Introduction to Computer Science Using Python: A Computational Problem-Solving Focus

Charles Dierbach

WILEY

VP & Executive Publisher:	Don Fowley
Executive Editor:	Beth Lang Golub
Assistant Editor:	Samantha Mandel
Marketing Manager:	Christopher Ruel
Marketing Assistant:	Ashley Tomeck
Photo Editor:	Hilary Newman
Cover Designer:	Thomas Nery
Associate Production Manager:	Joyce Poh
Production Editor:	Jolene Ling
Cover Illustration:	Norm Christiansen

This book was set in 10/12 Times LT Std by Aptara. Text and cover were printed and bound by Courier Kendallville.

This book is printed on acid free paper.

Library of Congress Cataloging-in-Publication Data

Dierbach, Charles, 1953–
  Introduction to Computer Science Using Python: A Computational Problem-Solving Focus/Charles Dierbach.
    p. cm.
  Includes index.
  ISBN 978-0-470-55515-6 (pbk.)
1. Python (Computer program language) I. Title.
  QA76.73.P98D547 2012
  005.13'3—dc23

2012027172

Printed in the United States of America
10 9 8 7 6 5 4 3 2 1

Python 3 Programmers Reference

APPENDIX

The author gratefully acknowledges the contributions of Leela Sedaghat to the style and content of this reference.

This reference provides the features of Python 3 that are most relevant for the text. Therefore, it is not intended to be an exhaustive resource. For complete coverage of the Python programming language, Standard Library and other Python-related information, we refer readers to the official web site: http://www.python.org

A. GETTING STARTED WITH PYTHON

A1. About Python

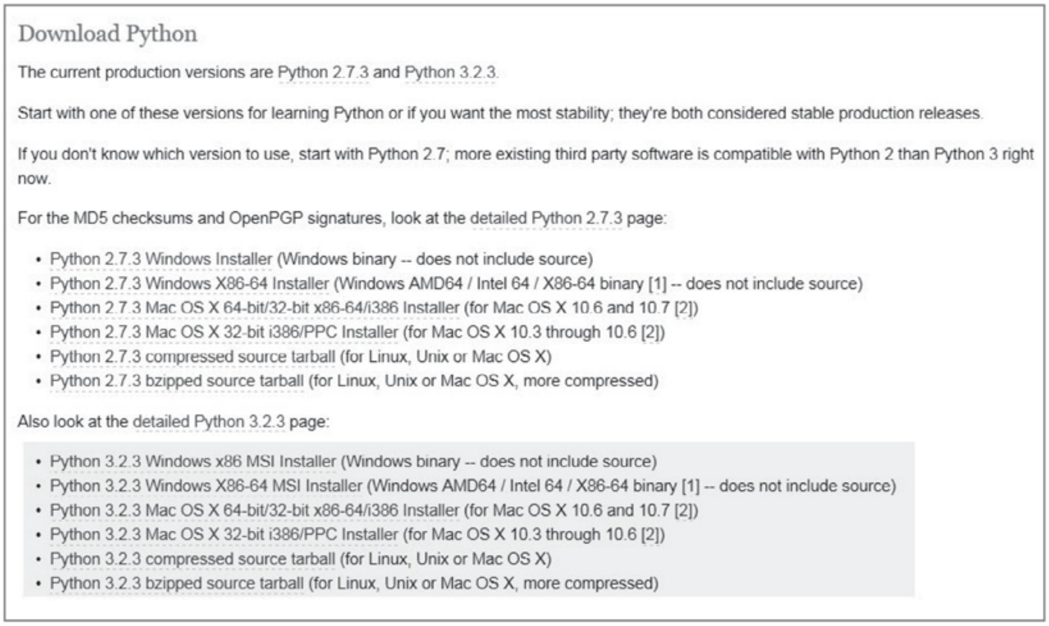
The Python programming language was created by Guido van Rossum (www.python.org/~guido) at the Centrum Wiskunde & Informatica (National Research Institute for Mathematics and Computer Science) in the Netherlands in the late 1980s. It is designed for *code readability*. It therefore has a clear and simple syntax. At the same time, Python also has a powerful set of programming features.

Python is free, open source software (http://www.python.org). The reference (standard) implementation of the Python programming language (called CPython) is managed by the non-profit Python Software Foundation. The language is bundled with a Python development environment called IDLE. The bundle is available for download at the official Python web site (see below). There are two, incompatible versions of Python currently supported: Python 2 (2.7.3) and Python 3 (3.2.3 at the time of this writing). Python 2.7.3 will be the last release version of Python 2. This text uses Python 3.

Python is growing in popularity. Many companies and organizations use Python including Google, Yahoo and YouTube. Python is also widely used in the scientific community, including the National Weather Service, Los Alamos National Laboratory, and NASA. Python also continues to gain popularity for use in introductory computer science courses.

A2. Downloading and Installing Python

To download and install the Python interpreter (and bundled IDLE program) go to http://www.python.org/download/ which displays the page shown in Figure A-1.


Download Python

The current production versions are Python 2.7.3 and Python 3.2.3.

Start with one of these versions for learning Python or if you want the most stability; they're both considered stable production releases.

If you don't know which version to use, start with Python 2.7; more existing third party software is compatible with Python 2 than Python 3 right now.

For the MD5 checksums and OpenPGP signatures, look at the detailed Python 2.7.3 page:

- Python 2.7.3 Windows Installer (Windows binary -- does not include source)
- Python 2.7.3 Windows X86-64 Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- Python 2.7.3 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and 10.7 [2])
- Python 2.7.3 Mac OS X 32-bit i386/PPC Installer (for Mac OS X 10.3 through 10.6 [2])
- Python 2.7.3 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 2.7.3 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)

Also look at the detailed Python 3.2.3 page:

- Python 3.2.3 Windows x86 MSI Installer (Windows binary -- does not include source)
- Python 3.2.3 Windows X86-64 MSI Installer (Windows AMD64 / Intel 64 / X86-64 binary [1] -- does not include source)
- Python 3.2.3 Mac OS X 64-bit/32-bit x86-64/i386 Installer (for Mac OS X 10.6 and 10.7 [2])
- Python 3.2.3 Mac OS X 32-bit i386/PPC Installer (for Mac OS X 10.3 through 10.6 [2])
- Python 3.2.3 compressed source tarball (for Linux, Unix or Mac OS X)
- Python 3.2.3 bzipped source tarball (for Linux, Unix or Mac OS X, more compressed)

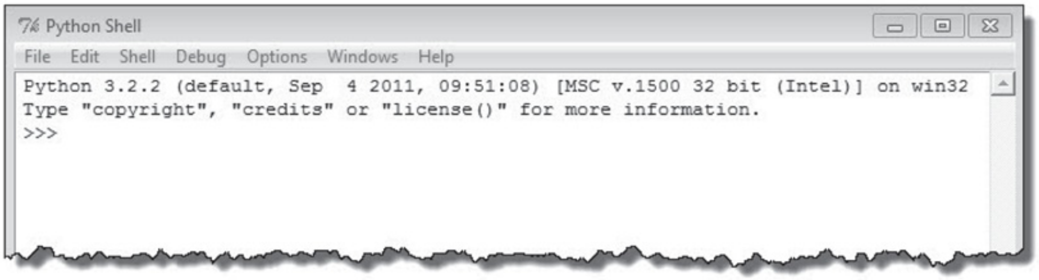
FIGURE A-1 Official Download Site of Python

Select the appropriate Python 3 download for your system: Windows x86 MSI installer (for typical Windows machines), Windows x86-64 (for 64-bit Windows machines, not typical), Mac OS X 64-bit/32-bit x86-64/i386 (for newer Macs) and Mac OS X 32-bit i386/PPC (for older Macs). There are also versions for Linux and Unix, as shown. Follow the installation directions.

A3. Program Development Using IDLE

What is IDLE? IDLE is an *integrated development environment* (IDE) for developing Python programs. An IDE consists of three major components: an **editor** for creating and modifying programs, an **interpreter** or **compiler** for executing programs, and a **debugger** for "debugging" (fixing errors in) a program.

When you execute IDLE on your system, a window such as shown in Figure A-2 will be displayed.



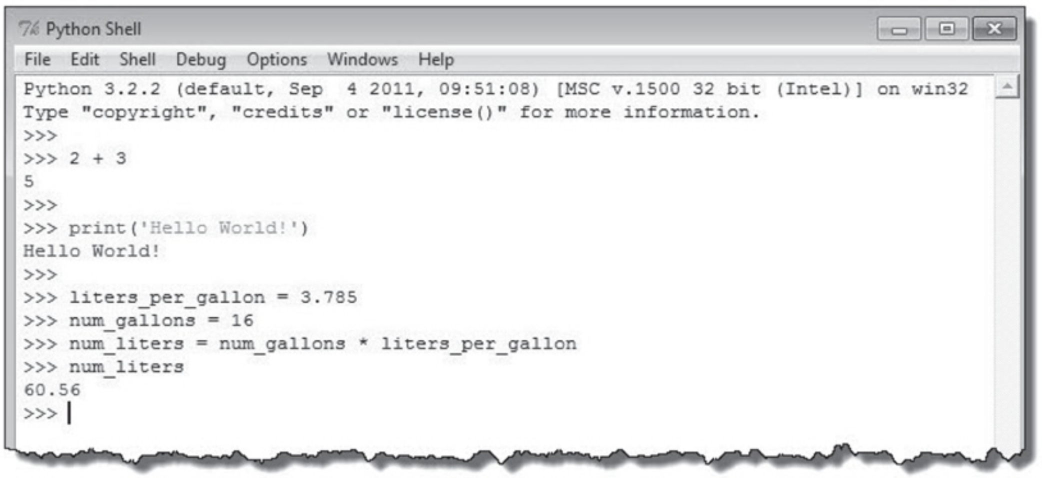
```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

FIGURE A-2 The Python Shell

Note that the version of Python is displayed on the lines right before the prompt (>>>).

Interacting with the Python Shell The Python Shell provides a means of directly interacting with the Python interpreter.

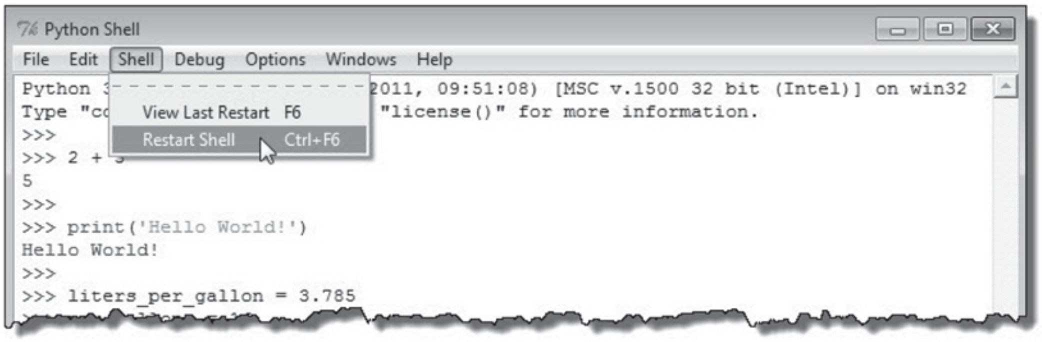
Thus, whatever Python code that is typed in will be immediately executed, as shown in Figure A-3.



```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> 2 + 3
5
>>>
>>> print('Hello World!')
Hello World!
>>>
>>> liters_per_gallon = 3.785
>>> num_gallons = 16
>>> num_liters = num_gallons * liters_per_gallon
>>> num_liters
60.56
>>> |
```

FIGURE A-3 Interacting with the Python Shell

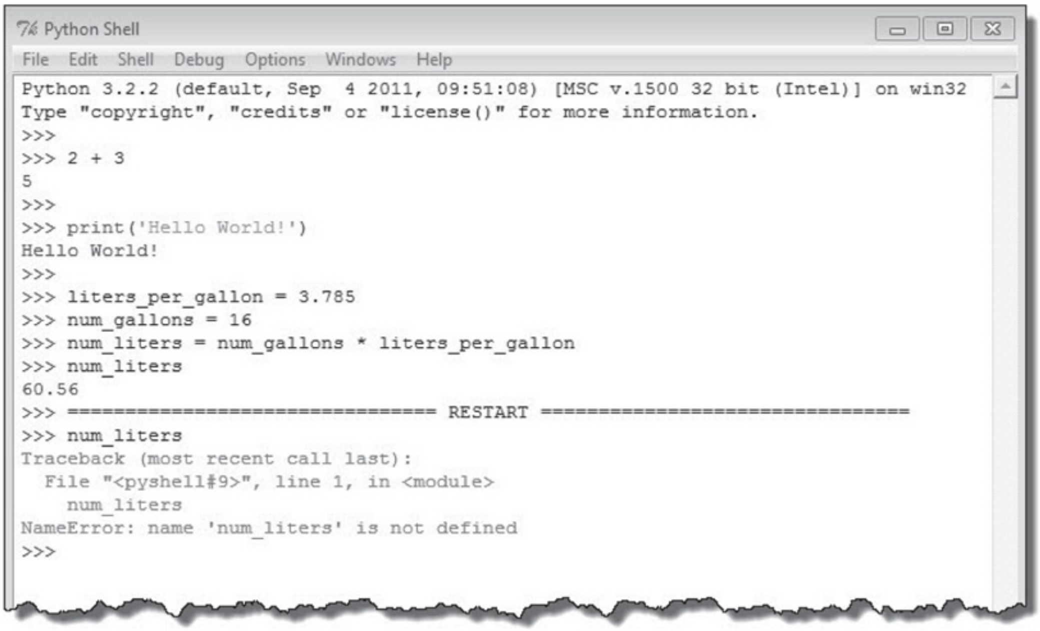
This is referred to in programming as *interactive mode*. All variables will remain defined until the shell is either closed or restarted. To restart the shell, select Restart Shell from the Shell dropdown list, as shown in Figure A-4.



```
Python 3                 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "co                  "license()" for more information.
>>>
>>> 2 + 3
5
>>>
>>> print('Hello World!')
Hello World!
>>>
>>> liters_per_gallon = 3.785
```

FIGURE A-4 Restarting the Python Shell

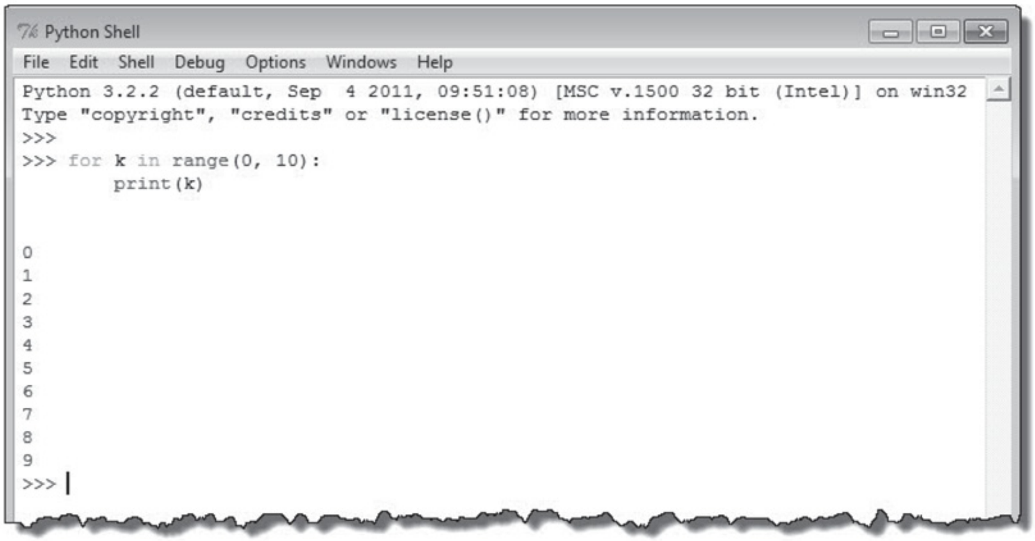
After the shell is restarted, all previously-defined variables become undefined and a "fresh" instance of the shell is executed, as shown in Figure A-5.



```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> 2 + 3
5
>>>
>>> print('Hello World!')
Hello World!
>>>
>>> liters_per_gallon = 3.785
>>> num_gallons = 16
>>> num_liters = num_gallons * liters_per_gallon
>>> num_liters
60.56
>>> ================================ RESTART ================================
>>> num_liters
Traceback (most recent call last):
  File "<pyshell#9>", line 1, in <module>
    num_liters
NameError: name 'num_liters' is not defined
>>>
```

FIGURE A-5 A New Instance of the Python Shell

Being able to immediately execute instructions in the Python Shell provides a simple means of verifying the behavior of instructions in Python. For example, if one forgot the specific range of numbers that built-in function `range(start, end)` produces for a given `start` and `end` value, the single `for` statement in Figure A-6 can be easily executed to determine that.

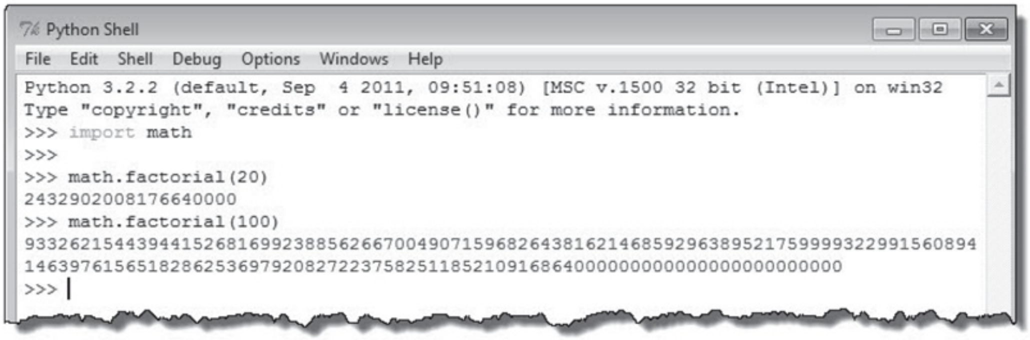


```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> for k in range(0, 10):
        print(k)


0
1
2
3
4
5
6
7
8
9
>>>
```

FIGURE A-6 Testing Small Program Segments in the Python Shell

We can also execute Python code in the shell that requires the use of modules. To utilize a particular module in the shell, the appropriate `import` statement is first entered.



```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> import math
>>>
>>> math.factorial(20)
2432902008176640000
>>> math.factorial(100)
9332621544394415268169923885626670049071596826438162146859296389521759999322991560894
146397615651828625369792082722375825118521091686400000000000000000000000000
>>> |
```

FIGURE A-7 Importing a Module into the Python Shell

Finally, when working in the interactive Python Shell, there are helpful keyboard shortcuts that the user may need to use. Some of the most commonly used commands are listed in Figure A-8.

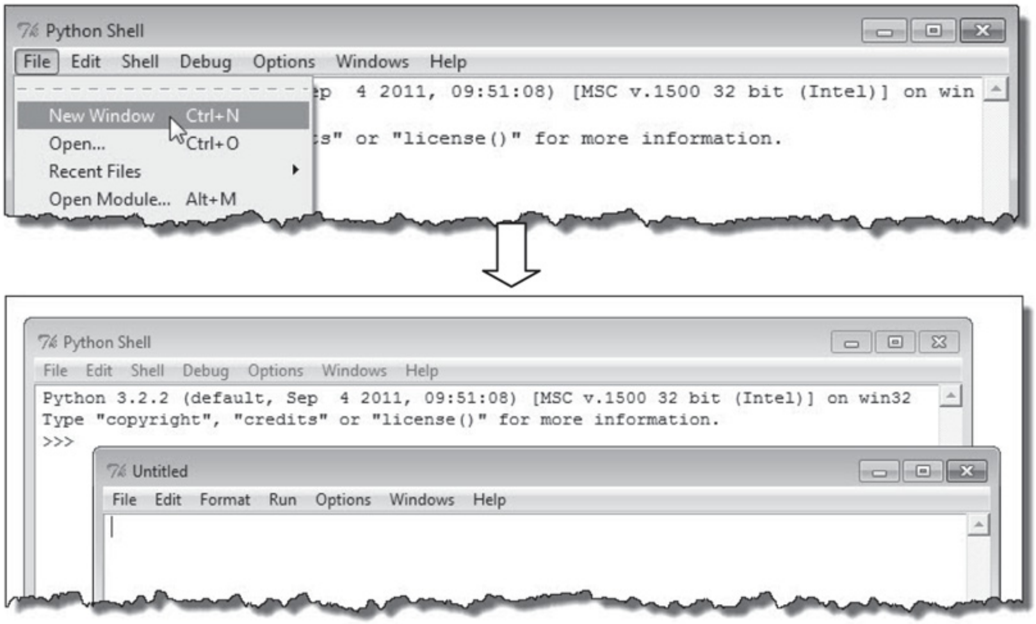
Windows Shortcut	Mac Shortcut	Action Performed
Ctrl+C	Ctrl+C	Terminates executing program
Alt+P	Ctrl+P	Retrieves previous command(s)
Alt+N	Ctrl+N	Retrieves next command(s)

FIGURE A-8 Python Shell Shortcuts

Creating and Editing Programs in IDLE Instructions typed into the shell are executed and then "discarded." For program development, however, we need to save instructions into a file. This is referred to as a **script** in Python. A program in Python is a script. To begin creating a script in IDLE, select New Window from the File dropdown menu, as shown in Figure A-9.

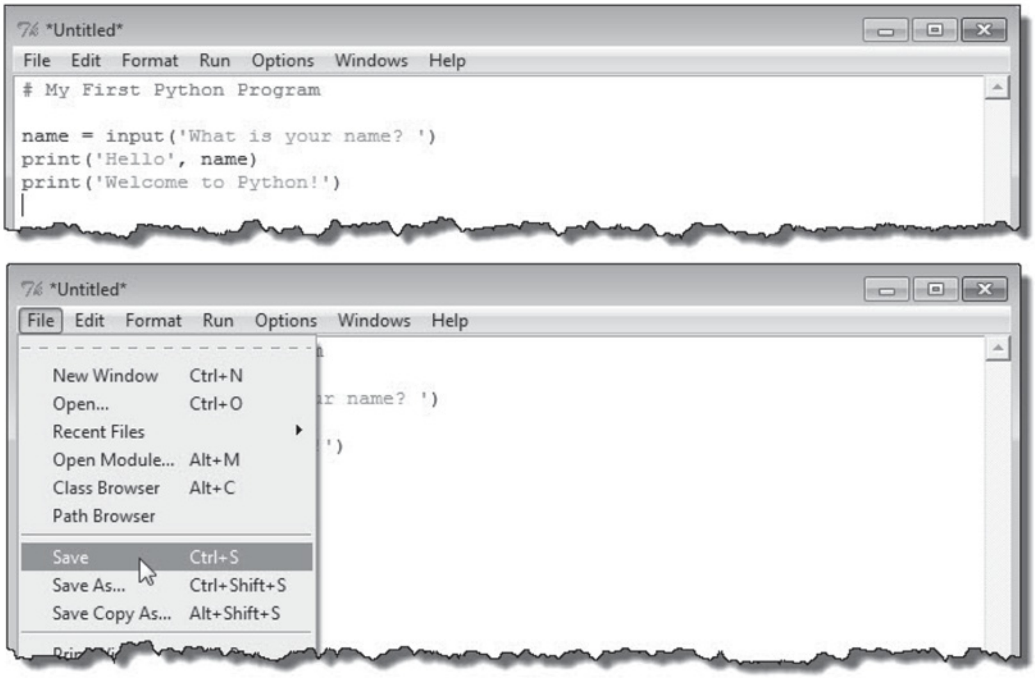
A Python program (or other text files) can be typed in this window. This program is saved by going to the File menu option and selecting Save (or by using shortcut key Ctrl+S), depicted in Figure A-10.

When selecting the Save option on a currently unnamed file, a file window appears allowing a name to be entered for the new program (Figure A-11). If the program was already named, choosing Save would save the file under the current name, overwriting the currently saved file. Using Save As allows a file to be named under a different name, such as for creating a backup file.


Python Shell
File Edit Shell Debug Options Windows Help
New Window Ctrl+N
Open... Ctrl+O
Recent Files
Open Module... Alt+M
Untitled
File Edit Format Run Options Windows Help

```
Python 3.2.2 (default, Sep  4 2011, 09:51:08) [MSC v.1500 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

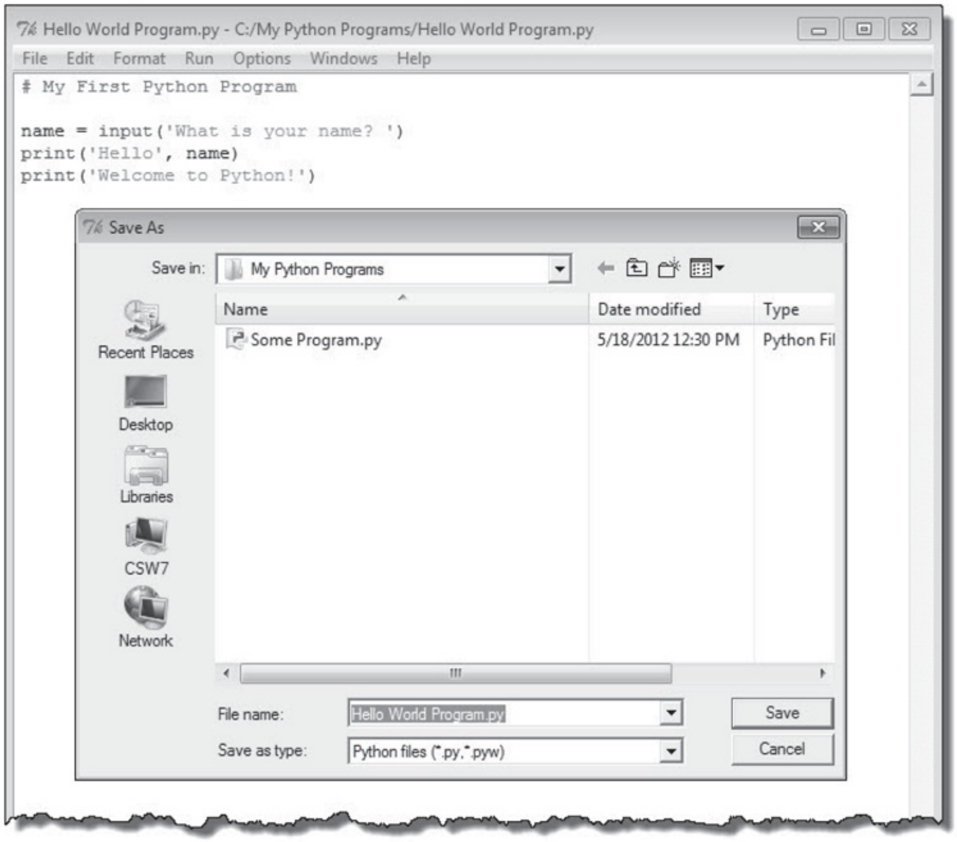
FIGURE A-9 Opening a Script Editor Window



```
# My First Python Program

name = input('What is your name? ')
print('Hello', name)
print('Welcome to Python!')
```

FIGURE A-10 Entering and Saving a Python Program File


Hello World Program.py - C:/My Python Programs/Hello World Program.py
File Edit Format Run Options Windows Help

```
# My First Python Program

name = input('What is your name? ')
print('Hello', name)
print('Welcome to Python!')
```

Save As
Save in: My Python Programs
Recent Places
Desktop
Libraries
CSW7
Network
Name
Date modified
Type
Some Program.py
5/18/2012 12:30 PM
Python Fil
File name: Hello World Program.py
Save as type: Python files (*.py,*.pyw)
Save
Cancel

FIGURE A-11 The File Window and Naming and Saving a File

When saving a Python program, *remember to add the file extension .py to the filename*. Otherwise, the saved file will not appear in the file window when working with Python programs in IDLE. It also will not be considered a Python file by the operating system. In this case, `Hello World Program.py` will become a new file in the `My Python Programs` folder in addition to the existing file `Some Program.py`.

When creating and modifying programs, there are a number of editing features that IDLE provides. Many of these commands will likely be familiar to you. However, in addition to the familiar commands (such as "cut" and "paste") there are editing features in IDLE that are specific to Python. These commands are listed in Figure A-12.

The backspace is for deleting characters *before* the cursor, and delete deletes characters *after*. Lines may be deleted, copied and pasted within a program. For navigating a program file, there is the ability to go to a specific line number, or search for lines containing a specific search string. Once a search has begun, the shortcut Ctrl+G provides a convenient way to continue the search to each next line found.

Windows Shortcut	Mac Shortcut	Action Performed
Backspace Key	Delete	Erases character before cursor
Delete Key	-	Erases character after cursor
Ctrl+C	⌘+C	Copies highlighted set of lines to be pasted
Ctrl+X	⌘+X	Cuts highlighted set of lines to be pasted
Ctrl+V	⌘+V	Pastes most recently copied or cut highlighted lines
Alt+G	⌘+J	Prompts for line number to place cursor
Ctrl+F	⌘+F	Prompts for characters to search for
Ctrl+G	⌘+G	Continues current search
Ctrl+0	Ctrl+0	Shows the innermost matching parentheses
Ctrl+Z	⌘+Z	Undo all previous editing actions one-by-one
Ctrl+]	⌘+]	Indents selected lines
Ctrl+[	⌘+[	Unindents selected lines back to the left
Alt+3	Ctrl+3	Comments out selected lines
Alt+4	Ctrl+4	Removes ## notations from start of selected lines
Ctrl+S	⌘+S	Saves program as current file name

FIGURE A-12 Editing Commands in IDLE

The Show Matching command (Ctrl+0) identifies certain matching delimiters. The innermost set of matching parentheses, square brackets or curly braces are highlighted from the current position of the cursor. This is useful for identifying mismatched parentheses in expressions, as well as mismatched delimiters in data structures containing parentheses, square brackets and/or curly braces. The Undo command (Ctrl+Z) will undo the most recent edit change in a file. This can be used to undo all the edit changes, even before the last time the file was saved, way back to the state of the file when it was first opened (or back to an empty file if being created).

One particularly useful pair of commands are the Indent / Dedent commands. These indent and “un-indent” a selected (highlighted) set of program lines. The number of spaces of indentation by default is four, which follows the Python convention for style. Therefore, it is recommended that you do not change this value. (It can be set under the Options / Configure IDLE menu selection.) Another useful set of commands is Comment Out and Uncomment. These commands add (and remove) a “double comment” symbol, ##, on a selected set of lines. This is useful for disabling certain portions of a program during program development and testing. A summary of some of the editing commands in IDLE is given in Figure A-12. The *shortcut keys given here are worth learning so that you can be more efficient in your programming*. Each command has a corresponding menu option.

We show all the menu options in the editor window of IDLE below. The **File menu** provides the ability to open, save and print files, shown in Figure A-13.

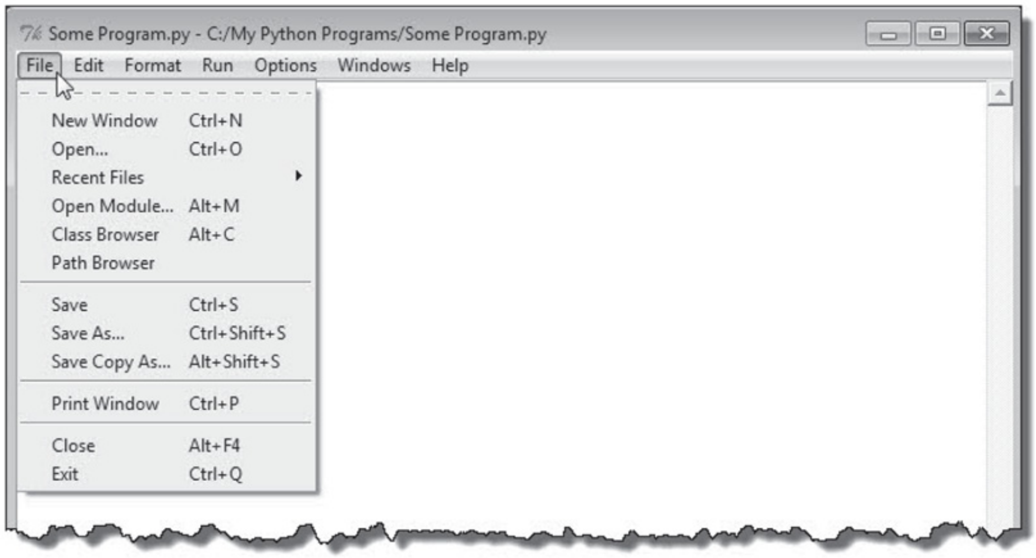


FIGURE A-13 The File Menu of the IDLE Editor Window

The **Edit menu** provides options for the usual editing commands such as Undo/Do, Copy/Cut/Paste, Find/Replace, shown in Figure A-14.

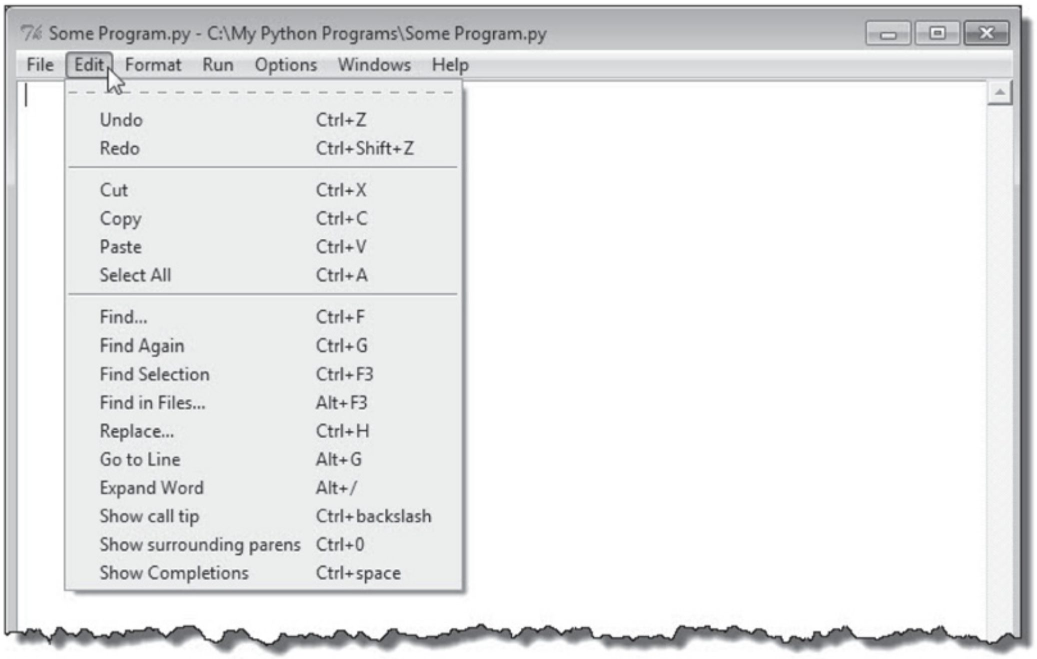


FIGURE A-14 The Edit Menu of the IDLE Editor Window

The **Format menu** provides options for Indenting / Dedenting, and Commenting Out / Uncommenting a section of code, shown in Figure A-15.

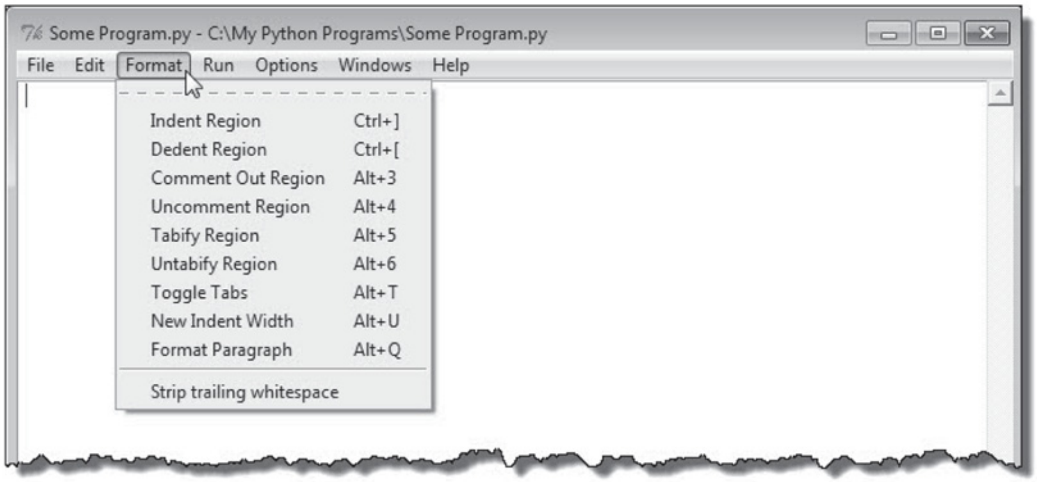


FIGURE A-15 The Format Menu of the IDLE Editor Window

The **Run menu** provides the Run Module, which executes the Python code currently in the editor window, shown in Figure A-16. *The F5 shortcut key is a convenient way to execute a program.*

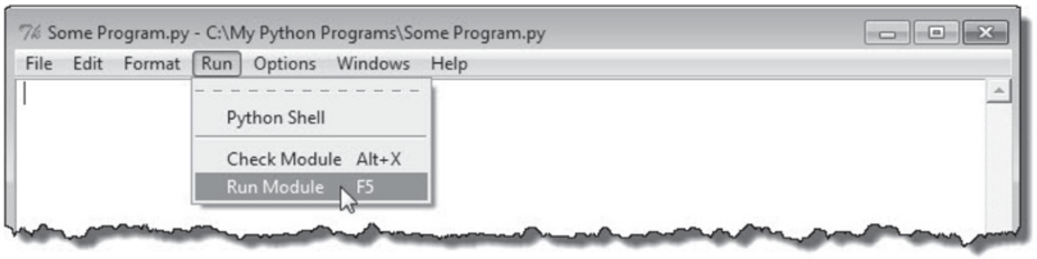


FIGURE A-16 The Run Menu of the IDLE Editor Window

The **Options menu** provides the option Configure IDLE for configuring various aspects of IDLE, including the fonts used, the color of keywords, the color of comment lines, etc., and the number of spaces used for each tab character (for indentation of program lines). Although there is a lot of control provided for the "look and feel" of IDLE, it is recommended to stick with the standard

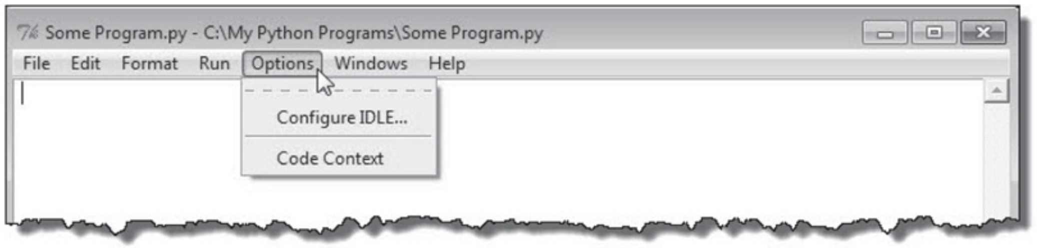


FIGURE A-17 The Options Menu of the IDLE Editor Window configuration options. (Code Content is a feature intended to help aid in keeping track of the lines in the same program block while scrolling through a file. We do not see the benefits of this option for our purposes.)

The **Windows menu** provides options for controlling the height of the window. It also provides a list of all currently open Python windows (by file name), including the Python Shell. This provides an easy way to switch from one window to another. The menu options are shown in Figure A-18.

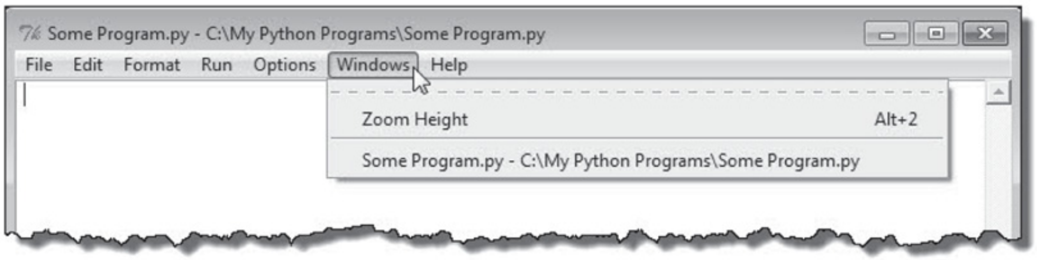

Some Program.py - C:\My Python Programs\Some Program.py
File Edit Format Run Options Windows Help
Zoom Height Alt+2
Some Program.py - C:\My Python Programs\Some Program.py

FIGURE A-18 The Windows Menu Options of the IDLE Editor Window

Finally, the **Help menu** includes About IDLE, which includes the version number of IDLE installed; and IDLE Help, which gives a brief summary of the commands of the menu bar.

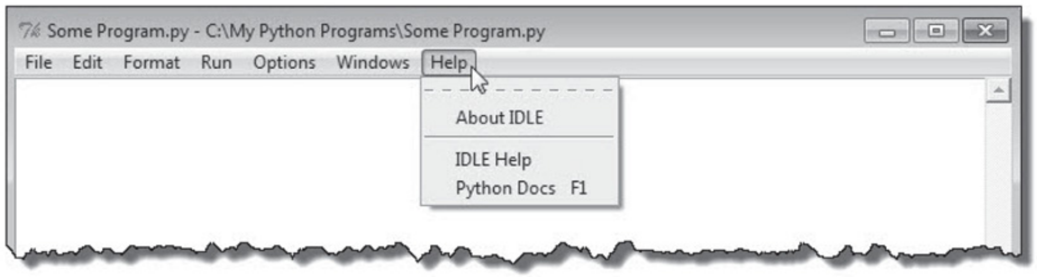
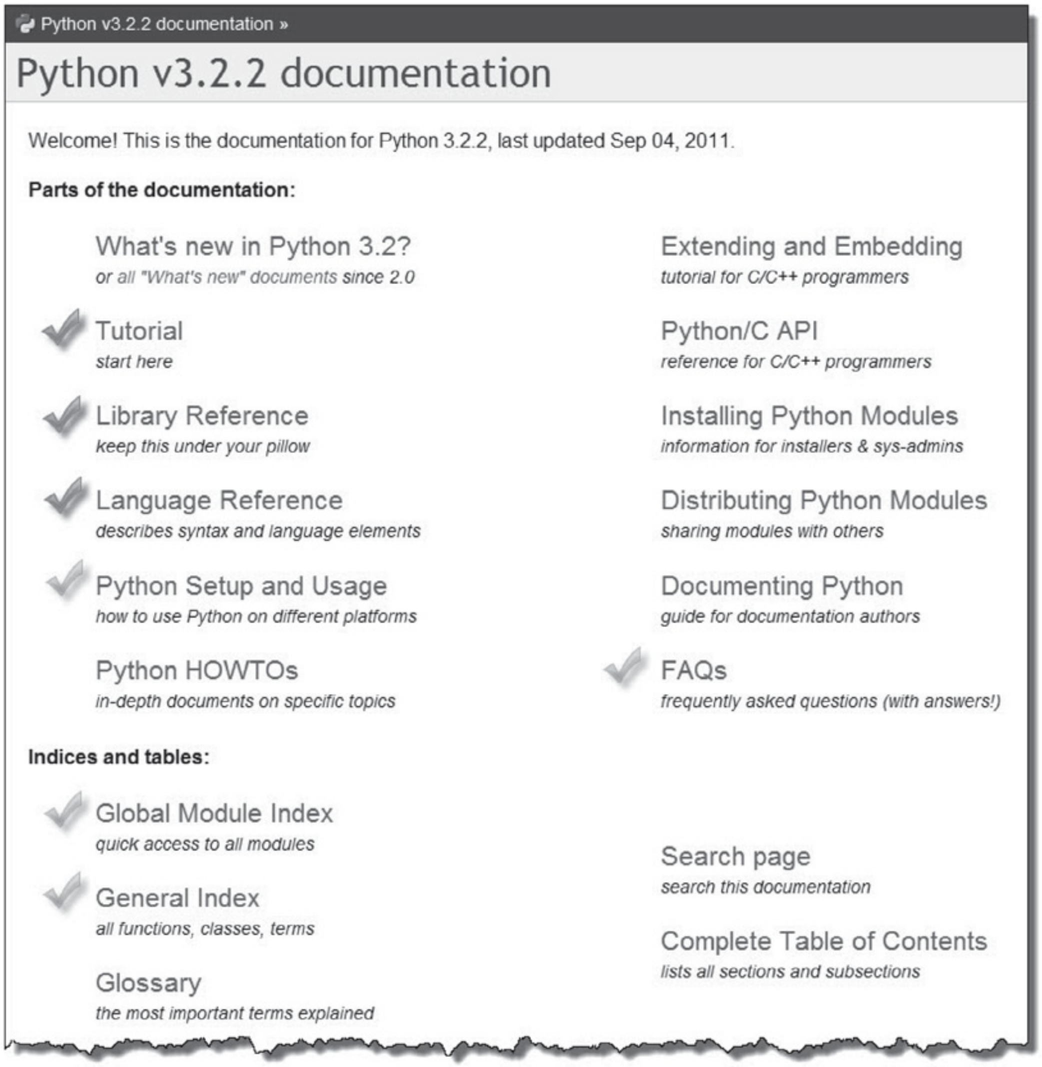

Some Program.py - C:\My Python Programs\Some Program.py
File Edit Format Run Options Windows Help
About IDLE
IDLE Help
Python Docs F1

FIGURE A-19 The Help Menu Options of the IDLE Editor Window

Using Python Docs from within IDLE A very useful feature of IDLE is the **Python Docs** option under the Help menu. This option links to the official documentation for the version of Python installed, shown in Figure A-20. The most relevant parts of the documentation for this text are marked with a (dark) checkmark. The Tutorial starts with an introduction to Python, and goes through all the features of the language, thus covering material beyond the scope of this text. It is a good starting point, however, for obtaining more information on a particular language feature. The Library Reference lists all the built-in functions, constants and types. In addition, it contains a categorized list of the Standard Library modules in Python. Finally, the Python Language Reference contains all information about the "core" of the language. This includes documentation on general syntax, expressions, statements, and compound statements (such as `if` and `while` statements). The Global Module Index, General Index and Python FAQs, indicated by lighter check marks, may also be of some help to the reader.



Python v3.2.2 documentation »

Python v3.2.2 documentation

Welcome! This is the documentation for Python 3.2.2, last updated Sep 04, 2011.

Parts of the documentation:

What's new in Python 3.2? *or all "What's new" documents since 2.0*	Extending and Embedding *tutorial for C/C++ programmers*
✓ Tutorial *start here*	Python/C API *reference for C/C++ programmers*
✓ Library Reference *keep this under your pillow*	Installing Python Modules *information for installers & sys-admins*
✓ Language Reference *describes syntax and language elements*	Distributing Python Modules *sharing modules with others*
✓ Python Setup and Usage *how to use Python on different platforms*	Documenting Python *guide for documentation authors*
Python HOWTOs *in-depth documents on specific topics*	✓ FAQs *frequently asked questions (with answers!)*

Indices and tables:

✓ Global Module Index *quick access to all modules*	Search page *search this documentation*
✓ General Index *all functions, classes, terms*	Complete Table of Contents *lists all sections and subsections*
Glossary *the most important terms explained*	

FIGURE A-20 Python Docs

A4. Common Python Programming Errors

We list the typical errors for novice programmers using Python. It would be wise to use the following as a checklist when developing and debugging your Python programs.

♦ Improper Indentation

All instructions in the same suite (block) must be indented the same amount. Each tab press will move the cursor the number of character spaces that is set under Options/Configure IDLE (four spaces by default, recommended).

- Forgetting About Truncated (Integer) vs. Real Division

  Keep in mind the difference between the / operator (5/4 → 1.25) and the // operator (5/4 → 1). This is very easy to forget when using arithmetic expressions.

- Confusing the Assignment Operator (=) with the Comparison Operator (==)

  This is a very common error for new programmers, but one that you should learn to avoid as early as possible.

- Forgetting to Convert String Values when Inputting Data

  Remember that the `input` function always returns a string type. It is easy to forget when reading in numeric values to convert them to integer or float before using arithmetically.

- Forgetting to Use Colons Everywhere Needed

  Don't forget to put colons where needed. Rather than trying to memorize where they are required, a simple rule can be followed. A colon is required after any keyword (such as `if` or `while`) in which the subsequent statements are indented.

- Forgetting about Zero-Based Indexing

  Zero-based indexing, in which the first index value of an indexed entity starts at 0, can lead to "off-by-one" errors if the programmer is not careful.

- Confusing Mutable and Immutable Types

  Remember that the value of a mutable type can be changed without the need for reassignment, for example, `list1.append(40)`. Since tuples and strings are immutable types and thus their values cannot be changed, reassignment of the variable is needed in order to "change" it. Thus, the statement `str1 + 'There!'` does not change the value of `str1`. To change its value, it must be reassigned, `str1 = str1 +' There!'`

- Forgetting the Syntax for Tuples of One Element

  Tuples are the only sequence type that requires a comma with tuples of only element, (1,) → (1). If the comma is left out, then the expression evaluates to that element, (1) → 1.

- Improperly Ended Program Lines

  Forgetting to use the backslash (\) when continuing a program line to the next line.

B. PYTHON QUICK REFERENCE

The references pages contained here summarize aspects of the Python programming language most relevant to the textbook. Therefore, the functions/operators listed, and the available optional arguments for each is not meant to be comprehensive. For complete coverage, see The Python Language Reference of the official Python site at http://docs.python.org/reference/index.html.

B1. Python Coding Style

Indentation

Use four spaces for each indentation level.

Blank Lines

Use blank lines, sparingly, to separate logical sections of code.
Separate function definitions with two blank lines. (Same for class definitions)

Line Length

Limit length of lines to 79 characters (i.e., do not wrap lines around screen).

Line Continuation

Statements containing open parentheses, square brackets or curly braces can be continued on the next line (except when containing open single or open double quotes):

```
result = (num1 + num2 + num3 + num4 + num5 +
          num6 + num7 + num8 + num9 + num10)
```

Statements without such delimiters can be continued by use of the line continuation character (\):

```
response = \
     int(input('(1)continue processing, (2)quit program '))
```

Use of Whitespace in Expressions and Statements

Assignment Statements…	**This:** `i = i + 1` **Not This:** `i=i+1`
Expressions…	**This:** `c = (a + b) * (a - b)` **Not This:** `c = (a+b) * (a-b)`
Comma-Separated Items…	**This:** `(a, b, c, d)` **Not This:** `(a,b,c,d)`

Keyword Arguments…	**This:**
	`def func1(real, imag=0.0):`
	Not This:
	`def func1(real, imag = 0):`

Function Calls…	**This:**
	`def func1(real, imag=0.0):`
	Not This:
	`def func1(real, imag = 0):`

B2. Python Naming Conventions

Symbolic Constants

All uppercase, underscores used when aids readability:

`RATIO, ANNUAL_RATE`

Variables

All lowercase, underscores used when aids readability:

`n, line_count`

Functions/Methods

All lowercase using underscores when needed, or mixed case ("camel case"), in which first character is lowercase, and first letter of all other words is uppercase:

`calcAverage, calc_average`

Classes

Mixed case, with first character a capital letter:

`VehicleClass`

Modules

Short name, all lowercase, underscores used when aids readability:

`math, conv_functions`

Keywords in Python

and	del	from	None	True
as	elif	global	nonlocal	try
assert	else	if	not	while
break	except	import	or	with
class	False	in	pass	yield
continue	finally	is	raise	
def	for	lambda	return	

B3. Comment Statements in Python

Syntax

```
# Comment line
```

Description

The hash sign (#) is used to make single line comments (when at the start of a line), or an in-line comment (when within a line). All characters following the hash sign until the end of line are treated as a comment.

Examples

(1) In-line comment, e.g.,

```
state_tax_rate = 0.08  # 2008 tax rate
```

(2) Single line comment serving as program section heading, e.g.,

```
# sum all values greater than 0
for i in range(0, len(values)):
   if values[i] > 0:
       sum = sum + values[i]
```

(3) As an alternative to use of a triple-quoted docstring, e.g.,

```
def function1(n):
#---------------------------------------------
# This function returns the largest integer
# less than or equal to n.
#---------------------------------------------
```

Never include an in-line comment that states what is obvious from the program line, e.g.,

```
n = n + 1   # increment n
```

But this is ok,

```
n = n + 1   # adjust n to avoid off-by-one error
```

Pragmatics

- If a # appears in a line with a prior #, then the second # is taken as part of the comment.
- There is no provision for block comments in Python (i.e., commenting out a sequence of lines with one set of comment delimiters).

B4. Literal Values in Python

Special Literal Value `None`

`None` is a special "place marker" in Python that can be used when there is no available value.

Numeric Literal Values

Numeric literals never contain commas…

```
Yes:   1200     204231
No:    1,200    204,231
```

Integer literals never contain a decimal point…

```
Yes:    1200
No:     1200.     1200.0
```

Range

- There is no limit for the size of an integer
- Floats are limited to 10^{-308} to 10^{308} with 16 to 17 digits of precision

Floating-point literals must contain a decimal point…

```
Yes:    1200.0    1200.
No:     1200
```

String Literal Values

Strings are delimited ("surrounded") by single or double quotes:

```
Examples: 'Hello World!'   "Hello World!"
```

Strings must be delimited by the same type of quote characters:

```
Yes:   'Hello World!'
No:    'Hello World!"
```

Strings Containing Quote Characters
Strings containing quotes must be delimited with the other quote type:

```
Yes:   "This is John's car"       'We all yelled "Hey John!"'
No:    'This is John's car'       "We all yelled "Hey John!""
```

Triple-Quoted Strings
Single and double-quoted strings must be contained on one line. Triple-quoted strings, however, can span more than one line. Triple-quoted strings may be denoted with three single quotes, or three double quote characters. These strings are mainly used as docstrings for program documentation.

Boolean Literal Values

There are only two literal values for the Boolean type:

```
Yes:  True     False
No:   true     false     TRUE     FALSE     'True'     'False'
```

B5. Arithmetic, Relational, and Boolean Operators in Python

Arithmetic Operators

Operator	Meaning
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`**`	Exponentiation
`/`	Float division
`//`	Truncated division
`%`	Modulus

Relational Operators

Operator	Meaning
`<`	Less than
`>`	Greater than
`<=`	Less than or equal to
`>=`	Greater than or equal to
`==`	Equal
`!=`	Not equal

Boolean Operators

Operator	Meaning
`and`	logical AND
`or`	logical OR
`not`	logical NOT

Usage

Given: `cond1 = True, cond2 = False`

```
2 + 3 - 1 → 4        2 < 3 → True          cond1 and cond2 → False
2 * 6 → 12           3 <= 2 → False        cond1 and not cond2 → True
2 ** 6 → 64          3 == 2 → False        cond1 or cond2 → True
3 / 2 → 1.5          3 != 2 → True         not cond1 or cond2 → False
3 // 2 → 1           'A' < 'B' → True      not(cond1 and cond2) → True
3.0 // 2 → 1.0       'a' < 'B' → False     (2 < 3) and cond1 → True
2025 // 100 → 20     'Hill' < 'Wu' → True  2025 % 100 → 25
```

Operator Precedence

highest

`**`	exponentiation
`*, /, //, %`	multiplication, division, modulus (remainder)
`+, -`	addition, subtraction
`<, <=, ... in, not in`	relational, membership, and identity operators
`not`	Boolean not operator
`and`	Boolean and operator
`or`	Boolean or operator

lowest

NOTE: All listed operators associate left-to-right, except `**` (which associates right-to-left).

B6. Built-in Types and Functions in Python

Built-in Types

Numeric Types

`int`	Integers
`float`	Floating point numbers
`complex`	Complex numbers
`bool`	Boolean values

Mapping Types

`dict`	Dictionary

Sequence Types

`str`	Strings
`list`	Lists
`tuple`	Tuples

Set Types

`set`	Sets
`frozenset`	Immutable sets

Built-in Functions

`abs(x)`
Returns the absolute value of a number (for arguments of type int, long, and float).

`chr(i)`
Returns a string of one character whose ASCII code is the integer `i`. The argument must be in the range [0…255] inclusive, otherwise a `ValueError` will be raised.

`cmp(x, y)`
Compare the two objects `x` and `y` and returns a negative value if `x<y`, zero if `x==y`, and a positive value if `x>y`.

`dict(arg)`
Creates a new dictionary where `arg` is a list of the form `[(attr1, value1), (attr2, value2), …]`.

`float(arg)`
Returns the absolute value of a number (for arguments of type int, long, and float).

`format(value, format_spec)`
Creates a formatted string from a provided sequence of format specifiers. (See details in Section B10)

`frozenset(arg)`
Creates a frozen (immutable) set where `arg` is a sequence (or other iterable type).

(continued …)

Built-in Functions

(…continued)

`help(arg)`
Used in interactive mode in the Python Shell. Displays a help page related to the string provided in `arg`. When an argument not provided, starts the built-in help system.

`id(object)`
Returns the "identity" of an `object`, which is an integer unique to all other objects, and remains the same during an object's lifetime.

`input(prompt)`
Prompts user with provided `prompt` argument to enter input, and returns the typed input as a string with the newline (\n) character removed. (See details in section B7)

`int(arg)`
Returns integer value of provided argument. Argument may be a string or an integer (in which case, the same integer value is returned).

`len(arg)`
Returns the length of `arg`, where `arg` may be a string, tuple, list, or dictionary.

`list(arg)`
Returns a list with items contained in `arg`, where `arg` may be a sequence (or other iterable type). If no argument given, returns an empty list.

`max(arg)`
Returns the largest value in a provided string, list or tuple.

`min(arg)`
Returns the smallest value in a provided string, list or tuple.

`open(filename, opt_mode)`
Opens and creates a file object for `filename`. Optional argument `opt_mode` either `'r'` (reading), `'w'` (writing), or `'a'` (appending). If omitted, file opened for reading.

`ord(str)`
Given a character (string of length one), returns the Unicode code point (character encoding value) for the character, e.g. `ord('a') = 97`.

(continued …)

Built-in Functions (…continued)

`pow(x, y)`
Returns `x` to the power `y`. Equivalent to `x**y`.

`print(arg1, arg2, ...)`
Prints arguments `arg1`, `arg2`, … on the screen. (See details in section B7)

`range(opt_start, stop, opt_step)`
Creates a list of integer values of a given progression. If `opt_start` not provided, then sequence begins at 0. If `opt_step` not provided, then increments by 1.

`repr(arg)`
Returns a string which is a printable representation of `arg`.

`reload(module)`
Reloads a given module that has already been loaded (imported). Useful for when making changes to modules when in interactive mode.

`round(x, opt_n)`
Returns the floating point value `x` rounded to `n` decimal places. If `n` is omitted, decimal place returned as 0.

`set(arg)`
Returns a new set with values in `arg`, where `arg` is a sequence (or other iterable type).

`sorted(arg)`
Returns a new sorted list with values in `arg`, where `arg` is a sequence (or other iterable type).

`str(arg)`
Returns a string version of `arg`. If `arg` is omitted, returns the empty string.

`tuple(arg)`
Returns a tuple whose items are the same as in `arg`, where `arg` may be a sequence (or other iterable type). If no argument is given, returns a new empty tuple.

`type(arg)`
Returns the type of a value (object) provided in `arg`.

B7. Standard Input and Output in Python

Standard Input

Standard input (`sys.stdin`) is a data stream used by a program to read data. By default, standard input is from the keyboard.

Standard Output

Standard output (`sys.stdout`) is a data stream used by a program to write data. By default, standard output goes to the screen.

Built-in Functions

`input(prompt)`
The `input` function sends (optional) string parameter, `prompt`, to the standard output (the screen) to prompt the user for input. It then returns the line read from the standard input (the keyboard) as a string, with the trailing newline character removed.

`print(value, ..., sep=' ', end='\n', file=sys.stdout)`
The `print` function sends values to the standard output (the screen). Multiple values may be given, each separated by commas. The displayed values are separated by a blank character and ended with a newline character. Each of the default parameter values, `sep`, `end` and `file`, may be changed as keyword arguments.

Examples

(1)
```
>>> name = input('Enter name: ')
Enter name: Audrey Smith
>>> name
'Audrey Smith'
```

(2)
```
>>> n = int(input('Enter age: '))
Enter age: 28
>>> n
28
```

(3)
```
>>> age = 38
>>> print('His age is', age))
His age is 38
```

(4)
```
>>> print(1, 2, 3, sep='*')
1*2*3
```

(5)
```
>>> print(1, 2, 3, end='...')
1 2 3...
```

Pragmatics

- The prompt string provided for the function `input` often is ended with a blank character to provide space between the end of the prompt and the typed user input.
- To keep the cursor on the same line when calling `print`, set default parameter `end` to the empty string as a keyword argument.
- The `input` function reads and returns any input by the user without generating an error. When the input is converted to another type, such as an integer in (2) above, the type conversion function used may raise an exception. Thus, exception handling is needed to check for invalid input.

B8. General Sequence Operations in Python

General Sequence Operations

`len(s)`
Returns the length of sequence `s`.

`s[i]` (*selection*)
Returns the item at index `i` in sequence `s`.

`s[i:j]` (*slice*)
Returns subsequence ("slice") of elements in `s` from index `i` to index `j - 1`.

`s[i:j:k]` (*slice with step*)
Returns subsequence ("slice") of every `k`th item in `s[i:j]`.

`s.count(x)`
Returns the number of occurrences of `x` in sequence `s`.

`s.index(x)`
Returns the first occurrence of `x` in sequence `s`.

`x in s` (*membership*)
Returns `True` if `x` is equal to one of the items in sequence `s`, otherwise returns `False`.

`x not in s` (*membership*)
Returns `True` if `x` is not equal to any of the items in sequence `s`, otherwise returns `False`.

`s1 + s2` (*concatenation*)
Returns the concatenation of sequence `s2` to sequence `s1` (of the same type).

`n * s` (or **`s * n`**)
Returns `n` (shallow) copies of sequence `s` concatenated.

`min(s)`
Returns the smallest item in sequence `s`.

`max(s)`
Returns the largest item in sequence `s`.

B9. String Operations in Python

String Operations

`str.find(arg) / str.index(arg)`
Both `find` and `index` return the lowest index matching substring provided. Method `find` returns `-1` if substring not found, method `index` returns `ValueError`.

`str.isalpha(arg) / str.isdigit(arg)`
Returns `True` if `str` non-empty and all characters in `str` are all letters (`isalpha`), or all digits (`isdigit`), otherwise returns `False`.

`str.isidentifier()`
Returns `True` is `str` is an identifier in Python, otherwise returns `False`.

`str.islower() / str.isupper() / str.lower() / str.upper()`
Methods `islower` / `isupper` return `True` if all letters in `str` are lower (upper) case, and there is at least one letter in `str`. Methods `lower` / `upper` return a copy of `str` with all letters in lower (upper) case.

`str.join(arg)`
Returns a string which is the concatenation of all strings in `arg`, where `arg` is a sequence (or some other iterable) type. If `arg` does not contain any strings, a `TypeError` is raised.

`str.partition(arg)`
For string separator in `arg`, returns a 3-tuple containing the substring before the separator in `str`, the separator itself, and the substring following the separator. If separator not found, returns a 3-tuple containing `str`, and two empty strings.

`str.replace(arg1, arg2)`
Returns a copy of `str` with all occurrences of `arg1` replaced by `arg2`.

`str.split(arg)`
Returns a list of words in `str` using `arg` as the delimiter string. If `arg` not provided, then whitespace (a blank space) is used as the delimiter.

`str.strip(arg)`
Returns a copy of `str` with leading and trailing chars contained in string `arg` removed. If `arg` not provided, then removes whitespace.

B10. String Formatting in Python

Syntax

`format(value, format specifier)`

Description

Built-in function `format` creates a formatted string from a provided sequence of format specifiers.

Format Strings

A format string may contain the following format specifiers in the order shown:
`fill_chr, align, width, precision, type`

Format Specifiers	Possible Values
fill_chr	Any character except `'{'` (requires a provided align specifier)
align	`'<'` (left-justified), `'>'` (right-justified), `'^'` (centered)
width	An integer (total field width)
precision	An integer (number of decimal places to display, rounded)
type	`'d'` (base 10), `'f'` (fixed point), `'e'` (exponential notation)

Examples

Given: `avg = 1.1275, sales = 143235 factor = 134.10456`

(1)
```
>>> print('Average rainfall in April:', format(avg,'.2f'), 'inches')
Average rainfall for April: 1.13 inches
```

(2)
```
>>> print('Yearly Sales $', format(sales, ','))
Yearly Sales $ 143,235
```

(3)
```
>>> print('Conversion factor:', format(factor, '.2e'))
Conversion factor: 1.34e+02
```

(4)
```
>>> print(format('Date', '^8'), format('Num Sold', '^12')
Date     Num Sold
```

(5)
```
>>> print('\n' + format('Date', '<8') + format('Num Sold', '^12') +
           '\n' + format('----', '<8') + format('--------', '^12'))

Date      Num Sold
----      --------
```

B11. Lists in Python

List Operations

`lst[i] = x`
Element `i` of `lst` replaced with `x` (for any object `x`).

`lst[i:j] = t`
Slice of `lst` from `i` to `j-1` replaced with sequence (or other iterable type) `t`.

`del lst[i:j]`
Removes slice `lst[i:j]` from `lst`.

`lst[i:j:k] = t`
Replaces items `lst[i:j:k]` in `lst` with `t`.

`del lst[i:j:k]`
Removes slice `lst[i:j:k]` from `lst`.

`s.append(x)`
Adds `x` to the end of sequence `s`.

`s.extend(x)`
Adds the *contents* of sequence `x` to the end of sequence `s`.

`s.insert(i, x))`
Inserts `x` at index `i` in sequence `s`.

`s.pop(i)`
Removes and returns `s[i]`. When argument `i` not provided, removes last item in `s`.

`s.remove(x)`
Removes and returns the *first* item in sequence `s` that equals `x`.

`s.reverse()`
Reverses the items in sequence `s`.

`s.sort()`
Sorts sequence `s` from smallest to largest.

B12. Dictionaries in Python

Dictionary Operations

`dict(arg)`
Returns a new dictionary initialized with items in `arg`, where `arg` may be a list, tuple or string (or other iterable type). If no argument provided, returns an empty dictionary.

`len(d)`
Returns the number of items in dictionary `d`.

`d[key]`
Returns item of dictionary d with key `key`.

`d[key] = value`
Sets `d[key]` to value.

`del d[key]`
Removes `d[key]` from dictionary `d`.

`key in d`
Returns `True` if `d` has key `key`, otherwise returns `False`.

`key not in d`
Returns `False` if `d` has key `key`, otherwise returns True.

`d.clear()`
Removes all items from dictionary.

`d.get(key)`
Returns the value for `key` in dictionary `d`.

`d.pop(key)`
Removes key/value pair in `d` for `key` and returns the associated value.

`d.popitem()`
Removes and returns an arbitrary key/value pair from dictionary `d`.

B13. Sets in Python

Set/Frozenset Operations

`set(arg) / frozenset(arg)`
Returns a new set or frozenset with items provided in `arg`, where `arg` is a sequence (or other iterable type).

`len(s)`
Returns the number of items in set `s`.

`x in s  (x not in s)`
Returns `True (False)` if `s` has item equal to `x`, otherwise returns `False  (True)`.

`s.add(x)` / **`s.remove(x), s.pop(), s.clear()`** (*Set type only*)
Adds / removes `x`, Removes-returns arbitrary item from `s`, Removes all items from `s`.

`s.isdisjoint(other)`
Returns `True` if set `s` had no items in common with the set provided by argument `other`, otherwise returns `False`.

`s.issubset(other)` (also **`set <= other`**)
Returns `True` if every item in `s` is also in set `other`, otherwise returns `False`.

`s.issuperset(other)` (also **`set >= other`**)
Returns `True` if every item in `other` is also in set `s`, otherwise returns `False`.

`set < other` (**`set > other`**)
Returns `True` if `s` is a proper subset (superset) of set `other`

`s.union(other, ...)`
Returns a new set containing all items in set `s` and all `other` provided sets.

`s.intersection(other, ...)`
Returns a new set containing all items common to set `s` and all `other` provided sets.

`s.difference(other, ...)`
Returns a new set containing all items in set `s` that are not also in `other` provided sets.

`s.symmetric_difference(other)`
Returns a new set containing items that are in set `s` or set `other`, but not both.

B14. if Statements in Python

Syntax

```
if condition:
     statements
elif condition:
     statements
else:
     statements
```

Description

An if statement may have zero or more `elif` clauses, optionally followed by an `else` clause. As soon as the condition of a given clause is found true, that clause's statements are executed, and the rest of the clauses are skipped. If none of the clauses are found true, then the statements of the `else` clause are executed (if present). As with all data structures, `if` statements may be nested.

Examples

(1)
```
if n == 0:
     print('n is zero')
```

(2)
```
if n == 0:
    print('n is zero')
else:
    print('n is non-zero')
```

(3)
```
if n < 0:
     print('n is less than 0')
elif n == 0:
     print('n is zero')
else:
     print('n is greater than zero')
```

(4)
```
if n1 == 0:
     print('n1 is zero')
if n2 == 0:
     print('n2 is zero')
```

(5)
```
if n1 == 0:
     print('n1 is zero')
     if n2 == 0:
          print('n2 is zero')
     else:
          print 'n2 is not zero'
else:
     print 'n1 is not zero'
```

Pragmatics

- It is not required to include `elif` or `else` clauses in `if` statements.
- When selecting among a set mutually exclusive conditional expressions, the `if/elif` (with optional else) form of `if` statement should be used.

Typical Errors

1. Forgetting to add a semicolon (`:`) after each conditional expression, and after keyword `else`.
2. Improper operator use when checking for equality (e.g., "==", not "=").
3. Improper Boolean expression evaluation resulting from unconsidered operator precedence.
4. Improper indentation of nested `if/elif/else` clause(s) resulting in faulty logic.

B15. for Statements in Python

Syntax

```
for k in sequence:
    statements
```

Description

The `for` statement is used to control a loop that iterates once for each element in a specified sequence of elements such as a list, string or other iterable type.

Examples

Given: `empty_str = '',  space = ' '`

(1)
```
for num in [2, 4, 6, 8]:
    print(num, end=space)
```
Output: `2 4 6 8`

(2)
```
for num in range(2, 10, 2):
    print(num, end=space)
```
Output: `2 4 6 8`

(3)
```
for num in range(10, 2, -2):
    print(num, end=space)
```
Output: `10 8 6 4`

(4)
```
for char in 'Hello':
    print(char, end=empty_str)
```
Output: `Hello`

(5)
```
lst = [2, 4, 6, 8]
for k in range(len(lst)):
    lst[k] = lst[k] + 1

print(lst)
```
Output: `3 5 7 9`

(6)
```
t = [[1, 2, 3],[4, 5, 6]]
for i in range(len(t)):
    for j in range(len(t[i])):
        print(t[i][j], end=space)
    print()
```
Output: `1 2 3`

Pragmatics

- When the elements of a sequence need to be accessed but not altered, a `for` loop that iterates over the values in the sequence is the appropriate approach (1-4).
- When the elements of a sequence need to be both accessed and updated, a `for` loop that iterates over the index values in the sequence is the appropriate approach (5,6).
- Nested `for` loops can be used to iterate through a list of sequence types (6).

Typical Errors

1. Forgetting to add a semicolon (`:`) after the loop header.
2. Improper indentation of statements contained in the body of the `for` loop.
3. Forgetting that `range(i,j)` (and `range(j)`) generates values up to, but not including `j`.

B16. while Statements in Python

Syntax

```
while condition:
    statements
```

Description

The `statements` contained in a while loop body continue to execute until the `condition` evaluates to `False`.

Examples

Given: `empty_str = '', space = ' '`

(1)
```
i = 1

while i < 10:
    print(i, end=space)
    i = i + 1
```
Output: `1 2 3 4 5 6 7 8 9 10`

(2)
```
i = 10

while i != 0:
    print(i, end=space)
    i = i - 1
```
Output: `10 9 8 7 6 5 4 3 2 1`

(3)
```
while True:
    print('This loop keeps running!')
```
Output:
```
This loop keeps running!
This loop keeps running!
etc.
```

(4)
```
file = open('filename.txt', 'r')
line = file.readline()

while line != empty_str:
    line = file.readline()
```

(5)
```
num = int(input('Enter a positive number: '))

while num < 0:
    print('Your input was invalid.')
    num = int(input('Enter a positive number: '))
```

Pragmatics

- A `while` loop can be used to implement any kind of loop, although some `while` loops are more conveniently implemented as a `for` loop (1,2).
- An *infinite loop* is caused by a condition that always evaluates to `True` (3).
- A loop to execute an indefinite number of times, such as when reading from a file, must be implemented as a `while` loop (4,5).
- A `while` loop is used when accepting and validating user input (5)

Typical Errors

1. Forgetting to add a semicolon (`:`) after the conditional expression.
2. Not making progress in the body, thereby creating a loop that never ends (i.e. an infinite loop).
3. Improper indentation of statements contained in the body of the `while` loop.

B17. Functions in Python

Description

A function is a named group of instructions accomplishing some task. A function is *invoked* (called) by providing its name, followed by a (possibly empty) list of arguments in parentheses. Calls to value-returning functions are expressions that evaluate to the returned function value. Calls to non-value returning functions are effectively statements called for their side effects. (Strictly speaking, they are value-returning functions since they return special value `None`).

Syntax

Value returning functions

```
def name(parameters):
    statements
    return expression
```

Non-value returning functions

```
def name(parameters):
    statements
```

Examples

Given: `num1 = 10, num2 = 25, num3 = 35, list1 = [1, 0, 3, 8, 0]`

(1)
```
def avg3(n1, n2, n3):
    '''Returns rounded avg.'''

    return round((n1 + n2 + n3) / 3)

>>> avg3(num1, num2, num3)
23
```

(2)
```
def dashLine(len, head=''):
    empty_str = ''
    space = ' '

    if head != empty_str:
        head = space + title
                 + space
    print(head.center(len,'-'))

>>> dashLine(16)
----------------
>>> dashLine(16, head='price')
---- price -----
```

(3)
```
def countdown(from, to):
    for i in range(from, to-1, -1):
        print(i, end='...')

>>> countdown(from=5, to=1)
5...4...3...2...1...
```

(4)
```
def hello():
    print('Hello World!')

>>> hello()
Hello World!
```

(5)
```
def removeZeros(lst):
    for k in range(len(lst)-1, -1, -1):
        if lst[k] == 0:
            del lst[k]

>>> list1
[1, 0, 3, 8, 0]

>>> removeZeros(list1)
[1, 3, 8]
```

Pragmatics

- A triple-quoted string as the first line of a function serves as its docstring (1).
- Functions may define default arguments (2), and be called with keyword arguments (3).
- A function may be defined having no parameters (4).
- Mutable arguments passed to a function can become altered (5).

B18. Classes in Python

Syntax

```
class classname(parentclass):

    def __init__(self, args):
       '''docstring'''
          .
          .
    def methodname(args):
       '''docstring'''
          .
          .
    def methodname(args):
       '''docstring'''
```

Description

A class consists of a set of methods and instance variables. The instances variables are created in special method `init`. The `init` method must have an extra first parameter, by convention named `self`, that is not passed any arguments when the method is called.

A *docstring* is a single or multi-line string using triple quotes that provides documentation for methods, classes and modules in Python.

Example

```
class XYCoord(object):

    def __init__(self, x, y):
        self.__x = x
        self.__y = y

    def getX(self):
        return self.__x

    def setX(self, x):
        self.__x = x
      .
      .
    def __repr__(self):

        return '(' + str(self.__x) + ',' + \
                     str(self.__y) + ')'


    def __eq__(self, xycoord):

        return self.__x == xycoord.getX() and \

               self.__y == xycoord.getY()
```

Special Methods

```
__init__
__repr__
__str__
__neg__
__add__
__sub__
__mul__
__truediv__
__floordiv__
__mod__
__pow__
__lt__
__le__
__eq__
__ne__
__gt__
__ge__
```

Private Members ("name mangling")

Instance variables beginning with two underscores are treated as private. They are accessible only if the mangled form of the name is used: with a single underscore followed by the class name added to the front. For example, private instance variable `__x` in class `XYCoord` becomes `_XYCoord__x`.

B19. Objects in Python

Syntax

identifier = *classname*(*args*)

Description

An object is an instance of a class. All values in Python are objects.

Examples

(1)	`loc1 = XYCoord(5,10)`	Creates a new `XYCoord` object, its reference assigned to `loc1`
(2)	`loc2 = XYCoord(5,10)`	Creates a new `XYCoord` object, its reference assigned to `loc2`
(3)	`loc1`	`(5,10)`, dereferenced value of identifier `loc1`
(4)	`loc2`	`(5,10)`, dereferenced value of identifier `loc2`
(5)	`id(loc1)`	`37349072`, reference value of identifier `loc1`
(6)	`id(loc2)`	`37419120`, reference value of identifier `loc2`
(7)	`loc1 == loc2`	`True`, comparison of their dereferenced values
(8)	`loc1 is loc2`	`False`, comparison their reference values
(9)	`loc3 = loc1`	Assigns reference value of `loc1` to `loc3`
(10)	`loc3 == loc1`	`True`
(11)	`loc3 is loc1`	`True`
(12)	`loc3.setX(10,10)`	Changes value of `loc3` to `(10,10)`
(13)	`loc3 == loc1`	`False`

Pragmatics

- When assigning an object to an identifier, the reference to the object is assigned, not the object itself. Thus, more than one identifier may reference the same object.
- Each newly-created object has a unique id. For variables `var1` and `var2`, if `id(var1)` equals `id(var2)` (or `var1 is var2` is `True`) they are referencing the same object.
- Mutable objects (e.g., lists) can be altered without reassignment:

```
>>> lst = [1, 2, 3]
>>> lst.append(4)
>>> lst
[1, 2, 3, 4]
```

- Immutable objects (e.g., strings) cannot be altered without reassignment:

```
>>> str1 = 'Hello'
>>> str1.replace('H', 'J')
>>> str1
'Hello'
>>> str1 = str1.replace('H', 'J')
>>> str1
'Jello'
```

B20. Exception Handling in Python

Syntax

```
try:
    statements
except ExceptionType:
    statements
except ExceptionType:
    statements
etc.
```

Description

Exception handling provides a means for functions and methods to report errors that cannot be corrected locally. In such cases, an *exception* (object) is *raised* that can be *caught* by its *client code* (the code that called it), or the client's client code, etc., until *handled* (i.e. the exception is caught and the error appropriately dealt with). If an exception is thrown back to the top-level code and never caught, then the program terminates displaying the exception type that occurred.

Example

```
def genAsterisks(x):
    """x an integer in range 1-79."""

    if x < 1 or x > 79:
        raise ValueError('Must enter 1-79')

    return x * '*'

#---- main
valid_input = False

while not valid_input:
    try:
        num = int(input("How Many '*'s?: "))
        print(genAsterisks(num))
        valid_input = True

    except ValueError as errorMesg:
        print(errorMesg)
```

Standard Exceptions

```
EOFError
FloatingPointError
ImportError
IOError
IndentationError
IndexError
KeyError
NameError
OverflowError
RuntimeError
SyntaxError
TypeError
ValueError
ZeroDivisionError
```

Pragmatics

- The `exceptions` built-in module is automatically imported in Python.
- Built-in exceptions, raised by the built-in functions/methods, may also be raised by user code.
- New exceptions may be defined as a subclass of the built-in `Exception` class.
- There may be any number of `except` clauses for a given `try` block.
- In addition to the `except` clauses, an `else` clause may optionally follow, only executed if the `try` block does not raise any exceptions. Following that, an optional `finally` clause, if present, is always executed, regardless of whether an exception had been raised or not. The `else` and `finally` clauses are often used for resource allocation, such as closing an open file.

B21. Text Files in Python

Description

A *text file* is a file containing characters, structured as lines of text. Text files can be directly created and viewed using a text editor. In addition to printable characters, text files also contain *non-printing* newline characters, `\n`, to denote the end of each line of text.

Using Input Files

A file that is open for input can be read from, but not written to.

File Operations	Action
fileref = `open(filename, 'r')`	Opens and creates a file object for reading `filename`. Raises an `IOError` exception if file not found.
fileref`.readline()`	Reads next line of file. Returns empty string if at end of file. Includes newline character (`'\n'`) in line read.
fileref`.close()`	Closes file. File can be reopened to read from first line.

Example Program

```
filename = input('Enter filename: ')
inFile = open(filename, 'r')

line = inFile.readline()
while line != '':
    print(line, end='')
    line = inFile.readline()
```

Program Execution

```
Enter filename: testfile.txt
Hi,

This is a test file.
Containing five lines.
Including one blank line.
>>>
```

Using Output Files

A file that is open for output can be written to, but not read from.

File Operations	Action
fileref = `open(filename, 'w')`	Opens and creates a file object for writing to `filename`.
fileref`.`*write(s)*	Writes string s to file. Does *not* include output of `'\n'`.
fileref`.close()`	Closes file. If not closed, last part of output may be lost.

Example Program

```
filename = input('Enter filename: ')
outFile = open(filename, 'w')

line = input('Enter line of text:')
while line != '':
    outFile.write(line + '\n')
    line = input('Enter line:')

outFile.close()
```

Program Execution

```
Enter filename: newfile.txt
This is the first entered line.
This is the second entered line.
This is the last entered line.
```

Contents of file `newfile.txt`

```
This is the first entered line.
This is the second entered line.
This is the last entered line.
```

B22. Modules in Python

Description

A Python module is a file containing Python definitions and statements. The module that is directly executed to start a Python program is called the *main module*. Python provides standard (built-in) modules in the Python Standard Library.

Module Namespaces

Each module in Python has its own *namespace*: a named context for its set of identifiers. The *fully qualified* name of each identifier in a module is of the form `modulename.identifier`.

Forms of Import

(1) `import modulename`

Makes the namespace of `modulename` available, but not part of, the importing module. All imported identifiers used in the importing module must be fully qualified:

```
import math
print('factorial of 16 = ', math.factorial(16))
```

(2) `from modulename import identifier_1, identifier_2, ...`

`identifier_1`, `identifier_2`, etc. become part of the importing module's namespace:

```
from math import factorial
print('factorial of 16 = ', factorial(16))
```

(3) `from modulename import identifier_1 as identifier_2`

`identifier_1` becomes part of the importing module's namespace as `identifier_2`

```
from math import factorial as fact
print('factorial of 16 = ', fact(16))
```

(4) `from modulename import *`

All identifiers of `modulename` become part of the importing module's namespace (except those beginning with an underscore, which are treated as private).

```
from math import *
print('factorial of 16 = ', fact(16))
print('area of circle = ', pi*(radius**2))
```

Pragmatics

Although the `import *` form of `import` is convenient in that the imported identifiers do not have to be fully qualified in the importing module, there is the risk of a name clash. In addition, it is not apparent which identifiers are imported, or which module they are imported from.

C. PYTHON STANDARD LIBRARY MODULES

These pages contain selected modules of the Python Standard Library. For a complete listing, see the official standard library at http://docs.python.org/reference/index.html.

The Module Search Path Python modules may be stored in various locations on a particular system. For this reason, when a module is imported, it must be searched for. The interpreter first searches for a built-in (standard) module with that name. If not found, it then searches for the module in the same directory as the executed program. If still not found, the interpreter searches in a list of directories contained in the variable `sys.path`, a variable of the built-in module `sys`. The `sys` module must be imported to access this variable:

```
>>> import sys
>>> sys.path
['C:\\Python32\\Lib\\idlelib','C:\\Windows\\system32\\python32.zip',
'C:\\Python32\\DLLs','C:\\Python32\\lib','C:\\Python32',
'C:\\Python32\\lib\\site-packages']
```

The particular value for `sys.path` depends on your particular Python installation.

The dir Built-In Function Built-in function `dir` can be used to find out the names that a particular module defines. This may be used on any module:

```
>>> import math
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'hypot', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf', 'pi',
'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Note that in addition to the available mathematical methods, there are three special identifers `__doc__`, `__name__` and `__package__` listed. The first two provide the docstring (providing brief documentation of the module's contents) and the module name, respectively:

```
>>> print(math.__doc__)
This module is always available. It provides access to the
mathematical functions defined by the C standard.

>>> print(math.__name__)
math
```

The `__package__`special identifier is used for modules that contain submodules, called packages.

C1. The math Module

Description

This module contains a set of commonly-used mathematical functions, including number-theoretic functions (such as factorial); logarithmic and power functions; trigonometric (and hyperbolic) functions; angular conversion functions (degree/radians); and some special functions and constants (including `pi` and `e`). A selected set of function from the `math` module are presented here.

Number-Theoretic Functions

`math.ceil`	returns the ceiling of `x` (smallest integer greater than or equal to `x`).
`math.fabs(x)`	returns the absolute value of `x`.
`math.factorial(x)`	returns the factorial of `x`.
`math.floor()`	returns the floor of `x` (largest integer less than `x`).
`math.fsum(s)`	returns an accurate floating-point sum of values in `s` (or other iterable).
`math.modf()`	returns the fractional and integer parts of `x`.
`math.trunc(X)`	returns the truncated value of `s`.

Power and Logarithmic Functions

`math.exp(x)`	returns `e**x`, for natural log base `e`.
`math.log(x,base)`	returns log `x` for `base`. If `base` omitted, returns `log x base e`.
`math.sqrt(x)`	returns the square root of `x`.

Trigonomeric Functions

`math.cos(x)`	returns cosine of `x` radians.
`math.sin(x)`	returns sine of `x` radians.
`math.tan(x)`	returns tangent of `x` radians.
`math.acos(x)`	returns arc cosine of `x` radians.
`math.asin(x)`	returns arc sine of `x` radians.
`math.atan(x)`	returns arc cosine of `x` radians.

Angular Conversion Functions

`math.degrees(x)`	returns `x` radians to degrees.
`math.radians(x)`	returns `x` degrees to radians.

Mathematical Constants

`math.pi`	mathematical constant `pi = 3.141592` …
`math.e`	mathematical constant `e = 2.718281` …

C2. The random Module

Description

This module provides a pseudorandom number generator using the Mersenne Twister algorithm, allowing users to generate random numbers with near uniform distribution over a long period.

Key Functions

`random.random()`	returns random float value x, where 0 <= x < 1.
`random.uniform(a, b)`	returns random float value x, where a <= x <= b.
`random.randint(a, b)`	returns random integer value x, where a <= x <= b.

`random.randrange(start, stop, step)`
returns random integer value x, where `start <= x < stop` and `x = start + n * step`, where n is an integer greater than or equal to zero

`random.choice(seq)`	returns random element from sequence `seq` (must be non-empty).
`random.shuffle(seq)`	randomly reorders sequence `seq` in place.
`random.sample(seq, k)`	returns list (length `k`) of unique items randomly chosen from `seq`.

Examples

Given: `list = ['a', 'b', 'c', 'd', 'e']`

(1) Generate random float
```
>>> random.random()
0.8568285775611655
```

(2) Generate random float in range
```
>>> random.uniform(1, 10)
9.71538746497116
```

(3) Generate random integer in range
```
>>> random.randint(1, 10)
6
```

(4) Select random item from a list
```
>>> random.choice(list)
'e'
```

(5) Generate random integer in range with step
```
>>> random.randrange(0, 10, 2)
6
>>> random.randrange(0, 10, 2)
8
>>> random.randrange(0, 10, 2)
2
```

(6) Randomly select subset from a list
```
>>> random.sample(list, 3)
['d', 'c', 'e']
```

(7) Randomly reorder a list
```
>>> random.shuffle(list)
>>> print(list)
['c', 'b', 'e', 'a', 'd']
```

Pragmatics

- The `random` module must be imported before it can be used.
- The current system time is used to initialize the random number generator. If comparability and reproducibility are important, supply a seed value `x` by invoking `random.seed(x)` before generating any random numbers.
- To generate only even numbers, use the `randrange` function with `start` of `0`, `step` of `2`.

C3. The turtle Module

Description

This module provides both procedure-oriented and object-oriented ways of controlling *turtle graphics*. A turtle is a graphical entity in an x/y coordinate plane (Turtle screen) that can be controlled in various ways including: its shape, size, color, position, movement (relative and absolute), speed, visibility (show/hide), and drawing status (pen up/pen down).

Creating a Turtle Graphics Window

```
# set screen size
turtle.setup(800, 600)

# get reference to turtle screen
screen = turtle.Screen()

# set window title bar
screen.title('My Turtles')
```

Creating a Turtle

```
# getting the default turtle
t = turtle.getturtle()

# creating a new turtle
t = turtle.Turtle()
```

Setting Screen Attributes

`screen.bgcolor(args)`	background color, specified by name, (RGB) or hex
`screen.bgpic(filename)`	sets GIF file as screen background
`screen,clear()`	clears the screen
`screen.reset()`	resets all turtles to their initial state
`screen.bye()`	closes turtle screen window
`screen.exitonclick()`	closes turtle screen window on mouse click

Setting Turtle Appearance

`showturtle() / hideturtle()`	makes turtle visible / invisible
`turtle.register_shape(filename)`	GIF file name, registers image for use as turtle shape
`shape(arg)`	sets turtle to regular or registered shape

Getting Turtle's State

`turtle.isvisible()`	returns `True` if turtle currently visible
`turtle.position()`	returns current position of turtle as `x,y` coordinate
`turtle.towards(x,y)`	returns angle between turtle and coordinate `x,y`
`turtle.xcor()`	returns turtle's current `x` coordinate
`turtle.ycor()`	returns turtle's current `y` coordinate
`turtle.heading()`	returns turtle's current heading
`turtle.distance(arg)`	returns distance to (`x,y`) or to another turtle

Pen Control

`turtle.pendown()`	puts turtle's pen down so draws when it moves
`turtle.penup()`	lift's turtle's pen so doesn't draw
`turtle.isdown()`	returns `True` if pen currently down
`turtle.pencolor(color)`	sets pen's color by color name or (RGB)
`turtle.pensize(size)`	sets pen's line thickness, `size` a positive number
`turtle.fillcolor(color)`	sets pen's fill color by color name or (RGB)
`turtle.clear()`	deletes turtle's drawing from screen
`turtle.write(arg)`	writes text representation of `arg` at turtle's location

Controlling Turtle

`turtle.forward(x)/backward(x)`	moves turtle forward/backward x pixels
`turtle/right(angle)/left(x)`	moves turtle left/right by `angle`
`turtle.goto(x,y)`	moves turtle to coordinate (`x,y`)
`turtle.setx(x)/sety(y)`	moves turtle to `x`/`y` locations
`turtle.setheading(angle)`	sets heading of turtle by `angle`
`turtle.undo()`	undoes last turtle action
`tilt(angle)`	rotates turtle relative to its current tile-angle
`settiltangle(angle)`	sets turtle to `angle`
`tileangle()`	returns the turtle's current tilt angle
`turtle.speed(speed)`	1-slowest, 2-faster, … 10-fast, 0-fastest
`turtle.home()`	moves turtle to original position
`turtle.circle(radius)`	draws a circle if size `radius`
`turtle.dot(size, color)`	draws a dot with given `size` and `color`
`turtle.stamp()`	stamps turtle shape screen, return unique stamp id
`turtle.clearstamp(id)`	clears turtle stamp with provided `id`
`turtle.clearstamps(n)`	clears last `n` stamps, if argument omitted, clears all

Event Handling

`turtle.onclick(func)`	`func` a function of two arguments called with the x,y coordinates of the location of mouse click
`turtle.onrelease(func)`	`func` a function of two arguments called with the x,y coordinates of the location of mouse release
`turtle.ondrag(func)`	`func` a function of two arguments called with the x,y coordinates of the location of mouse click when mouse dragged
`turtle.mainloop()`	starts event loop. Must be last statement in a turtle graphics program

C4. The webbrowser Module

Description

This module provides functionality to open Web-based documents in a browser from within a Python program. The module launches the system's default browser to open the target URL, unless a different (supported) browser is specified.

Key Functions

`webbrowser.open(`*`url, new=0, autoraise=True`*`)`
Opens the page at the specified `url` in a browser window. If possible, a value of 0 for `new` opens the page in an existing browser window, 1 in a new browser window, and 2 in a new browser "tab." When `autoraise` is passed as True, the browser window is raised to the top of the stack (this may happen on some systems regardless of `autoraise` value).

`webbrowser.open_new(url)` / **`webbrowser.open_new_tab(url)`**
Always opens the specified `url` in a new browser window or tab, respectively.

`webbrowser.get(name)`
Returns a controller object for the specified browser type `name`, allowing the user to use any browser that is registered with or predefined in the module.

Examples

(1) Open a page in the browser
```
import webbrowser
webbrowser.open('http://docs.python.org/, 0, True')
```

(2) Open a page in the browser
```
import webbrowser
webbrowser.open_new('http://docs.python.org/')
```

(3) Specify browser to open page
```
import webbrowser
browser = webbrowser.get('firefox')
browser.open('http://docs.python.org/')
```

Pragmatics

- Python scripts can be written to generate HTML code, which can then be opened in a browser as part of the program using this module.
- This module is not limited to opening Web pages. On some platforms, it can also be used to open documents and media files. Instead of passing an http URL to `open`, pass the address of the target file (e.g. `file://`*`host`*`/`*`path`* or, on local machine, `file:///`*`path`*).
- The functionalities of this module are also available through the command line.