**How to use the interactive shell:**

* To start IDLE, use the features of your operating system. This opens an interactive shell.
* To close the interactive shell, click on its close button or select File 🡪 Close.
* To restart an interactive shell, select Run🡪Python Shell from another IDLE window.
* Enter Python code after the >>> prompt. Then, press Enter.

Open Python interactive shell and type at prompt:

print(“Hello World!”)

print()

1. + 5

30 - 12

3 \* 4

12 / 3

1 / 3

3 \* 4 + 30

3 \* (4 + 30)

X=5

X+10

Python shell traceback the most recent line, therefore typing

X+15 next will return an error.

**How to compile and run a program**

When using an IDLE, you can just press the F5 key.

If you are using Spider, you can press F5 as well or you can click Run on the menu and then Run again. You can also click the green arrowhead in the menu bar.

**How to fix syntax and runtime errors**

After you write the code for a program, you need to test the program. When you do that, your goal is to find all of the errors (or bugs) in the program. Whenever you find a bug, you need to debug it by fixing it. Then, you need to test the program again, fix the next bug that’s found, and continue this process until you finish debugging the entire program.

When you’re ready to test a program and you try to compile and run it, two types of errors are likely to be detected.

* The first type is a ***syntax error***. These are errors that occur because the Python coding rules have been violated. As a result, the program can’t be compiled.
* The other type of error is a ***runtime error***. This type occurs after all the code has compiled cleanly and the program is being executed. Then, an error message is displayed that helps you identify the cause of the error. This type of error is also known as an exception.

When an exception occurs, the program ends, or “crashes,” unless the exceptions are handled by the program. However, you can often prevent exceptions from occurring in the first place by fixing the code that caused the exception.

**Summary**

* Python is a powerful programming language with a simple syntax that’s easy to read and understand. That’s why it’s such a good first language for beginners.
* Python can be used to develop desktop applications such as console applications and GUI applications. Python can also be used to develop web applications. And Python can be used as a scripting language for purposes such as system administration.
* Before the source code for a Python program can be run, it is compiled by the Python interpreter into bytecode that can be run by the Python virtual machine for a computer. The bytecode and virtual machine are what makes Python platform-independent.
* When a program is being run, the operating system, the program, and the program’s data are all stored in the main memory of the computer.
* Disk storage provides persistent data storage for the data that’s operated upon by a program. Disk storage also stores all of the systems software and applications software of the system.
* An integrated development environment (IDE) like IDLE makes it easier to develop Python programs.
* You can use IDLE’s interactive shell to enter and test Python code such as expressions, functions, and statements. You use its editor to enter and edit Python source code.
* To run a program with IDLE, you can press the F5 key. If the program uses the console, the IDLE shell is used as the console.
* The goal of testing is to find all of the errors in a program. The goal of debugging is to fix all the errors.
* If a program has syntax errors, you need to correct them before the program can be compiled. If a program has runtime errors, you need to debug each one until the program runs correctly.
* A runtime error is also known as an exception. To make sure that a program doesn’t crash, you must learn how to handle the exceptions that occur.

**How to write Python code:**

* You’ll want to divide a long statement over two or more lines. Then, you can use implicit continuation:
* You divide a statement before or after an operator like a plus or minus sign.
* You can also divide a statement after an opening parenthesis. When you divide the statement, it’s a good practice to indent its continuation lines.
* The other way to continue a statement is to use explicit continuation. Then, you code a backslash to show that a line is continued. In general, this is discouraged and isn’t usually required, so you shouldn’t need to use it often.
* The first line in a Python program is often a shebang line:

#!/usr/bin/env python

**Coding rules**

* Python relies on proper indentation. Incorrect indentation causes an error.
* The standard indentation is four spaces whenever it is required.
* With implicit continuation, you can divide statements after parentheses, brackets, and braces, and before or after operators like plus or minus signs.
* With explicit continuation, you can use the \ character to divide statements anywhere on a line.

**Description**

• A statement performs a task. Each statement must be indented properly.

• A program typically starts with a shebang line that begins with a hash (#) symbol followed by a bang (!) symbol. This line identifies the interpreter to use when running the program.

• If you’re using IDLE to run your programs, you don’t need a shebang line. However, it’s generally considered a good practice to include one.

**How to code comments**

* To code a block comment, you code a pound sign (#) at the start of each line.
* To code an inline comment, you code the pound sign on the same line but after a Python statement.
* Since all comments are ignored by the Python interpreter, they have no effect on the operation of the program.
* Most of the time, comments are used to document portions of code that are difficult to understand. This can be helpful to the programmer who develops the program as well as to programmers who maintain the program later.
* Comments can also be used to disable statements that you don’t want to run when you compile and run a program. This is called commenting out statements. To do that, you code a pound sign before each statement that you want to comment out. Then, when you’re ready to test those statements, you can uncomment them by removing the pound signs.
* When you’re using IDLE, it’s easy to comment out or uncomment statements. To do that, you can select the statements. Then, you can use the commands available from the Format menu to comment out or uncomment the selected statements.

**Guidelines for using comments**

* Use comments to describe portions of code that are hard to understand, but don’t overdo them.
* Use comments to comment out (or disable) statements that you don’t want to test.
* If you change the code that’s described by comments, change the comments too.

**Description**

* Comments start with the # sign, and they are ignored by the compiler. Block comments are coded on their own lines. Inline comments are coded after statements to describe what they do.
* Comments are used to document what a program or portion of code does. This can be helpful not only to the programmer who creates the program, but also to those who maintain the program later on.
* Comments can also be used to comment out statements so they aren’t executed when the program is tested. Later, the statements can be uncommented so the statements will be executed when the program is tested. This can be helpful when debugging.
* When you’re using IDLE, you can use the Format menu to comment out and to uncomment the statements that you’ve selected.

**How to use functions:**

A function is a group of statements that perform a specific task. Python provides many built-in functions that you’ll use throughout this book. You will learn how to create functions later in the course.

To call a function, then, you code the function name, a set of parentheses, and any arguments that are required by the function within the parentheses. If a function requires more than one argument, you separate the arguments with commas.

The print() function shows how this works. This function displays the data that is passed to it as arguments. But note that the arguments are optional. If no argument is passed to this function, it prints a blank line.

Lets try:

print(“Hello out there!”)

print()

print(“Goodbye!”)

**Description**

* A function is a reusable unit of code that performs a specific task.
* Python provides many built-in functions that do common tasks like getting input data from the user and printing output data to the console.
* When you call a function, you code the name of the function followed by a pair of parentheses. Within the parentheses, you code any arguments that the function requires, and you separate multiple arguments with commas.
* In a syntax summary like the one at the top of this page, brackets [ ] mark the portions of code that are optional. And the italicized portions of the summary are the ones that you have to supply.

**Python data types:**

1. *str* (or string) holds characters like “Mike” or “40”
2. *int* (or integer) holds whole numbers like 21 or -25
3. *float* (or floating-point) holds numbers with decimal places like 41.3 or -25.78

When you work with these data types in a program, you normally assign them to variables. To do that, you code assignment statements like those in the first group of examples in this figure. These statements consist of a variable name like first\_name or quantity1, an equals sign, and the value that should be assigned to the variable. These variables and values are stored in the main memory of the computer.

The names of variables are case-sensitive.

**Code that initializes variables and assigns data to them**

first\_name = "Mike" # sets first\_name to a str of "Mike"

quantity1 = 3 # sets quantity1 to an int value of 3

quantity2 = 5 # sets quantity2 to an int value of 5

list\_price = 19.99 # sets list\_price to a float value of 19.99

**Code that assigns new data to the variables above**

first\_name = "Joel" # sets first\_name to a str of "Joel"

quantity1 = 10 # sets quantity1 to an int of 10

quantity1 = quantity2 # sets quantity1 to an int of 5

quantity1 = "15" # sets quantity1 to a str of "15", not an int of 5

**Code that causes an error because of incorrect case**

quantity1 = Quantity2 # NameError: 'Quantity2' is not defined

.

**How to code literal values**

* To code a literal value for a string, enclose the characters of the string in single or double quotation marks. This is called a string literal.
* To code a literal value for a number, code the number without quotation marks. This is called a numeric literal.

**Description**

* A variable can change, or vary, as code executes.
* A data type defines the type of data for a value.
* An assignment statement uses the equals sign (=) to assign a value to a variable. The value can be a literal value, another variable, or an expression like the arithmetic expressions in the next figure.
* You can assign a value of any data type to a variable, even if that variable has previously been assigned to a value of a different data type.
* Because variable names are case-sensitive, you must be sure to use the correct case when coding the names of variables.

**How to name variables:**

* Use camel case convention: the first letter of each word is uppercase except for the first word.
* To create an identifier that has more than one word in it, most Python programmers use underscores to separate the words in a variable name.
* give your variables meaningful
* names. That means that it should be easy to tell what a variable name refers to and easy to remember how to spell the name.

**Rules for naming variables**

* A variable name must begin with a letter or underscore.
* A variable name can’t contain spaces, punctuation, or special characters other than the underscore.
* A variable name can’t begin with a number, but can use numbers later in the name.
* A variable name can’t be the same as a keyword that’s reserved by Python.

**Python keywords:**

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

**Two naming styles for variables:**

variable\_name # underscore notation

variableName # camel case

**How to code arithmetic expressions:**

The first table in this figure summarizes the arithmetic operators. Here, the

+, -, and / operators are the same as those used in basic arithmetic, and the \* operation is used for multiplication.

Python also has an operator for integer division (//) that truncates the decimal portion of the division. It has a modulo operator (%), or remainder operator, that returns the remainder of a division. And it has an exponentiation operator (\*\*) that raises a number to the specified power.

When an expression includes two or more operators, the order of precedence determines which operators are applied first.

If you need to override the default order of precedence, you can use

parentheses. Then, Python performs the expressions in the innermost sets of parentheses first, followed by the expressions in the next sets of parentheses, and so on. This works the same as it does in algebra, and this is typical of all programming languages.

|  |  |  |
| --- | --- | --- |
| Operator | Name | Description |
| + | Addition | Add two operands |
| - | Subtraction | Subtracts the right operand from the left operand |
| \* | Multiplication | Multiplies two operands |
| / | Division | Divides the right operand into the left operand. The result is always a floating-point number. |
| // | Integer Division | Divides the right operand into the left operand and drops the decimal portion of the result. |
| % | Modulo / Remainder | Divides the right operand into the left operand and returns the remainder. The result is always an integer. |
| \*\* | Exponentiation | Raises the left operand to the power of the right operand. |

**How to use arithmetic expressions in assignment statements:**

Python has three of the compound assignment operators. These operators provide a shorthand way to code common assignment statements. For instance, the += operator modifies the value of the variable on the left of the operator by adding the value of the expression on the right to the value of the variable on the left. When you use this operator, the variable on the left must already exist and have a value assigned to it.

***Code that calculates sales***

tax subtotal = 200.00

tax\_percent = .05

tax\_amount = subtotal \* tax\_percent # 10.0

grand\_total = subtotal + tax\_amount # 210.0

***Code that calculates the perimeter of a rectangle***

width = 4.25

length = 8.5

perimeter = (2 \* width) + (2 \* length) # (8.5 + 17) = 25.5

***The Python code for a Test Scores program***

#!/usr/bin/env python3

counter = 0

score\_total = 0

test\_score = 0

while test\_score != 999:

test\_score = int(input("Enter test score: "))

if test\_score >= 0 and test\_score <= 100:

score\_total += test\_score

counter += 1

average\_score = round(score\_total / counter)

print("Total Score: " + str(score\_total))

print("Average Score: " + str(average\_score))

**An indentation error**

print("Total Score: " + str(score\_total))

print("Average Score: " + str(average\_score))

**Two ways to continue one statement over two or more lines**

**Implicit continuation**

print("Total Score: " + str(score\_total)

+ "\nAverage Score: " + str(average\_score))

**Explicit continuation**

print("Total Score: " + str(score\_total) \

+ "\nAverage Score: " + str(average\_score))

**Coding rules**

* Python relies on proper indentation. Incorrect indentation causes an error.
* The standard indentation is four spaces whenever it is required.
* With implicit continuation, you can divide statements after parentheses, brackets, and braces, and before or after operators like plus or minus signs.
* With explicit continuation, you can use the \ character to divide statements anywhere on a line.

**The test score program with comments:**

#!/usr/bin/env python3

# This is a tutorial program that illustrates the use of the while

# and if statements

# initialize variables

counter = 0

score\_total = 0

test\_score = 0

# get scores

while test\_score != 999:

test\_score = int(input("Enter test score: "))

if test\_score >= 0 and test\_score <= 100:

score\_total += test\_score

counter += 1

# calculate average score

#average\_score = score\_total / counter

#average\_score = round(average\_score)

average\_score = round(score\_total / counter)

# display the result

print("======================")

print("Total Score: " + str(score\_total) # implicit continuation

+ "\nAverage Score: " + str(average\_score))

**Guidelines for using comments**

• Use comments to describe portions of code that are hard to understand, but don’t overdo them.

• Use comments to comment out (or disable) statements that you don’t want to test.

• If you change the code that’s described by comments, change the comments too.

**Description**

• Comments start with the # sign, and they are ignored by the compiler. Block comments are coded on their own lines. Inline comments are coded after statements to describe what they do.

• Comments are used to document what a program or portion of code does. This can be helpful not only to the programmer who creates the program, but also to those who maintain the program later on.

• Comments can also be used to comment out statements so they aren’t executed when the program is tested. Later, the statements can be uncommented so the statements will be executed when the program is tested. This can be helpful when debugging.

• When you’re using IDLE, you can use the Format menu to comment out and to uncomment the statements that you’ve selected.

**How to use functions:**

A function is a group of statements that perform a specific task.

To call a function, then, you code the function name, a set of parentheses,

and any arguments that are required by the function within the parentheses. If a function requires more than one argument, you separate the arguments with commas.

brackets [ ] mark the portions of code that are optional.

A script with three statements

print("Hello out there!")

print()

print("Goodbye!")

The str() function for converting numbers to strings.

***How to assign strings to variables***

first\_name = "Bob" # first\_name = Bob

last\_name = 'Smith' # last\_name = Smith

name = "" # name = empty string

name = "Bob Smith" # name = Bob Smith

***How to join three strings with the + operator***

name = last\_name + ", " + first\_name # name is "Smith, Bob"

***How to join a string and a number with the str() function***

name = "Bob Smith"

age = 40

message = name + " is " + str(age) + " years old."

***What happens if you don’t use the str() function***

message = name + " is " + age + " years old."

Traceback (most recent call last):

File "<pyshell#33>", line 1, in <module>

message = name + " is " + age + " years old."

TypeError: Can't convert 'int' object to str implicitly

***Implicit continuation of a string over several coding lines***

print("Total Score: "

+ str(score\_total)

+ "\nAverage Score: "

+ str(average\_score))

**How to include special characters in strings**

***The new line character***

print("Title: Python Programming\nQuantity: 5")

***The tab and new line characters***

print("Title:\t\tPython Programming\nQuantity:\t5")

***The backslash in a Windows path***

print("C:\\murach\\python")

***Four ways to include quotation marks in a string***

"Type \"x\" to exit" # String is: Type "x" to exit.

'Type \'x\' to exit' # String is: Type 'x' to exit.

"Type 'x' to exit" # String is: Type 'x' to exit.

'Type "x" to exit' # String is: Type "x" to exit.

**How to use the input() function**

The input() function can take one string argument that prompts the user to enter data.

Code that gets string input from the user

first\_name = input("Enter your first name: ")

print("Hello, " + first\_name + "!")

Another way to get input from the user

print("What is your first name?")

first\_name = input()

print("Hello, " + first\_name + "!")

**How to use the int(), float(), and round() functions**

The int() and float() functions convert the data argument, which is typically a str value, to int or float values.

the round() function rounds a numeric value to the specified number of digits.

**How to chain function**

you code one function as the argument of another function. In the chained part of the second example, the input() function is coded as the argument of the int() function. As a result, two functions are chained together in a single statement.

***Code that causes an exception***

x = 15

y = "5"

z = x + y # TypeError: can't add an int to a str

***How using the int() function fixes the exception***

x = 15

y = "5"

z = x + int(y) # z is 20

***Code that gets an int value from the user***

quantity = input("Enter the quantity: ") # quantity is str type

quantity = int(quantity) # quantity is int type

***How to use chaining to get the int value in one statement***

quantity = int(input("Enter the quantity: "))

***Code that gets a float value from the user***

price = input("Enter the price: ") # price is str type

price = float(price) # price is float type

***How to use chaining to get the float value in one statement***

price = float(input("Enter the price: "))

***Code that uses the round() function***

miles\_driven = 150

gallons\_used = 5.875

mpg = miles\_driven / gallons\_used mpg = round(mpg, 2)

***How to combine the last two statements***

mpg = round(miles\_driven / gallons\_used, 2)

**The Miles Per Gallon Program**

#!/usr/bin/env python3

# display a title

print("The Miles Per Gallon program")

print()

# get input from the user

miles\_driven= float(input("Enter miles driven:\t\t"))

gallons\_used = float(input("Enter gallons of gas used:\t"))

# calculate and round miles per gallon

mpg = miles\_driven / gallons\_used

mpg = round(mpg, 2)

# display the result

print()

print("Miles Per Gallon:\t\t" + str(mpg))

print()

print("Bye")

**The test score program**

#!/usr/bin/env python3

# display a title

print("The Test Scores program")

print()

print("Enter 3 test scores")

print("======================")

# get scores from the user and accumulate the total

total\_score = 0

total\_score += int(input("Enter test score: "))

total\_score += int(input("Enter test score: "))

total\_score += int(input("Enter test score: "))

# calculate average score

average\_score = round(total\_score / 3)

# format and display the result

print("======================")

print("Total Score: ", total\_score,

"\nAverage Score:", average\_score)

print()

print("Bye")

**Three types of errors that can occur**

* ***Syntax errors*** prevent your program from compiling and running. Since syntax errors occur when Python attempts to compile a program, they’re known as compile-time errors. This type of error is the easiest to find and fix. When you use IDLE, it highlights the location of the first syntax error that it finds each time you attempt to run the program. Then, you can correct that error and try again.
* Unfortunately, some errors can’t be detected until you run a program. These errors are known as ***runtime errors***, and they throw exceptions that stop the execution of a program.
* Even if a program runs without throwing exceptions, it may contain ***logic errors*** that prevent the program from working correctly. This type of error is often the most difficult to find and fix.

***The goal of testing:***

• To find all errors before the program is put into production.

***The goal of debugging:***

• To fix all errors before the program is put into production.

***The three types of errors that can occur:***

• Syntax errors violate the rules for how Python statements must be written. These errors, also called compile-time errors, are caught by IDLE and the Python compiler before you run the program.

• Runtime errors don’t violate the syntax rules, but they throw exceptions that stop the execution of the program. Many of these exceptions are due to programming errors that need to be fixed. But some exceptions need to be handled by the program so the program won’t crash.

• Logic errors are statements that don’t cause syntax or runtime errors, but produce the wrong results. In the console for the Future Value program above, the future value isn’t correct, which is a logic error.

***Common syntax errors***

• Misspelling keywords.

• Forgetting the colon at the end of the opening line of a function definition, if clause, else clause, while statement, for statement, try clause, or except clause.

• Forgetting an opening or closing quotation mark or parenthesis.

• Using parentheses when you should be using brackets, or vice versa.

• Improper indentation.

***Problems with names and values***

• Misspelling or incorrectly capitalizing a variable or function name.

• Using a keyword as a variable or function name.

• Not checking that a value is the right data type before processing it. For example, using a number when it needs to be converted to a string, or vice versa

**How to use the IDLE shell to test functions**

One of the benefits of using IDLE is that you can use the IDLE shell to test any of the functions in your programs. This lets you test a function in variety of ways without running the whole program. Often, that means you can test a function more thoroughly in less time.

This is illustrated by the first two examples in this figure. In the first example, the Future Value program has been run and thus loaded into the shell. Then, statements are entered into the shell that call the functions in the program. First, the get\_float() function is tested with the three arguments it requires. Then, the calculate\_future\_value() function is executed.

In the second example, the Convert Temperatures program of the last chapter has been run and loaded into the shell. Then, the convert\_temp() function of the program is called twice. For each call, this function gets two user entries, and then displays the result.

The third example in this figure shows that you can also use the shell to test the functions of a module that isn’t a program. To do that, you first import the module. In this example, the temperature module that’s used by the Convert Temperatures program is imported into the shell. Then, its functions are called.

When you test functions this way, you isolate them from the rest of the program or module. That makes it easier to see whether they work the way they’re supposed to. As a result, this is an efficient way to test each function of a program with a variety of arguments.

**How to use the IDLE debugger**

The first step in using the IDLE debugger is to set a breakpoint at the point in the program that precedes the statements that you think are causing the bug. To set a breakpoint, you right-click on a statement in the IDLE editor and select Set Breakpoint from the context menu.