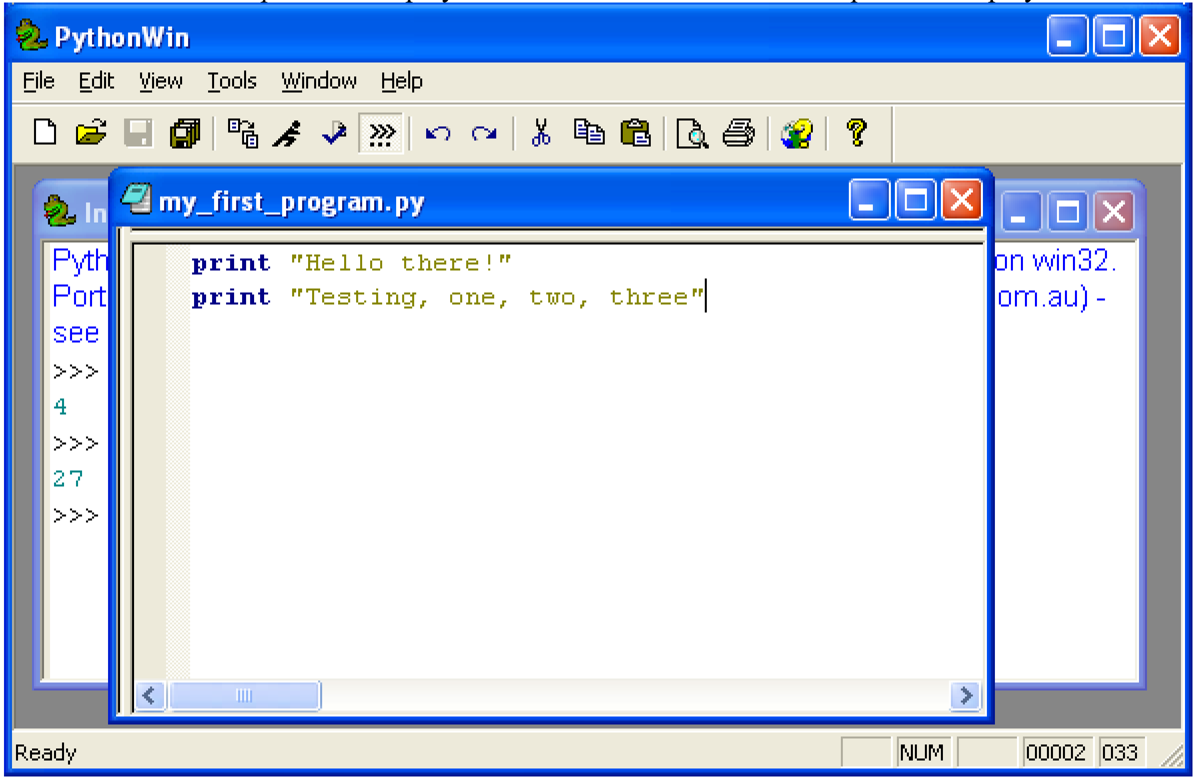
You will find this interactive mode very helpful, but for the most part you'll be writing Python programs in text files. This is similar to working with Microsoft Word or any text editor. Here's how to do it. To work on a new document you can either click on the new document icon in PythonWin (the blank sheet of paper icon in the far left of the button toolbar):



or you can go under the file menu and choose ‘new’. In either case you will get a dialog box asking if you want to create a new Python script or a new “Grep”. When you choose ‘Python script’ and click ‘OK’ you’ll get a new window appearing in the PythonWin application. In that new window type

```
    print "Hello there!"
    print "Testing, one, two, three"
```

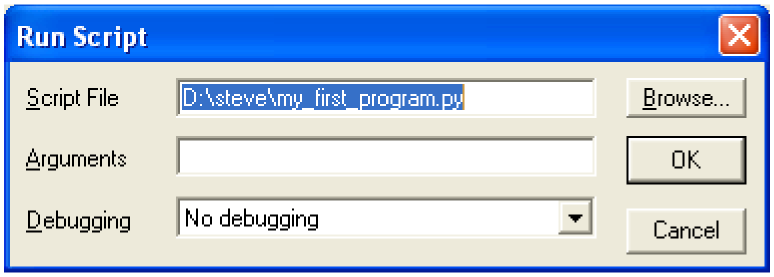
Notice that the words *print* are displayed in bolded blue and the text in quotes is displayed in an olive font.



```
print "Hello there!"
print "Testing, one, two, three"
```

PythonWin helps you write programs by showing words that are in its vocabulary in bolded blue. As you work more in Python you’ll find this is very useful. For example, if you intended to type *print* and you typed *pirnt* instead, you are apt to notice the error because the word is not in bold.

Now go ahead and save the document. Again, it’s similar to working with Microsoft Word. You can either click on the save icon (the floppy disk) or select *save* from the file menu. You’ll be prompted for a name. Give it some meaningful name. I chose *my_first_program*. When PythonWin saves the file, it adds a *.py* to the end of the filename to indicate that it’s a Python file. Now, we can run the program by clicking on the run icon (the little running person) or choosing *run* from the file menu. The following popup window will appear.



Script File: D:\steve\my_first_program.py
Arguments:
Debugging: No debugging

If you click 'OK' at this point, PythonWin will run your script *myFirstProgram.py* You should see the results in the interactive window:

```
>>> Hello there!
Testing, one, two, three
```

These results show a funky detail about PythonWin—namely that one line of printed text is displayed in black after the prompt '>>>' and the next line of text is displayed in turquoise without the prompt. For now don't worry about this weirdness.

What happens if you wrote my_first_program.py yesterday and now you want to run it again today? You click on the little running person (or choose *run* from the menu) and use the *browse* function in the above window to find you script and execute it. If you wrote my_first_program.py yesterday and now you want to edit it, you either can select 'open' from the file menu or click on the 'open' icon (the half-open file folder).

That's it for the quick tour of Pythonwin. In sum, you can type statements directly in the Interactive Window, and you can create Python scripts with the built in editor and run those scripts right in Pythonwin.

I like the Pythonwin environment, but if you prefer, you can run Python directly from a Windows command prompt window, and create scripts using your favorite text editor. This method is similar to that for Unix and Linux machines and is discussed on the companion website.

1.3 Getting started

In the previous section you saw how you could type statements directly in the Python interpreter:

```
>>> 2 + 2
4
>>> print 'Hello there!'
Hello there!
```

And you also saw how you could create a file with Python statements and execute that file. Throughout this book, when I say let's write a script I mean let's create a file containing Python statements by opening a new python script window (or by using a text editor). For now, let's start by typing statements directly into the interpreter. In fact, let's start with a statement we already know:

```
>>> print 'Hello there!'
Hello there!
```

The line starting with *print* is a Python statement. The special word[2] *print* indicates that the following value should be displayed on the screen. In this case, the following value was "Hello there!", so that was printed.

[2] In Python special words like "print" are called keywords.

Dissecting the line

```
print 'Hello there!'
```

Gives us:

print	'Hello there!'
Keyword	Value

1.3.1 Strings

The value in this example is

```
'Hello there!'
```

This is called a string, which is defined as a sequence of characters enclosed in quotes. The quotes can either be single or double quotes, but the opening and closing quotes must match. For example,

```
'Hello there!'
"Hello there!"
```

are fine but

```
"Hello there!'
'Hello there!"
```

are not (The first starts with a double quote and ends with a single quote and the reverse is true about the second example).

What happens when you have double quotes as string delimiters and you have a double quote in the string? Let's give it a try:

```
>>> print "Pat wrote "Learn Python While You Sleep""
Traceback (  File "<interactive input>", line 1
    print "Pat wrote "Learn Python While You Sleep ""
                           ^
SyntaxError: invalid syntax
>>>
```

Python interprets the first quote, the one just before Pat, as the start of a string. It considers any characters following that quote part of the string until it encounters another quote. Python finds a quote just before the word *Learn* and considers it as marking the end of the string. Thus, it considers only the bold part to be the string:

"Pat wrote "Learn Python While You Sleep""

Python doesn't know what to do with the rest of the line, and, as a result, displays the error message "invalid syntax." To prevent this we use \" (that is, two characters—a backslash, '\', and a quote, '"') to represent quotes within a string. The Python statement:

>>> **print "Pat wrote \"Learn Python While You Sleep\""**

would display

```
Pat wrote "Learn Python While You Sleep"
```

The ‘\’ means ‘interpret the next character as a real genuine character.’

What do we do if we really want a backslash printed? In that case we use \\:

```
>> print "vp\\n: a vp missing a noun to its left"
```

displays

```
vp\n: a vp missing a noun to its left
```

Of course, since Python allows both single and double quote delimiters, you can also prevent problems by being a bit smarter as to which pair you use:

```
>>> print 'Pat wrote "Learn Python While You Sleep "'
Pat wrote "Learn Python While You Sleep"
```

Since I want to have double quotes in the string I use the single quotes as delimiters. In the next example I want to have a single quote used as an apostrophe in *don't* so I use double quotes to delimit the string.

```
>>> print "I don't understand"
I don't understand
```

If we used double quote delimiters in the first example or single ones in the second we get very different results:

```
>>> print 'I don't understand'
Traceback (  File "<interactive input>", line 1
    print 'I don't understand'
                 ^
SyntaxError: invalid syntax
>>>
```

Errors

When you type an ungrammatical sentence in a word processor you don’t expect your computer to lock up, and smoke to come out of it. You don’t expect your ungrammatical sentence to break your computer requiring a trip to the computer repair shop. It’s no big deal to create error-filled text with a word processor. Similarly it’s not a big deal to create a Python program containing an error—your computer is not going to break because you wrote a program containing an error. Errors are no big deal. You fix them and go on. Feel free to experiment. If at anytime reading these tutorials you wonder ‘*what* if’ just give your idea a try. There are errors that programmers commonly make and throughout these tutorials I will be describing how to detect them.

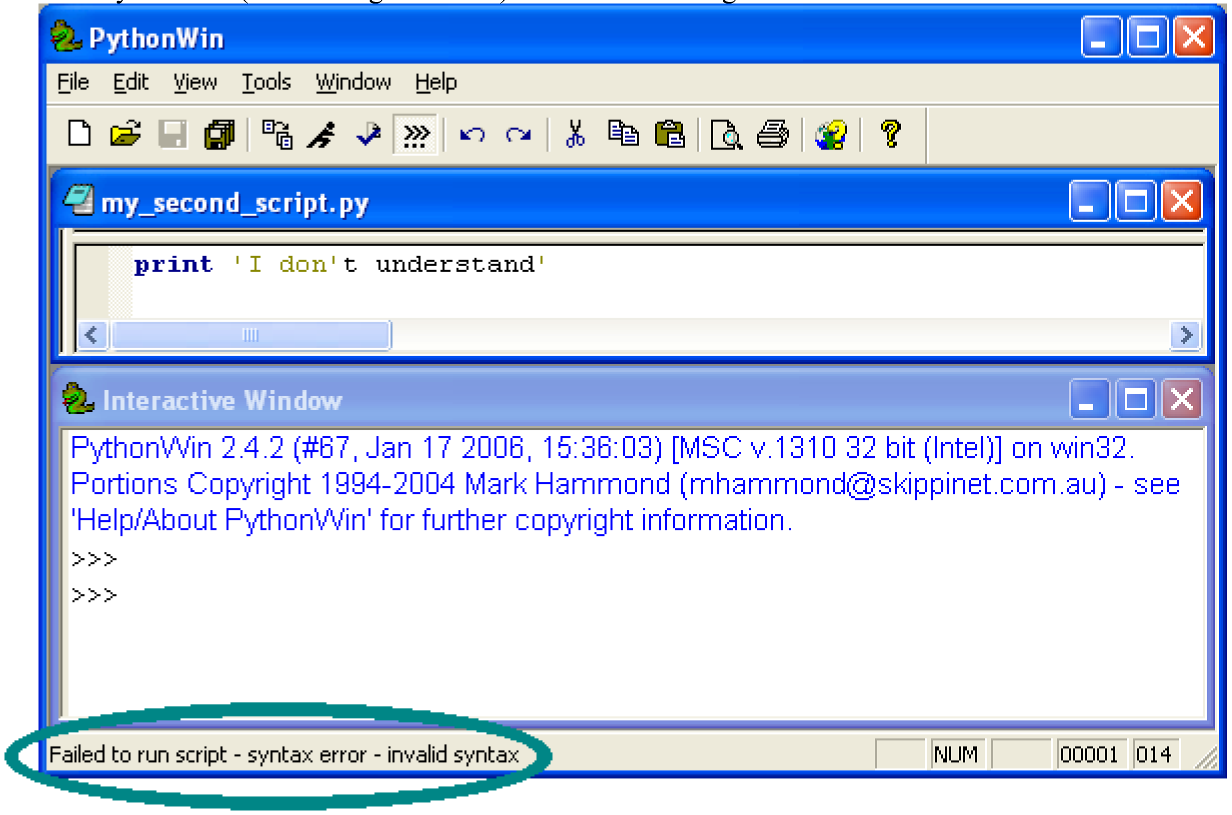
As shown above, when you type an expression that contains a syntax error in the interactive window, Python responds with an informative error message. Let’s take a look at what happens when there is an incorrect expression in a script. First I create a new file by clicking on the file icon (the blank page icon) or by select ‘new’ under the file menu. In the new window I type

```
print 'I don't understand'
```

When I typed this example I see PythonWin's automatic color coding mentioned above. I imagine you reading this text as a printed non-color document so I'm going to represent the different colors by different fonts. Using this convention the line looks something like:

print 'I don'`t understand`**'**

PythonWin sees the string as *I don*, terminating at the single quote after that, and cannot process *t understand.* Hopefully, this color coding will help me eliminate some of my errors, but let's say I don't notice this error and save the file as my_second_script.py (remember that Python automatically adds the 'py' extension to the filename).
When I try to run it (the running man icon) I see the following:



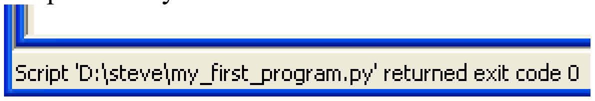
```
print 'I don't understand'
```

```
PythonWin 2.4.2 (#67, Jan 17 2006, 15:36:03) [MSC v.1310 32 bit (Intel)] on win32.
Portions Copyright 1994-2004 Mark Hammond (mhammond@skippinet.com.au) - see
'Help/About PythonWin' for further copyright information.
>>>
>>>
```

Failed to run script - syntax error - invalid syntax

Notice the error message on the bottom of the PythonWin window: *Failed to run script – syntax error – invalid syntax.* PythonWin also attempts to place the cursor at the location of the error. In this case it places the cursor immediately after the apostrophe in don't.

It's extremely common for beginning Python programmers to ignore messages that occur in this position on the bottom of the window. When your scripts get just a bit longer, failing to notice this error message can lead to hours of frustration. It's good to get in the practice of glancing down to this message area every time you run a script. When you see return exit code zero:



Script 'D:\steve\my_first_program.py' returned exit code 0

you know your script executed without errors.

Python scripts can contain multiple lines

Let's continue talking about Python scripts (writing statements in a file and then executing that file) A Python script can consist of a number of Python statements, which are executed in turn. Let's say we have the following script in a file called pat.py

```
print "Pat"
print "Hillary"
```

Let's run it by using the Pythonwin run command as described above. The result will be

```
Pat
Hillary
```

Let's play around with this a bit. It will seem silly now, but shortly what we learn through this playing will become useful.

Suppose I wanted the words *Pat* and *Hillary* to appear on the same line. The obvious and best solution would be to use one print statement:

```
print "Pat Hillary"
```

But there's an alternative. If we add a comma to the end of a print statement it means "add a space and stay on the same line." So,

```
print "Pat",
print "Hillary"
```

(with a comma after the first print statement) prints:

```
Pat Hillary
```

Conversely, if I wanted to use one print statement and have the names appear on different lines I can use the special character sequence `\n` which means go to a new line (that is, the '\n' represents the newline character).

```
>>> print "Pat\nHillary"
Pat
Hillary
```

Now, look at the following example and see if you can determine what the output will look like:

```
print "Person  ",
print "Phone number"
print "--------",
print "----------"
print "Pat     ",
print "541-1360"
print "Hillary ",
print "646-6520"
print "Kim     ",
print "646-3307"
```

It looks like the following:

```
Person   Phone number
```

```
-------- ----------
Pat      541-1360
Hillary  646-6520
Kim      646-3307
```

1.3.2 Names

In Python, you can give names[3] to bits of information. A name in Python must begin with a letter or an underscore '_'. The remaining characters can be letters, digits, or underscores. By *letters* I mean the letters a-z and their uppercase equivalents. No fair using Σ or even piżmo!!

The following are examples of allowable names:

My_phonenumber
test1
test2
NounPhrase

The following names are not allowed:

2ndtest	the first character cannot be a digit
John's_Phonenumber	the apostrophe character is not allowed in a name
Çie	the character 'Ç; is not allowed

Your names can be anything you like. If in one of my examples I use `part_of_speech` as a name, but you can use `pos` or `x39`. My `part_of_speech` is not 'special' in any way—it's just a name I invented. Of course it helps to pick a name that is meaningful enough to help you remember what you were doing in that part of your script. The name `x39` is not that helpful when you are trying to debug, `but part_of_speech`, or even `pos` is. A common naming convention is to use only lowercase letters and the underscore character, for example, to use `part_of_speech` rather than `partOfSpeech.` I will use this convention throughout this book.

Values

Names can be given to all sorts of values. For now, let's take a look at strings and numbers. We assign a name to a value by using the assignment operator, '='. Let's start by typing some things into the Python interpreter.

```
>>> word = 'glupstwo'
>>> part_of_speech = 'noun'
>>> gender = 'neuter'
>>> translation = 'Stupid thing'
>>> frequency = 512
```

Notice that when we typed a print statement we immediately received a response from Python:

```
>>> print "Pat"       ← our input
Pat                   ← Python's response
```

When we assign a name to a value, Python doesn't generate a response:

```
>>> part_of_speech = 'noun'
```

[3] You may also hear names called "variables".

```
>>>                          ← no response other than presenting a prompt
```

Internally, Python has associated the name with the value (in this case the name 'part_of_speech' with the value 'noun'), but externally, we just don't see it.

To get the value associated with a name we just use the name itself. For example, after I typed in the above I can do the following:

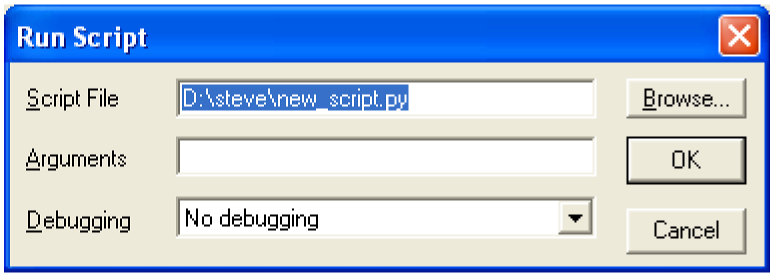
```
>>> word
'glupstwo'
>>> part_of_speech
'noun'
```

(I'm typing what is in the bold text and the Python interpreter is responding with what is in the non-bolded text.) Let's construct a script that does something similar. That is, create a new document in PythonWin and type in the following.

```
word = "glupstwo"
part_of_speech = "noun"
gender = "neuter"
translation = "stupid thing"

print word
print part_of_speech
```

Now save the document. For example, let's say we save it with the name new_script.py. Now, let's run the script by pressing the 'run' icon (the running man). You'll see the dialog box:



Script File: D:\steve\new_script.py
Arguments:
Debugging: No debugging

Now press 'OK' to run the script and you'll see the following results in the Interactive Window of the PythonWin application:

```
glupstwo
noun
```

Can you predict what output you'll see if you run the script:

```
word  = "glupswto"
part_of_speech = "noun"
gender  = "neuter"
translation  = "stupid thing"

print word,
print part_of_speech
```

3.3.3 String addition

Python's addition operator '+' joins together two strings. So

The added string	is the same as
`"walk" + "ing"`	`"walking`
`"Accusative " + "Case"`	`"Accusative Case"`

In this way we can change the block of print statements:

```
print word,
print part_of_speech
```

to

```
print word + " " + part_of_speech
```

If we forget to use the plus sign we get an error:

```
>>> print word + " " part_of_speech
Traceback (  File "<interactive input>", line 1
    print word + " " part_of_speech
                                  ^
SyntaxError: invalid syntax
>>>
```

Here I forgot to put a '+' between the " " and the part_of_speech.

3.3.4 Common numerical operators

Now we will take a quick look at the typical operators you can use for numbers. The following table shows these basic operators.

Numerical Operators	Result	Operation
3 + 2	5	Addition
3 - 2	1	Subtraction
3 * 2	6	Multiplication
3 / 2	1.5	Division
3 % 2	1	Modulo[4]

Python knows about different types of numbers. Of importance to us right now are two types: integers (for example, 2, 214, -5,291) and real numbers (0.0, -777.1, 6.25).

Just as a verb can be intransitive, transitive, or ditransitive, Python operators take a specific number of arguments.[5] The operations in the above table take one or more arguments and return a result. For example, in the first row of the table the addition operation is taking two integers (3 and 2) and returning an integer result (5). **The key point** here is that what Python returns (whether an integer or a real number) depends on what it

[4] The remainder when you divide 3 by 2.

[5] Anyone who has studied subcategorization knows that things are much more complex than I describe. For example, Thompson and Hopper suggest that there is a continuum of transitivity. Similarly, the argument structure of Python commands is not as simple as I outline here. Later chapters will discuss these complexities.

received as arguments. If it receives two integers as arguments, these operations will return an integer. If they receive 2 real numbers, they'll return a real. Finally, if these operations receive one integer and another real, they will return a real. This is particularly important for **division**. If we divide one integer by another we will get an integer result:

```
>>> 5 / 2
2
```

If you want a real number result you need to have one of the arguments be a real:

```
>>> 5 / 2.0
2.5
>>> 5.0 / 2
2.5
```

Let's go through a simple example. Say there are 2 teaching assistants for a linguistics class. We create a name "ta" with the value 2:

```
ta = 2
```

There are 27 quizzes to grade

```
quizzes = 27
```

We want the TA's to grade an equal number of quizzes:

```
to_grade = quizzes / ta
```

and now let's figure out the remainder:

```
remainder = quizzes % TA
```

the TAs (let's call them ta1 and ta2) should grade

```
ta1 = to_grade + remainder
ta2 = to_grade
```

Let's now put this together into one program:

```
ta = 2
quizzes = 27
to_grade = quizzes / ta
remainder = quizzes % ta
ta1 = to_grade + remainder
ta2 = to_grade
print "TA #1 should grade:",
print ta1,
print "quizzes"
print "TA #2 should grade:",
print ta2,
print "quizzes"
```

This results in the output

```
TA #1 should grade: 14 quizzes
TA #2 should grade: 13 quizzes
```

You may be wondering why I didn't use Python's append operator ('+') and do something like

```
print "TA #1 should grade: " + ta1 + " quizzes"
```

The '+' operator does different things depending on its arguments. If its arguments are strings, it appends them together:

```
>>>"walk" + "ing"
walking

>>>"1" + "2" + "3"
123
```

If the arguments to '+' are numbers, it performs addition:

```
>>>1 + 2 + 3
6
```

hint: Remember this distinction between "1" + "2" + "3" and 1 + 2 + 3. It will help you answer a puzzler that is coming up at the end of this tutorial.

Once again, when its arguments are strings '+' appends the arguments, but when both are numbers it performs addition. With that in mind let's look at the line

```
print "TA #1 should grade: " + ta1
```

The value of ta1 is a number, 14, so the line is equivalent in some way to

```
print "TA #1 should grade: " + 14
```

One argument is a string and the other a number. This is a case that the '+' operator cannot deal with and this statement generates an error. Shortly we will be examining ways around this problem. For now we can get around this problem by using multiple print statements with commas as illustrated above.

3.3.5 Comments

Comments are bits of text in your program that are intended to be an aid to you and others looking at your program. Comments are preceded by the '#' symbol. Python ignores everything between the '#' and the end of the line. For example, if you need to hand in a program you wrote for a class you are taking, you may want to put a 'header' in your program:

```
# Assignment 1.1
# 'The healthy linguist'
#
# 8/23/2005
# Hillary McMaster
#
```

You may also want add little notes elsewhere:

```
    pos = "n"  # the current part of speech
```

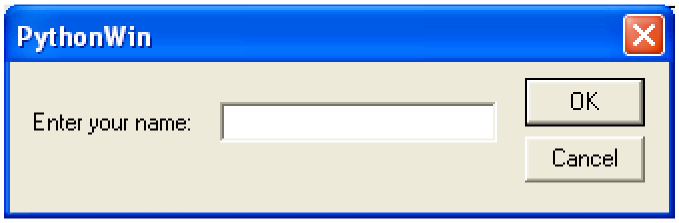
Comments will be much more useful as your programs get more complex. One cool thing about using PythonWin, is that it color codes various things. We've already seen that the names of Python's special keywords (like 'print') are displayed in blue and strings are displayed in olive. Comments are displayed in green. This color coding improves the readability of your program.

3.3.6 Getting input from the user

We can get input from the user by using the function raw_input(). Let's try this out and then I'll explain the details. In PythonWin create a new document (program) and type in the following:

```
    name = raw_input('Enter your name: ')
    print 'Hi, ' + name + ', nice to meet you!'
```

Save this program using whatever name you'd like. When you run it, the following dialog box will appear:



Let's say we type in the name 'Ann' and press 'OK'. Now, in the PythonWin Interactive Window (this window may be blocked by the window containing your code) we should see:

```
    Hi, Ann, Nice to meet you!
```

Pretty cool.

Let's take a look at the first line of your program:

```
    name = raw_input('Enter your name: ')
```

That "raw_input" thing is a function. Just like operators, functions in Python take a certain number of arguments. The raw_input function takes one argument, a string that will serve as a prompt to the user. This prompt can be any string:

```
        name = raw_input('Enter your name: ')
        name = raw_input('Name: ')
        name = raw_input('Can you please enter your name: ')
```

The function raw_input, opens up a dialog box containing the prompt string and a text field where the user can type in a response.[6] When the user presses the 'OK' button, raw_input returns the response as a string. That is, if the user types in "Ann" and presses 'OK', raw_input returns the string "Ann". You can think of it as:

```
        name = raw_input('Enter your name: ')
```

getting converted to:

[6] This function, raw_input, does slightly different things on other types of computers. For example, on Unix it does not open a dialog window, it just presents a command line prompt.

```
            name = 'Ann'
```

This now looks like the assignment statements we looked at earlier, with the value ‘Ann’ being given the name ‘name’.

This isn’t a wonderfully sophisticated user interface—it is a bit rough—but at least it will get us started doing some fun stuff.

Let’s look at another example:

```
            num1 = raw_input('Enter a number: ')
            num2 = raw_input('Enter another number: ')
            print num1 + num2
```

When I run this program I enter 12 as the first number and 34 as the second. What gets printed in the Interactive Window may surprise you:

```
      1234
```

Why does this get printed and not the answer I expected, 46? (I gave you a hint earlier in this tutorial.)

Try to answer this on your own before looking at the next paragraph.

It’s because raw_input treats everything the user types in as a string. When the user types in 12, raw_input returns the string '12' and when the user types in 34, raw_input returns the string '34'. Then when we perform the operation:

```
      '12' + '34'
```

the ‘+’ is interpreted as the append operator and we get the string ‘1234’.

Luckily, Python has a special function that converts strings to numbers—that is, it will convert the string '12' to the number 12. The function is called int and it takes one argument, a string, and returns a number (an integer). So we can type the following in the interactive window:

```
      >>> int('12')
      12                    (returns the number 12)
      >>> print '12' + '34'
      1234                  (appends the strings and then prints the result)
      >>> print int('12') + int('34')
      4                     (converts the strings to integers, adds them, and then prints the result)
```

Now, armed with this additional knowledge, revise this script and run it:

```
            num1 = raw_input('Enter a number: ')
            num2 = raw_input('Enter another number: ')
            print num1 + num2
```

[if you are stumped see hint 1 at the end of this tutorial]

Exercise 1.1 Healthy linguists

As you know, it's important that linguists take part in aerobic exercise. Now, as I'm sure you are all aware, there are two dangers related to linguists doing aerobic activities. The first occurs when 2 or more linguists go cycling or running. They start off humming along nicely; the conversation turns to some linguistic issue (e.g., parasitic gaps) and all of sudden the linguists' energies are directed to argumentation and the exercise pace drops dramatically. The other danger stems from the overachieving quality that many linguists have. Many a promising linguist who looked like she was on the high road to linguistic stardom lost it all when she decided to give 120% effort running or cycling, got her heart rate up to Lance Armstrong sprint levels, and had her heart explode.[7] As you may remember, this led to the Linguistic Society of America's Council of Physical Fitness recommendation that linguists should always exercise with a computer science student to serve as pacer. Unfortunately, not all of us have access to computer science students and we need to take it upon ourselves to exercise at an appropriate level. We can do this by exercising in a particular heart rate range. A rough rule of thumb for computing maximum heart rate is:

$$220 - \text{age}$$

The exercise range is from 70-85% of the maximum heart rate. For example, consider a person who is 36 years old. 220 minus the age (36) is 184. 70% of 184 is 128 and 85% of 184 is 156. So the exercise range is between 128 and 156.

What does this have to do with Python?
Well, I'd like you to write a program that will ask the user for the following input:

```
Enter name:
Enter age:

Ann, a 36 yr. old linguist, should exercise for 30 minutes with a
heartrate between 128 and 156.
```

Okay, that's the exercise. Good luck!

If you want to see one way of doing this problem see the solutions section at the end of this tutorial or you can download a file containing the solution code at the companion website (http://www.zacharski.org/python/solutions).

[7] This is just for dramatic effect. I know of no linguist who died this way.

Zipping along versus presenting all the gory details
My approach in writing this set of tutorials is to zip along and present you with enough Python details for you to start doing interesting things. I am not presenting an exhaustive description of Python—in other words, I'm leaving out stuff. Some of the gaps I'll fill in during later tutorials. If at any point you want to know more about some aspect of Python you may want to consult the reference material accessed from the 'help' menu of PythonWin.

Zipping along versus plodding along
It's been my experience that most beginners get very confused over the minutia that is most often ignored in intro to programming courses and books. For this reason, I have started out very slowly and given detailed instructions on how to install Python and write and run your first programs. As you get more comfortable with Python and with the PythonWin programming enviroment I'll be speeding things up.

Comments
In order to make this series of tutorials better, I am very interested in getting your comments. If at any point things are unclear, or if at some point you write a program that you are convinced should work but for some inexplicable reason does not, email me. My email address is ron@zacharski.org.

Hints:

1) Just change the last line to print int(num1) + int(num2)

Solutions

Exercise 1.1 Healthy linguists

```
# one possible solution for exercise 1.1
# (the healthy linguist)
#

# first get input from user
name = raw_input('Enter name: ')
age_string = raw_input('Enter age: ')

# convert the age string to a number
age = int(age_string)

# Maximum heart rate
max_hr = 220 - age

# low range
low = max_hr * .7

# high range
high = max_hr * .85

# Now print response
print name + ', a ' + age_string + ' yr. old linguist,',
print 'should exercise for 30 minutes'
print 'with a heartrate between %d and %d' % (low, high)
```