chapter W19Python

**Objectives**

\* To give an overview of the Python programming language

\* To cover the basic syntax of Python

\* To show how to install and run Python on a Linux system

\* To provide several basic practical examples of using Python in the Linux environment

\* To cover the commands and primitives:

python

W19.1 Introduction

In the Linux environment, if you are presented with a task that requires you to do either script writing or programming, the first thing you have to decide is what programming model you are going to adopt to accomplish the task. In simple terms, this means using any of the three predominant programming models: the procedural/imperative programming model, the object oriented programming (OOP) model, or the logic programming model. Of course, how well you accomplish the task depends on how much script writing or programming experience you have. And, perhaps, if you are doing this task with a group of people, their experiences and preferences count for a great deal as well. But, how you formulate the task in terms of any of the models is related, most importantly, to your experience in using these models. There are no simple guidelines for applying any of the models to the vast number of possible script writing or programming tasks that exist in Linux.

However, the bottom line in the accomplishment of your task is how familiar you are with the syntax of the languages that implement the model you choose.

That is why we explain Python syntax in detail. Python can utilize all of these programming models, and can, in fact, mix the techniques used in the models. We make some commentary on this here, and illustrate some simple uses of the models in the next few sections.

Python is a *high-level*, *structured* (in this case, this means built of regular components), and *interpreted* programming (or scripting) language, as opposed to a low-level, compiled language like C or Java. As stated, it is also a *multi-paradigm* programming language, which allows you to use data abstractions from the three predominant paradigms. For our purposes, “scripting” and “programming” can be thought of as the same thing.

Why Use the “Pure” OOP Approach?

Instead of totally defining the most fundamental aspects and parts of OOP (which you can find easily in the extensive literature), we choose to contrast and compare it to the more traditional procedural programming method, to give you a more intuitive feel for the approach OOP uses.

Procedural, non-OOP programs, as we have noted, can run through their structured, flow-of-control operation using the Python syntax we show in the sections below. Non-trivial programs can possibly be made up of groupings of flow-of-control operations, known as procedures, modules, or subroutines. This way of using Python syntax, since the language “accommodates” the procedural model, can put together a program that emphasizes more traditional algorithms and Abstract Data Types (ADT). Simple examples of these two things are the algorithm (or mathematical plan) for computing the greatest common factor of two integers, or the definition of integers themselves as a certain kind of numerical value, along with the prescribed set of possible operations on them. One of the biggest problems with this approach is that if variables take on a global scope, outside of the “container” they are defined in, the assignment of values to these can be conflicting and cause errors. That is, in part, why using namespaces, as we show in Chapters 17 in the printed book, and W26 at the book website, to isolate processes, is so effective. Variables (which are active in a process namespace) are exclusively, and locally, defined and used.

In Python OOP, this is basically done using what is known as “encapsulation”. The idea is that the data and functions that manipulate the data, are one entity. This keeps both safe from unwanted changes to it from any environment outside the entity. This fundamental component of OOP, that combines data with a set of methods for accessing and managing the data, is called an object. Its data treatment model is completely different from the procedural program method previously described.

Of course, it is possible with the rich set of syntax available in Python, to combine both non-OOP and OOP styles. Python does not faithfully implement the functional, or logic, programming structure known as *tail recursion*, as does a language such as MIT scheme. But it is capable of recursion, as we show below.

W19.1.1 Python Program Data Model

Even though Python incorporates many of the features of a multi-paradigm programming ensemble, its realistic, and perhaps more so, its fundamental basis is the following:

Everything in the Python program can be referred to as an *object*, with the same exact meaning of the word *object* used in OOP. These objects have three parts: an *identifier*, a *type*, and a *value*.

For example, when you assign X = 62.25 interactively in the Python interpreter, or in a Python module, script file, or library of modules, a real number object type is created; it has a value of 62.25, and it is identified as an object with a pointer to its location in memory. X is the identifier that refers to its specific location in memory.

In OOP languages such as Python and Java, the type that an object assumes gives it membership in a particular set, called its *class*. The class of the object limits and also defines, in a fashion, what are known as the *methods*, or operations, which can be performed on or with it.

When a particular object of some type is created, that particular object is called an *instance* of that type. In general, an object’s identity and type cannot be changed. They are known as *immutable*. If an object’s value can be modified, the object is said to be *mutable*. An object that refers to other objects to obtain value and type is known as a *container*.

Objects can also define their own *attributes* or characteristics of the data they are comprised of, and even the *methods* used on them. An attribute is a property or value associated with an object. A method is a function internal to a class of objects that performs some sort of operation on those objects when the method is invoked.

Attributes and methods are accessed using the dot (.) operator, as shown in the following examples:

x = 2 + 4j creates a complex number x.

A = a.real uses a method known as real to extract the real part (an attribute) of a.

c = [1, 2, 3] creates an instance of type list identified as c of the integers 1, −2, and −3.

c.append(7) adds a new element to c using the append method.

W19.1.2 Python References and Releases

Before you begin this section, and as you proceed through the rest of this chapter, it would be helpful for you to reference, and read for understand in a “top-down” manner, the following references (the versions and editions of which were available at the time this book became available). The Python online documentation, and a printed book, for the release of Python we are using, are as follows:

*The Python Language Reference -- https://docs.python.org/3/reference/*

https://docs.python.org/3/tutorial/index.html

*Programming Python: Powerful Object-Oriented Programming*, 4th edition by Mark Lutz

Use the latest versions and editions of the above references, and if necessary, have a printed form of them. Whatever top-down principles you can carry with you from these references throughout your Python programming experience, both in this chapter and beyond, are very important, and will enable you to see the complex details in a larger context.

Since 2017, version 2.7.13 was expected to be the last major release of version 2.7.X, and any new features in the language will not be supported. Security and bug fixes for version 2.7 will continue until 2020.

This is a very good reason to learn, and apply Python 3, to everything we present in this chapter.

By default, we use the Python 3 syntax.

We also provide an abbreviated Python command syntax reference at the end of this chapter, in Table W19.9. You can refer to this whenever you need a handy reference for syntactic components of the language.

W19.1.3 Ultimate Reference Glossary

A simplified and abbreviated glossary of some of the terms we have abstracted from the references in Section W19.1.2 are as follows:

**Class**: A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**Expression**:A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators, or function calls, which all return a value.

In contrast to many other languages, not all language constructs are expressions. There are also statements which cannot be used as expressions, such as print or if. Assignments are also statements, not expressions.

**Immutable**: An object with a fixed value. Immutable objects include numbers, strings, and tuples. Such objects cannot be altered. A new object has to be created if a different value has to be stored; for example, a key in a dictionary.

**Iterable**: An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as a list, string, and tuple) and some nonsequence types like dict and file and objects of any classes you define with an \_\_iter\_\_() or \_\_getitem\_\_() method. Iterables can be used in a for loop and in many other places where a sequence is needed. When an iterable object is passed as an argument to the built-in function iter(), it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call iter() or deal with iterator objects yourself. The for statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop.

**Lambda**: An anonymous inline function consisting of a single expression which is evaluated when the function is called. The syntax to create a lambda function is lambda [arguments]: expression.

**Method**:A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first argument (which is usually called self).

**mutable**:Mutable objects can change their value but keep their class identity.

**Pythonic**:An idea or piece of code which closely follows the most common usages of the Python language, rather than implementing code using structures common to other languages. For example, a common usage in Python is to loop over all elements of an iterable using a for statement. Many other languages do not have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

for i in range(len(money)):

print (money[i])

The Pythonic way:

for bills in money:

print (bills)

**Sequence**:An iterable which supports efficient element access using integer indices via the \_\_getitem\_\_() special method and defines a len() method that returns the length of the sequence. Some built-in sequence types are list, str, tuple, and unicode. Note that dict also supports \_\_getitem\_\_() and \_\_len\_\_(), but is considered a mapping rather than a sequence because the lookups use arbitrary immutable keys rather than integers.

**type**:The kind of object, such as integers or character strings.

W19.1.4 Python Standard Type Hierarchy

The *type* of an object describes the Python data structure representation of the object as well as the methods and operations that can be carried out on that object. Table W19.1 is a listing of the type categories, and following it is a brief description of some of the categories in the table.

|  |  |  |
| --- | --- | --- |
| **Category** | **Name** | **Description** |
| None | None | Null object |
| Numbers | int | Plain integer |
|  | Long | Arbitrary-precision integer |
|  | Float | Floating point number |
|  | Complex | Complex number |
|  | Bool | Boolean (True or False) |
| Sequences (immutable) | str | Character string |
|  | Unicode | Unicode character string |
|  | tuple | Tuple |
| Sequences (mutable) | list | List |
|  | bytearray | Returned by bytearray() |
| Mapping | dict | Dictionary |
| Sets | set | Mutable set |
|  | Frozenset | Immutable set |
| Callable | BuiltinFunctionType | Built-in functions |
|  | BuiltinMethodType | Built-in methods |
|  | type | Type of built-in types and classes |
|  | object | Ancestor of all types and classes |
|  | FunctionType | User-defined function |
|  | InstanceType | Class object instance |
|  | MethodType | Bound class method |
|  | UnboundMethodType | Unbound class method |
| " Modules | ModuleType | Module |
| " Classes | object | Ancestor of all types and classes |
| " Types | type | Type of built-in types and classes |
| " Files | file | File |
| " Internal | CodeType | Byte-compiled code |
|  | FrameType | Execution frame |
|  | GeneratorType | Generator object |
|  | TracebackType | Stacks traceback of an exception |
|  | Slice | Generated by extended slices |
|  | Ellipsis | Used in extended slices |
| " Classic | Classes ClassType | Legacy class definition |
|  | InstanceType | Legacy class instance |

Table W19.1 Python Type Categories

The None type has a single value that contains a null object (an object with no value). Its truth value is False.

**Numeric types**: Booleans, integers, long integers, floating point numbers, and complex numbers.

**Sequence types**: *Sequences* represent ordered sets of objects indexed by nonnegative integers and include strings, Unicode strings, lists, and tuples.

**Mapping types**: A *mapping object* represents an arbitrary collection of objects that are indexed by another collection of nearly arbitrary key values. Unlike a sequence, a mapping object is unordered and can be indexed by numbers, strings, and other objects. *Dictionaries* are the only built-in mapping type and are similar to hash.

**Set types**: A *set* is an unordered collection of unique items. Unlike sequences, sets provide no indexing or slicing operations. They are also unlike dictionaries in that there are no key values associated with the objects. In addition, the items placed into a set must be immutable.

**Callable types**: These represent objects that support the function call operation. There are several flavors of objects with this property, including user-defined functions, built-in functions, instance methods, and classes.

**Classes and types**: When you define a class, the class definition normally produces an object of type type.

**Modules**:The module type is a container that holds objects loaded with the import statement.

**Files**:The *file* object represents an open file and is returned by the built-in open() function.

**Internal types**: Objects used by the interpreter are exposed to the user, such as *traceback objects*, *code objects*, *frame objects*, *generator objects*, *slice objects*, and the *ellipsis object*.

**Code objects**: These represent raw byte-compiled executable code, or *bytecode*, and are typically

returned by the built-in compile() function.

**Frame objects**: These are used to represent execution frames and most frequently occur in

traceback objects.

**Traceback objects**: These are created when an exception occurs and contains *stack trace information*.

**Generator objects**: These are created when a generator function is invoked . A generator function is defined whenever a function makes use of the special yield keyword.

**Slice objects**: These are used to represent slices given in extended slice syntax, such as a[i:j:stride], a[i:j, n:m], or a[..., i:j].

**Ellipsis object**: The ellipsis object is used to indicate the presence of an ellipsis (...) in a slice. There is a single object of this type, accessed through the built-in name Ellipsis. It has no attributes and evaluates as True.

**Classic classes**: In versions of Python prior to version 2.2, classes and objects were implemented using an entirely different mechanism that is now deprecated. For backward compatibility, however, these classes, called *classic classes* or *old-style classes*, are still supported.

W19.1.5 Basic Assumptions We Make

Four basic and important assumptions we make in this chapter are:

1.You have Python 3.X installed on your Linux system. This installation was either done by the system administrator at the time the Linux system was installed on the computer you are using, or by you. On our base Linux systems, Debian-family Debian 9.1, Ubuntu 16.04, and Linux Mint 18.2, and CentOS 7.4 (the ones we have used to illustrate *everything* in this book), Python 3.X was already installed as part of the installation of the system itself.

If you type in the following command (as we did on our Linux Mint 18.2 system), you will see where Python and its components (and what versions) are installed on your system:

$ **whereis python**

python: /usr/bin/python3.5m /usr/bin/python /usr/bin/python3.5 /usr/bin/python2.7 /usr/lib/python2.6 /usr/lib/python3.5 /usr/lib/python2.7 /etc/python /etc/python3.5 /etc/python2.7 /usr/local/lib/python3.5 /usr/local/lib/python2.7 /usr/include/python3.5m /usr/include/python2.7 /usr/share/python /usr/share/man/man1/python.1.gz

From the above output, both Python 2.7 and 3.5, are installed on our Linux Mint 18.2 system.

2. The path of execution to the Python program and the path of execution to all the Python scripts you create in this chapter include the current working directory that you want to do Python in! If you do not know, given the particular shell you are using (we use the Bash shell, with the $ prompt), what your path of execution is set to, go back to Chapter 2, Section 2.8 and examine your path and set it properly. For example, in the Bash shell, you can see your path of execution by typing **echo $PATH** at the shell prompt. On our representative Linux systems (Debian-family and CentOS 7), Python 2.7.X and Python 3.5.X are installed by default in /usr/bin, as seen for example in Linux Mint 18.2 from the **whereis python** command output of 1. above.

3. You are doing Python in a console or terminal window *without* an integrated development environment (IDE). Therefore, the basic procedures are as follows:

a. You edit Python scripts in your favorite text editor, save them to the current working directory, and execute Python in that current working directory.

b. You execute Python interactively by typing commands into its “interpreter” shell window.

c. On our representative Linux systems (Debian-family and CentOS 7), when you type **python** on the command line, you launch Python version 2.7.X., and when you type **python3** on the command line, you launch Python version 3.5.X. See Section w19.1.6 for various further details of this procedure.

When you become more familiar with Python, you may wish to make your work with the language more efficient by using an IDE.

4. In general, whenever we want you to type something on the Python command line, we will indicate what is to be typed in **bold** text. In addition, output from Python will be shown in unbolded text.

Computer programs execute and accomplish their objectives in a particular order, from start to finish. They may “branch” within that order, perhaps to only execute some of their instructions, based on certain logical tests or conditions. They may also repeat segments of their operation, either for some predetermined number of times, or indeterminately, based on changing conditions. Python conforms to this model, and operates using the following scheme of levels:

1. *Everything* in Python programs, or scripts, is composed of modular components.

2. These modular components contain syntactically correct Python statements.

3. These statements contain expressions.

4. The expressions create and manipulate objects.

W19.1.6 Running Python Using the Three Ways

The following subsection illustrates the three ways that we use to run Python on our representative systems.

*W19.1.6.1 Way 1 (Interactive Mode)*

As shown in Section W19.1.5, in a console or terminal window, you type **python3**. On our representative Debian-family Linux Mint 18.2 system, this launches Python 3.5.2. Of course, the exact release number of Python launched by this command is system-dependent, according to what flavor of Linux you’re using.

For example, on a Linux Mint 18.2 system, if you were to type in **python3**, you would launch Python version 3.5.2.

The program executes, and you are presented with the Python command prompt, **>>>**. Then, you type single or multiple lines of Python code on the Python command line, and see the results immediately. A good reason to use this mode is that you can test small fragments of Python code, one line at a time, directly in the Python interpreter. A simple example of this would be as follows:

**Example W19.1**

**>>>print ("How about some more?")**

How about some more?

**>>>**

To submit a line of Python code to its interpreter, at the end of the line, press <Enter> on the keyboard.

*W19.1.6.2 Way 2 (Script Mode)*

You use a text editor of your choice to create and save possibly multiple, properly formatted Python commands in a file, called a *script file* (perhaps named **first.py).** That script file is in the current working directory in which you will be executing Python. You must also be sure that you have execute privilege on **first.py**, levied via the **chmod** command. Then, you run Python with the **python3** command, and **first.py** as the command argument.

A good reason to use this mode is if you have scripts with more than a few lines of code in them, and you do not want to type that code in every time you want to run it. A simple example of this would be as follows:

**Example** **W19.2a**

$ **python3 first.py**

where **first.py** is a file full of syntactically correct Python commands in the current working directory, and the preferred, selected, and available Python program is in the path of execution of programs in your Linux working environment.

This method of executing the Python code is sometimes called running it as a user-written *library* module.

If the commands in **first.py** do not contain any output directed to the screen, such as using print statements, the shell prompt will immediately reappear, and you will not be in the Python interpreter when the script file terminates without error!

There is an alternative way of using this method, that depends upon the working environment within which you are executing Python. That alternative, very similar to the way of executing a Bash script file as we have shown in Chapter 12, is to include this line as the first line in the Python script file (which we have named **first.py** in this example case)-

**Example** **W19.2b**

**#! /usr/bin/env python3**

Again, you must also be sure that you have execute privilege on **first.py**, levied via the **chmod** command. Then, to execute the **first.py** script file, on the Bash command line, type the following:

**$./first.py**

This method, on our Linux Mint 18.2 system, uses Python Version 3 to execute the script file.

The main advantage of the method of Example W19.2b is that, depending on which version of Python you want to run, you can place the command name for that version in the first line of the script file. For example, if you want to use Python Version 2.7.X to run the script file, you could modify the first line in the script file to read-

**#! /usr/bin/env python**

There are portability issues with this method, for example when the working environment is in conflict with what version of Python you want to execute the script file code with. But for beginners, you can ignore those issues for now.

*W19.1.6.3 Way 3 (Import Script Mode)*

Similarly to Way 2, you use a text editor of your choice to create and save multiple Python commands in a script file, perhaps named **first.py**, in the current working directory in which you are executing Python. Then, you run Python, and at the Python command prompt you bring the script file into Python with the **import** command. A good reason to use this mode is if your script files contain function definitions. A simple example of this would be as follows:

**Example** **W19.3**

**>>>import first**

>>>

where **first** is the file without the **.py** extension. It should contain Python commands and be in the current working directory. Now, the objects, statements, expressions, and modules (like Python functions) in **first.py** are available to you in the Python interpreter. A good reason to use this mode is to bring those structures and functions into the current interactive Python session environment, or namespace.

*Caution*: Once you leave Python by holding down the <Ctrl> and D keys on the keyboard, the current interactive session is ended, and the environment you have created in Python is lost.

W19.1.7 Uses of Python

Python can accomplish several kinds of programming tasks, which might be broken down into the following sample categories:

Shell scripting

Systems programming

Network and Internet scripting

Database programming

Systems administration scripting

Graphical user interface (GUI) scripting

Scientific and math programming

Data mining

In this chapter, we use all three modes of running Python shown, and we give a beginners’ introduction to the Python language. We also follow the model of level schemes shown, going roughly from the bottom of the scheme to the top. The chapter can be divided into three parts:

Section W19.2 gives you information on the installation of Python.

Section W19.3 gives the basic syntax of Python within that model scheme.

Section W19.4 gives a few simple, worked examples of some of the practical programming task categories for which you can use Python in Linux.

W19.2 Information on the Installation of Python

The reference Linux systems (Debian-family Debian 9.1, Ubuntu 16.04, Linux Mint 18.2, and CentOS 7.4) we use in this book come with some versions of Python 3.X already installed and usable by a normal user. But, you may need to install Python on your system if you are doing the install of some other Linux operating system and it does not include Python, or if you want to install a later version of the software alongside or to replace the version your system already has on it. Be aware that because of variations in the way your system has been installed by you or the system administrator, and exactly what “flavor” and version of Linux has been installed, the installation procedure may have to be done by a system administrator for you! A good example of a similar situation would be that you do not have a C compiler available, or you want to upgrade to the latest *gcc* compiler, and do not know how to do that with, or without, a package manager!

We give installation instructions for the gcc compiler in Appendix A of the printed book. We do not give installation instructions for the installation of any version of Python in Appendix A in the printed book. But we do give instructions for use of package managers on our base Linux systems in Appendix A of the printed book.

W19.2.1 Finding Out if Python 2.7.X is Installed on Your Linux System

The easiest way of knowing whether or not Python 3 is already installed on the computer you are using, is to simply type **python** and press <Enter> at the Bash shell prompt ($) in a console or terminal window. When we do this on our Linux Mint 18.2 system we obtain the following output:

~$ **python**

Python 2.7.12 (default, Dec 4 2017, 14:50:18)

[GCC 5.4.0 20160609] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

Also, when we typed **python3** and press <Enter> at the Bash shell prompt ($) in a console or terminal window, we got the following output on out Linux Mint 18.2 system:

$ **python3**

Python 3.5.2 (default, Nov 23 2017, 16:37:01)

[GCC 5.4.0 20160609] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>>

The three greater-than symbols (>>>) are the Python interpreter prompt, letting you know you are in Python!

To exit to the command line prompt in the terminal window, press <Ctrl-D>after the >>>.

The first line of responses from the system shows that Python 2.7.12 or Python 3.5.2 are running on this system.

If you get an error message on the Bash command line in the terminal window, such as command not found, either Python has not been installed on your system, or you do not have access to it. You would need to then contact your system administrator to install the program or set up your working environment to give you privileges to execute it. Or if you are the system administrator, for example on your own single-user Linux computer, you would need to install the appropriate versions of Python, and set up your working environment so that you have access to those versions of the program.

Every program in this chapter can be done, by default, in Python 3, particularly those in Section W19.4.3 through W19.4.5.

If you want to run the examples in Sections W19.4.3 through W19.4.5, you *must* obtain Python 3.5.2 (or the latest release of Python 3 at the time you are reading this), and install that on your system if it is not already installed.

The easiest way of installing Python on your particular Linux system (if it is not already installed!) is to use the package manager for your system, and the methods current at the time you are reading this.

W19.3 Basic Setup and Syntax, and Getting Help

To use Python in practical examples, as shown in later sections of this chapter, it is first necessary to become familiar with the syntax of the language; for example, how it is used as a calculator to execute single lines of Python code to accomplish short, meaningful tasks.

If you need help on a particular module, keyword, or topic in Python, you can always type in the function call to help as follows to get into the help system:

>>> **help ( )**

Welcome to Python 3.5's help utility!

If this is your first time using Python, you should definitely check out

the tutorial on the Internet at http://docs.python.org/3.5/tutorial/.

Enter the name of any module, keyword, or topic to get help on writing

Python programs and using Python modules. To quit this help utility and

return to the interpreter, just type "quit".

To get a list of available modules, keywords, symbols, or topics, type

"modules", "keywords", "symbols", or "topics". Each module also comes

with a one-line summary of what it does; to list the modules whose name

or summary contain a given string such as "spam", type "modules spam".

help> quit

You are now leaving help and returning to the Python interpreter.

If you want to ask for help on a particular object directly from the

interpreter, you can type "help(object)". Executing "help('string')"

has the same effect as typing a particular string at the help> prompt.

>>>

W19.3.1 Printing Text, Comments, Numbers, Grouping Operators, and Expressions

One of the first things you must know about how a calculator works is how to enter numbers and mathematical expressions on the calculator. So, with Python, instead of listing all the syntactic rules, we will do a number of examples to illustrate and have you work with those rules.

Here are a few important considerations you need to be aware of before you enter any Python code on the Python command line or into a file.

1. The rule of four: The *indentation* spaces that you place on each line of Python code are very important! Since Python is a structured programming language that uses specific structures in blocks, those blocks are delimited or defined by the indentation you give them on each line (unlike in other languages that use specific printing characters to delimit blocks). This means you must line up your blocks of Python structures vertically, starting from the left-hand side, and for our purposes, use four spaces for each indented block. For example, the following sample shows this four-space indentation constraint:

Block 1 head

xxxxxxxxx

xxxxxxxxx

Block 2 head

xxxxxxxxx

xxxxxxxxx

Block 3 head

xxxxxxxxx

xxxxxxxxx

end of Block 3

end of Block 2

end of Block 1

where Blocks 1, 2, 3 and so on and their statements xxxxxxxxx line up vertically with an indentation of four spaces for each block from left to right. This is shown in more detail next.

2. Normal order and applicative order evaluation: The order of execution of a mathematical expression used by Python is PEMDAS: parentheses, exponents, multiplication, division, addition, and subtraction. See Table W19.2 for even more details of operator precedence in Python expressions.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| () | Parentheses (grouping) |
| *f*(args...) | Function call |
| *x*[index:index] | Slicing |
| *x*[index] | Subscription |
| *x.attribute* | Attribute reference |
| \*\* | Exponentiation |
| ~*x* | Bitwise not |
| +*x*, -*x* | Positive, negative |
| \*, /, % | Multiplication, division, remainder |
| +, - | Addition, subtraction |
| <<, >> | Bitwise shifts |
| & | Bitwise AND |
| ^ | Bitwise XOR |
| | | Bitwise OR |
| in, not in, is, is not, <, <=,  >,  >=, <>, !=, == | Comparisons, membership, identity |
| not *x* | Boolean NOT |
| and | Boolean AND |
| or | Boolean OR |
| lambda | Lambda expression |

Table W19.2 Python Order of Evaluations

The first thing we want Python to do is to print, or echo, a line of text we type at the keyboard. This is done by typing the following at the Python 3 command prompt:

>>>**print ("This is the number of fingers I am holding up:**")

This is the number of fingers I am holding up:

To add comments to a script file of Python commands, you place the pound sign(#) before everything on the line you want commented. For example, in interactive mode:

>>> **#** **This is a comment, which you can use to annotate your script file code.**

…

>>> **# Anything after the # is ignored by python.**

…

>>> **print "You could have comments appear like this:" # this comment is ignored****.**

You could have comments appear like this:

>>> **# You can also use the pound sign to comment out a piece of code:**

…

>>> **# print "This won't run."**

…

>>> **print "This will run."**

This will run.

>>>

In this interactive session, just press <Enter> on the Python command line when the … appears.

Quotation marks are used for “string literals”, or characters of text you want to include “literally” as they are in the code of a computer program. To include a single quotation mark in a string literal, enclose it inside of double quotation marks. For example:

>>> **'"Don\'t," he said.'**

'"Don\'t," he said.'

>>> **print('"Don\'t," he said.')**

"Don't," he said.

>>> **s = 'First place.\nSecond place.'**# \n means newline

>>> **s** # without print(),\n is included in the output

'First line.\nSecond line.'

>>> **print(s)** # with print(), \n produces a new line

First place.

Second place.

Triple quotation marks are used to enclose long lines of string literals.

Next, we want to combine some text with an arithmetic expression that Python evaluates for us:

>>> **print (“Not Ring Fingers”, 7 − (1 + 1))**

Not Ring Fingers 5

Notice that in evaluating the mathematical expression, Python evaluates what is in parentheses first by doing the addition of 1 + 1, then, from left to right, subtracts 2 from 7.

In-Chapter Exercises

Have Python evaluate the following expressions, and list what Python prints:

1. ((7 + 5) \* (3 + 2))/(6/18)

2. How can you change the previous expression so that it yields a numeric answer, and why?

3. 3 + 2 + 1 – 5 + 4 % 2 – 1 / 4 + 6

4. What kind of operator is the percent sign (%)?

We can also use relational operators in arithmetic expressions, such as less than (<), greater than (>), greater than or equal to (>=), and less than or equal to (<=).

For example:

>>> **print ("Is it true that 3 +** **1** **<** **5** **-** **7?")**

Is it true that 3 + 1 < 5 - 7?

>>> **print (3 +** **1** **<** **5** **- 7)**

False

>>> **print ("Is it greater?", 4 > -2)**

Is it greater? True

>>> **print ("Is it greater or equal?", 4 >= -2)**

Is it greater or equal? True

>>> **print ("Is it less or equal?", 4 <=** **-2)**

Is it less or equal? False

In-Chapter Exercise

5. What are the results of typing in the following Python statements, and why?

>>> **5 + 7>= 6 <= 78 – 9**

>>> **5 + 7>= 6 >= 78 – 9**

>>> **(5 + 7>= 6 >= 78 – 9)/8**

>>> **(5 + 7>= 6 >= 78 – 9)/–8**

W19.3.2 Variables

An important feature of a high-level programming language like Python is providing for names that allow you to refer to computational objects. The name represents, or stands for in any particular computational environment of interest, the value or values which the object can take on; thus it is called a ‘variable’.

Python variable names can contain both letters, numbers, and the underscore (\_) character, but must begin with a letter. If you get an error message about the use of a variable name, it may be a reserved word, or keyword, in Python. Table W19.3 lists the 31 keywords that may not be used as variable names in a Python statement.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| and | del | from | not | while |
| as | elif | global | or | with |
| assert | else | if | pass | yield |
| break | except | import | print |  |
| class | nonlocal | in | raise |  |
| continue | finally | is | return |  |
| def | for | lambda | try |  |

Table W19.3 Python 3.5.2 Keywords

To get a list of keywords in your current version, type the following at the Python prompt. In our version of Python 3.5.2, we got the output shown:

>>> **import keyword**

>>> **print(keyword.kwlist)**

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

>>>

In the following simple example, we define some variables, and use them:

>>> **pie = 3.14159**

>>> **radius = 10**

>>> **pie \* (radius \* radius)**

314.159

>>> **circumference = 2 \* pie \* radius**

>>> **circumference**

62.8318

W19.3.3 Functions

Python provides a programming construct called a *function* that allows you to define your own named procedures and lets you reuse those procedures in a modular fashion in your code. You can think of a function as a black box machine that takes named objects present before the function call or invocation as inputs, processes them inside the black box with names that are only seen inside the black box, and then spits them out as named objects available to the Python program via an assignment statement.

The general form of a function definition is:

def name (formal parameters)**:**

body of the function

return (returned parameters)

where name is the name you give the function (make sure you are not using a Python keyword!), “formal parameters” are the named objects that are passed to the function so it can carry out some operations on them and, optionally, “returned parameters” are named objects that are used by your program.

There are three basic ways you can execute a function:

1. Use Way 1 (Interactive mode), and by typing or copying and pasting each line of code that is the function definition into the interpreter at the Python command line prompt.

2. Use Way 2 (Script mode), and create the function definition in a file with a **.py** extension. Then run Python at the shell prompt with the name of the function definition file plus extension as an argument to the Python command.

3. Use Way 3 (Import script mode), and by importing a file (sometimes called a *module*), which you have created with a text editor and which contains the function definition, into the current session of Python. You can then use the function definitions and named objects in that file.

A simple example of the first way to use a function definition in Python is as follows. Type the following seven lines of Python code at the Python command prompt. The ellipses (…) are another form or the Python prompt, requesting more command line entry. Remember to indent the lines shown, and press <Enter> on the keyboard after each).

>>> **w = 7**

>>> **x = 12**

>>> **def add(a, b):**

… **c = a + b**

… **return (c)**

…

>>> **z = add(w, x)**

>>> **z**

19

>>>

From this example, we can call w and x the actual parameters passed to the function definition, a and b the formal parameters used in the body of the function definition, and c a returned parameter. Notice that, for the returned parameter to be used in the remainder of your Python session after the function definition, you must assign a named variable on the left-hand side of the equals sign to the function call or invocation on the right-hand side of the equals sign.

A simple example of using Way 3 (Import script mode) to bring a function definition into Python is as follows:

Use the text editor of your choice to create the following file, named **math1.py**, in the current working directory you are executing Python in:

**Example W19.4**

def add(a, b):

c = a + b

return(c)

def subtract(a, b):

c = b - a

return(c)

def multiply(a, b):

c = a \* b

return(c)

def divide(a, b):

c = a / b

return(c)

Then, at the Python command line, type the following three lines of Python code:

>>> **import math1**

>>> **z = math1.add(3, 4)**

>>> **z**

7

>>>

The first line you typed in at the Python command line made the named objects in the **math1.py** module available in the current session of Python. The second line allowed you to address the add function from that module as math1.add. The assignment statement on the second line also allowed you to add 3 and 4 together and assign the returned value to a variable named z. We will cover more about Python modules, and global and local scopes in functions, in Section W19.3.

W19.3.4 Conditional Execution

As mentioned in Section W19.1, the order in which computer programs execute includes a “branching”, or conditional execution structure. Python uses the truth value of test conditions to determine whether certain blocks of code will be executed or not. This is implemented in Python with the if statement. The general form of the if statement construct is:

if cond1 is true:

initial statement(s)

elif cond2 is true:

additional statement(s)

else:

final statement(s)

where:

cond1 is a test condition whose truth value determines whether the initial statement(s) block gets executed

cond2 is a test condition whose truth value determines whether the additional statement(s) block gets executed

else is the default which executes final statement(s)

A simple example of the use of this structure using Way 1 (Interactive mode) in Python is as follows (remember to press <Enter> on the keyboard after each line you type in):

**Example W19.5**

>>> **x = 1**

>>> **if x == 0:** # the == is a logical, or Boolean operator

... **print ("x equal 0")**

... **elif x == 1:** #one or more of these optional blocks are allowed

... **print ("x equal 1")**

... **else:** #the optional block

... **print ("x is something else")**

...

x equal 1

>>>

In-Chapter Exercises

6. If you leave out the elif block in Example W19.5, what prints out?

7. If you change the first line to read x = 3, and leave out the else line in Example W19.5, what prints out?

8. If you change the first line to read x = 1.8 in Example W19.5, what prints out?

It is also possible to nest conditional execution blocks inside of one another. For example:

**Example W19.6**

**>>> w =** **36**

**>>> y = 13**

**>>> z = 20**

>>> **if w < 37:**

...**print ("w is less than 37")**

... **if y > 13:**

... **print ("y is greater than 13")**

... **elif y == 13:**

... **print ("y is equal to 13")**

...  **else:**

...  **print ("****y is less than 13")**

... **if z > 21:**

... **print ("z is greater than 21")**

... **elif z == 21:**

... **print ("z is equal to 21")**

...  **else:**

... **print ("z is less than 21")**

... **else:**

... **print ("w is greater than or equal to 37")**

...

w is less than 37

y is equal to 13

z is less than 21

>>>

W19.3.5 Determinate and Indeterminate Repetition Structures and Recursion

Python can repeat segments of program or script file structure in two basic ways: via *determinate repetition structures* or *indeterminate repetition structures*. Traditionally, a determinate repetition structure is called *counting repetition*, and an indeterminate repetition structure is called *logical repetition*. These two methods are implemented with the for procedural statement and the whileprocedural statement. Generally, if you know ahead of time (at the time you are writing the code) how many repetitions of a block of code you want to execute, you use the for statement, and if you do not (when, for example, you allow the user to input the number of repetitions as the script file is run), you use the while statement.

Of course, it is possible to implement the same two ways of repetition by *not* using a specific structured programming approach; for example, by using conditional execution and unstructured switching to obtain the same results. But, we choose to show the structured approach in Python.

Make sure that in the body of statements included in the indeterminate repetition block that while is executing, the test condition for continued execution becomes false. Otherwise, this will result in *infinite repetition*! To halt infinite repetition in an executing script file, hold down the <Ctrl> and C keys on the keyboard.

Proceeding from Guido van Rossum’s definition of *Pythonic* in the Python Language Reference, a common technique in Python is to loop over all elements of an *iterable* object using a for statement. Many other languages do not have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

>>>**for i in range(len(money)):**

... **print money[i]**

The Pythonic way is:

>>>**for bills in money:**

... **print bills**

The Pythonic way can be characterized as *iteration*, whereas the traditional language construct can be characterized as *counting*.

The for structure can repeat a block of operations on any iterable sequence, such as strings, lists, tuples, or user-defined iterable objects in classes.

The general form of the for statement structure is:

for a certain number of times

repeat these statements

The while structure can repeat a block of operations as long as a test condition is true.

The general form of the while statement structure is:

while a certain condition is true

repeat these statements

An object, K, is iterable if it can be successfully run with the following code. This code also shows that a counting form of iteration such as the traditional for loop structure can be implemented with a while structure (be careful of indentation, and press **<Enter>** after the last ellipsis […]):

>>>**K = [22,33,44,55]** #Lists are iterable.

>>>**c = K.\_\_iter\_\_()**  #c is the counter

>>>**while 1:**  #execute while true

... **try:**

... **item = c.next()** #get the next one

... **print item** # Do operations on each item as you count through

... **except StopIteration:**  #Nothing left

... **break**

...

22

33

44

55

>>>

Following are two simple examples of both forms of repetition. We use Way 2 (Script mode)to run them:

Save the following three lines of Python in a file named **for1.py** in the current working directory (notice that the keywords in the Python script file are in **bold** type):

**Example W19.7**

limit = [1, 2, 3, 4, 5]

**for** number **in** limit:

**print** ("number of repeats %d" % number)

Then execute Python on your shell command line with the command **python for1.py**. Your output should be:

number of repeats 1

number of repeats 2

number of repeats 3

number of repeats 4

number of repeats 5

Save the following seven lines of Python in a file named **while1.py** in the current working directory (Notice that the keywords in the Python script file are shown in **bold** type):

**Example W19.8**

s= **int**(**input**("Enter an Integer"))

i = 0

numbers = []

**while** i < s:

numbers.append(i)

i = i + 1

**print (**"numbers now: ", numbers)

Then execute Python on your shell command line with the command **python while1.py**. When prompted for an integer, type in 6 and then press **<Enter>** on the keyboard. Your output should be:

numbers now: [0]

numbers now: [0, 1]

numbers now: [0, 1, 2]

numbers now: [0, 1, 2, 3]

numbers now: [0, 1, 2, 3, 4]

numbers now: [0, 1, 2, 3, 4, 5]

Try this same script file using different input integers each time you run it to confirm that indeterminate repetition is happening.

Another interesting indeterminate repetition method that Python can implement is known as *recursion*. Basically, recursion is the repetition of a body of calculations to accumulate intermediate results, until some basic state is reached, and at that point the calculations yield the final results. The following is an example of a recursive process that calculates the factorial of an integer, implemented in Python as a function that calls, or invokes, itself an indeterminate number of times (the Python keywords are shown in **bold** type, and we are using Way 3 [Import script mode] to execute the script):

**Example W19.9**

**def** factR(n):

**if** n == 1:

**return** n

else:

**return** n\*factR(n – 1)

If you create this function definition in a file named **FactR.py** in the current working directory in which you are executing Python, then typing the following into the Python interpreter will yield the factorial of the input argument:

>>> **import FactR**

>>> **FactR.factR(7)**

5040

>>>

In-Chapter Exercises

9. What error message do you get if you supply a real number, such as 9.76, when you run the code of Example W19.9? Why do you get this error message?

10. How can you find the factorial of a real number in Python? Such as from 1.?

W19.3.6 File Input and Output

If you look back to the beginning of this chapter, at the programming tasks that Python is capable of in a Linux environment (such as systems programming, network and Internet scripting, database programming, systems administration scripting, GUI scripting, scientific statistical math programming, and data mining), the common thread which runs through all of those tasks is the ability to interface with the Linux system via utilities that work with Linux files. For example, you may be programming a statistical analysis script in Python that perhaps has its data generated from some other program or utility stored in a file somewhere in the file structure of your system. These files can be text, Unicode text, or binary, raw 8-bit bytes.

The general form of a file operation is:

name = **open**(filename, **mode**)

name.**method**(argument(s))

name.**close()**

where name is a file object name in the current procedure

**open** is the keyword that opens Python’s connection to an external file

filename is the name of the external file, which may include directory paths, and so on

**mode** is a method of accessing the file, like reading from it, or writing to it

**method** is one of several operations that can be performed on the open external file

argument(s) is(are) one or more qualifiers on the operation specified in method

**close** is the termination of connection to the external file

The mode can be r, w, or a, for reading (the default, meaning you do not need to specify this to open with read), writing, or appending to the file, respectively. The file will be created if it does not exist, and opened for writing or appending. It will be truncated when opened for writing. Add b to the mode for binary files. Add + to the mode to allow simultaneous reading and writing.

The preferred way to open a file is with the built-in open() function. Add U to the mode to open the file for input with universal newline support.

In the following examples, we illustrate some simple operations on files, such as how to open, write/read from, and close files.

In the first example, we input a string of text into a named file. Run Python and type the following three lines of code into the interpreter:

**Example W19.10**

>>> **file = open('sometext', 'w')**

>>> **file.write('This is a line of text.')**

**23**

>>> **file.close()**

>>>

Then, when you are in the same working directory in which you are executing Python, type the following line at the Linux Bash shell prompt (shown as $):

$ **more sometext**

This is a line of text.

$

In the next example, we input some integer data into an external file using your favorite text editor, and then do some Python operations on that data. Run your favorite text editor and into a file you name **somedata.txt**, type the following four integers (one integer per line):

23

33

43

54

Then, run Python and execute the following lines of code. The variable named x1 is a *list*, which we will discuss in more detail in Section W19.3.7. So, the list element x1[0] is the first element of the list that has been read from the first line in the external file. The **float** and **int** functions convert the text strings in the file to integers and real numbers:

**Example W19.11**

>>> **file = open('somedata.txt')** #the default mode is reading

>>> **x1 = file.readlines()**

>>> **x1**

['23\n', '33\n', '43\n', '54\n']

>>> **s = float(x1[0])**

>>> **s**

23.0

>>> **r = int(x1[0])**

>>> **r**

23

>>> **s + r**

46.0

>>> **file.close()**

>>>

We can also write list elements, such as numbers, to an external file. For example, the following Python code uses the write method to place three lists into a file named **listw.txt**:

**Example W19.12**

>>> **L = [1, 2, 3]**

>>> **M = [4, 5, 6]**

>>> **N = [7, 8, 9]**

>>> **F = open('listw.txt','w')**

>>> **F.write(str(L) + '\n')**

>>> **F.write(str(M) + '\n')**

>>> **F.write(str(N) + '\n')**

>>> **F.close()**

>>>

On the shell commend line, view the contents of **listw.txt**:

$more listw.txt

[1, 2, 3]

[4, 5, 6]

[7, 8, 9]

$

In-Chapter Exercises

11. What commands do you use to add the first three elements of the list x1 of Example W19.11, and print that sum?

12. What are the \n characters shown on line 4 of Example W19.11?

13. What happens if you edit the file **somedata.txt**, and enter the numbers 23, 33, 43, and 54 on a single line in your text editor instead of on four different lines, save the file, and try to perform the same action in Example W19.11 in Python on the new file **somedata.txt**?

14. After you close the file, can you still access the values 43 and 54 in any way?

15. How would you specify the third element of the list x1?

W19.3.7 Lists and the List Function

On the lowest level of the organizational scheme for Python, the *list* is an object that can contain multiple elements of possibly different types. Just like a shopping list can contain different kinds of things from a store, such as food, household goods, automotive supplies, and so on, a Python list can be made up of different types of elements, like integers, real numbers, strings, and so on; in fact, any type of Python object. For example:

>>>**A = [ 34, 'Bob', 54.76, [4,7,9]]**

is an expression that assigns the integer 34, the string “Bob”, the real number 54.76, and another list comprised of the numbers 4, 7, and 9 to a variable named A. List indices are integers, starting with 0. So, the following statements in Python yield the results:

>>>**A[2]**

54.76

>>>**A[0]**

34

>>>**A[3][2]**

9

A 2 × 2 matrix (or array) can be specified as:

B = **[[x, x],[y, y]]**

Here is Python code to create a 3 × 3 matrix (or array) named x of random numbers, using the list function that works on objects (please notice that your results will differ from the output shown here, since the numbers shown in the example are randomly generated):

**Example W19.13**

>>> **import random** # random is a function from the Standard Library

>>> **x = list(list (random.random() for i in range(3)) for j in range(3))**

>>> **x**

[[0.1455440585876967, 0.7525092872509719, 0.30168961326498955], [0.6967960669374997, 0.8715621457012694, 0.24960628313623423], [0.891389814359208, 0.9591605275600708, 0.5240885874508074]]

>>>

W19.3.8 Strings, String Formatting Conversions, and Sequence Operations

*W19.3.8.1 Strings*

Strings are a class of objects in Python that can represent text, and are basically seen in their single-quoted and double-quoted form, which are interchangeable. For example:

>>> **'program', "program's"**

('program',"program's")

>>>

To format strings in an expression, you can use the % binary operator to format values as strings according to a specific format definition. On a line of code, on the left of the % operator, put in a format string that has one or more code types. On the right of the % sign, put in objects you want to substitute in for the types.

The operator (*s* % *k*) produces a formatted string, given a format string *s* and a collection of objects in a tuple or mapping object (dictionary).The string *s* may be a standard or Unicode string. The format string contains two types of objects: ordinary characters (which are left unmodified) and conversion specifiers, each of which is replaced with a formatted string representing an element of the associated tuple or mapping.

If *k* is a tuple, the number of conversion specifiers must exactly match the number of objects in *k*. If *k* is a mapping, each conversion specifier must be associated with a valid key name in the mapping, using parentheses. Each conversion specifier starts with the % character and ends with one of the conversion characters shown in Table W19.4.

|  |  |
| --- | --- |
| **Character** | **Output Format** |
| d, I | Decimal integer or long integer |
| u | Unsigned integer or long integer |
| o | Octal integer or long integer |
| x | Hexadecimal integer or long integer |
| X | Hexadecimal integer (uppercase letters) |
| f | Floating point as [−]m.dddddd |
| e | Floating point as [−]m.dddddde±xx |
| E | Floating point as [−]m.ddddddE±xx |
| g, G | Use %e or %E for exponents less than −4 or greater than the precision |
| s | String or any object. The formatting code uses str() to generate strings |
| r | Produces the same string as produced by repr() |
| c | Single character |
| % | Literal % |

Table W19.4 String Formatting Conversions

The following example allows you to perform some basic operations on strings, such as *concatenating* them (adding their characters together), embedding escape sequences in them (to include special characters), finding their lengths (an integer representing their length), or *slicing* them (extracting smaller substring parts of them).

**Example W19.14**

>>> **a = 'programming'**

>>> **b = 'programmer\n'**

>>> **c = 'programs'**

>>> **print (a + ' ' + b + c)**

programming programmer

programs

>>> **len(a+b+c)** #len is the length operator

30

>>> **d = a[0:3] + b[3:7] + c[7]** #b[3:7] is gram

>>> **d**

'programs'

>>> **b[3:]**

'grammer\n'

>>> **print b[3:]**

grammer

>>> **q = c[:]**

>>> **q**

'programs'

>>>

*W19.3.8.2 Sequence Operations*

Three important and useful operations you can perform on sequence types of objects are *indexing*, *slicing*, and *extended slicing*. Objects such as strings and tuples are immutable and cannot be modified after creation. But, lists can be modified with the following operators, as shown in TableW19.5.

|  |  |
| --- | --- |
| **Operation** | **Description** |
| s[n] | Returns *n*th element of s |
| s[i] = x | Index assignment |
| s[i:j] = r | Slice assignment |
| s[i:j:stride] = r | Extended slice assignment |
| del s[i] | Deletes an element |
| del s[i:j] | Deletes a slice |
| del s[i:j:stride] | Deletes an extended slice |

Table W19.5 Indexing, Slicing, and Extended Slicing Operations

The following section describes and gives examples of some sequence operations on mutable objects.

Indexing into the sequence:

Gets components using offsets, where the first element indexed is at zero (0) offset.

Negative indices count backward from the end, where the last element is at offset –1.

s[0] gets the first element, s[1] gets the second element, and so on.

s[–2] gets the second from last element.

Slicing the sequence:

Extracts contiguous sections of a sequence, from i to j-i.

Slice boundaries i and j default to 0 and sequence length len(s).

s[1:4] retrieves elements from offset 1–3.

s[1:] retrieves from offset 1 until the end of the sequence object.

s[:−1] retrieves from offset 0 to the next to last element.

s[:] makes a copy of the sequence object.

Extended slicing of the sequence:

The third element is a *stride*, which defaults to 1, added to the offset of each element extracted.

s[::2] is every other item in the sequence.

s[::–1] is the reverse of the sequence.

s[4:1:–1]retrieves from offset 4, up to but not including 1, in reverse.

Slice assignments:

On mutable objects, deleting elements of the sequence and then reinserting new ones.

Iterable objects assigned to slices s[i:j] do not have to be the same size.

Iterable objects assigned to extended slices s[i:j:k] must match in size.

Here are several interactive code examples of sequence object operations on a list of integers:

>>>**m = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]**

>>>**n = m[::2]**

>>>**m**

[2, 6, 10, 14, 18 ]

>>>**p = m[::-2]**

>>>**p**

[20, 16, 12, 8, 4]

>>>**q = m[0:5:2]**

>>>**q**

[2, 6, 10]

>>>**r = m[5:0:-2]**

>>>**r**

[12, 8, 4]

>>>**s = m[:5:1]**

>>>**s**

[2, 4, 6, 8, 10]

>>>**t = m[:5:-1]**

>>>**t**

[20, 18, 16, 14]

>>>**u = m[6::1]**

>>>**u**

[14, 16, 18, 20]

>>>**v = m[5::-1]**

>>>**v**

[12, 10, 8, 6, 4, 2]

>>>**w = m[5:0:-1]**

>>>**w**

[12, 10, 8, 6, 4]

>>>

Here are interactive code examples of some further uses of formatting expressions on different objects.

**Example W19.15**

>>>**x = 400**

>>>**y = 75.142783**

>>>**z = "master"**

>>>**d = {'x':13, 'y':1.54321, 'z':'unive'}**

>>>**q = 1234567812345678**

>>>**print ('x is %d' % x)**

x is 400

>>>**print ('%10d %f' % (x,y))**

400 75.142783

>>>**print ('%+010d %E' % (x,y))**

+0000000400 7.514278E+01

>>>**print ('%(x)-10d %(y)0.3g' % d)**

13 1.54

>>>**print ('%0.4s %s' % (z, d['z']))**

mast unive

>>>**print ('%\*.\*f' % (5,3,y))**

75.143

>>>**print ('q = %d' % q)**

q = 1234567812345678

>>>

Here are more interactive code examples showing slicing operations on a list of integers:

>>>**x = [1,2,3,4,5]**

>>>**x[1] = 6**

>>>**x**

[1,6,3,4,5]

>>>**x[2:4] = [10,11]**

>>>**x**

[1,6,10,11,5]

>>>**x[3:4] = [-1,-2,-3]**

>>>**x**

[1,6,10,-1,-2,-3,5]

>>>**x[2:] = [0]**

>>>**x**

[1,6,0]

>>>

A slicing assignment may be supplied with an optional stride argument. The argument on the right side of the assignment statement must have exactly the same number of elements as the slice that is being replaced. Here are a few interactive code examples of this:

>>>**y = [1,2,3,4,5]**

>>>**y[1::2] = [10,11]**

>>>**y**

[1,10,3,11,5]

>>>**y[1::2] = [30,40,50]**

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: attempt to assign sequence of size 3 to extended slice of size 2

>>>

In-Chapter Exercises

16. Instead of printing the concatenated strings in line 4 of Example W19.15, what do you get echoed to you on the Python command line if you just type **a + b + c**?

17. The variable c has 12 characters in it. Why are there 30 characters returned for the length of the concatenation of a, b, and c?

18. What is the first index value used to extract substrings from a string?

19. What is the last index value used to extract substrings from a string?

20. What does the indexing operation [:] shown in Example W19.15 accomplish?

21. What does a[1:3] return, and why?

W19.3.9 Tuples

Similar to the list object described in Section W19.3.7, the Python *tuple* is a simple object that creates data structures using any other object type. Tuples support most of the same operations as lists, such as indexing, slicing, and concatenation. But, you cannot modify the contents of a tuple after creation, as you can with a list.

This is its most important feature as a sequence object used in data structures; that it cannot be changed. The following example uses Way 2 (Script mode)to execute the code, and shows the creation, querying, and manipulation of tuples in a database.

**Example W19.16**

With your favorite text editor, create the following file of six lines (named **data.txt**) in the current working directory in which you run Python in:

Conditions,23,45.8

Methods,12,11.75

Objects,40,17.1

Modules,1023,1.4

Dictionaries,45,120.71

Comprehensions,5,234.75

Next, using your text editor again, create the following file (named **searcher.py**) in the current working directory:

**filename = "data.txt"**

**collection=[]**

**for line in open(filename):**

**fields = line.split(",") #splits line by commas**

**name = fields[0] #create the fields**

**uses = int(fields[1])**

**value = float(fields[2])**

**card = (name,uses,value) #create the tuple**

**collection.append(card)**

**print (collection[0])**

**print (collection[3][2])**

**sum = 0.0**

**for name, uses, value in collection:**

**sum += uses / value**

**print (sum)**

Then, at the shell prompt, type **python3 searcher.py**

You should see the following output on your screen:

('Conditions', 23, 45.8)

1.4

734.9710205576091

W19.3.10 Sets

Another elementary object type in Python is the *set*, which is an *unordered* collection of objects that have no duplicate elements. The distinguishing feature of a set as an unordered sequence is that you cannot address or index into the set using the index operations on sequences we illustrated for lists and tuples. To create sets, and do some operations on them, do the following example (you can omit typing in the comments shown!):

**Example W19.17**

>>> **x = set([1.0, 2.0, 3.0, 4.0])**

>>> **x**

set([1.0, 2.0, 3.0, 4.0])

>>> **z = set([10,11,12,13,14])**

>>> **z**

set([10, 11, 12, 13, 14])

>>> **q = set("Hello")**

>>> **q**

set(['H', 'e', 'l', 'o']) #no repeated elements

>>> **a = z | q** #union of z and q

>>> **a**

set(['e', 'H', 10, 11, 12, 13, 14, 'l', 'o']) #ordered ascending

>>> **r = set([9, 8, 7, 6])**

>>> **b = q | r**

>>> **b**

set(['e', 6, 7, 8, 'H', 'l', 'o', 9]) #ordered ascending even

#though r is descending!

>>> **c = r | a**

>>> **c**

set(['e', 6, 7, 8, 9, 10, 11, 12, 13, 14, 'l', 'o', 'H'])

>>> **c = r & a** #intersection of r and a

>>> **c**

set([]) #the empty set

>>> **x.add(5.0)** #add an element

>>> **x**

set([1.0, 2.0, 3.0, 4.0, 5.0])

>>> **x.remove(4.0)**  #remove an element

>>> **x**

set([1.0, 2.0, 3.0, 5.0])

>>> **x.update([6.0, 7.0, 8.0])** #add multiple elements

>>> **x**

set([1.0, 2.0, 3.0, 5.0, 6.0, 7.0, 8.0])

>>>

In-Chapter Exercises

22. In Example W19.17, why is c finally the empty set?

23. What would the set x contain if you added three number 5.0’s with only one **x.add** command? What would the set x contain if you added three number 5.0’s with the **x.update** command?

24. Can you use slice assignment statements to reassign new values to elements of a tuple?

W19.3.11 Dictionaries

A *dictionary* is a data structure or “container” that acts like a table of objects that can be indexed into, or various parts of it can be addressed by, keys. The keys can be strings or one of several other Python objects. The container is not a sequence object, in the way that a list is. An example of dictionary creation that uses strings as keys is as follows:

>>> **function = {**

... **"name":"operator",**

... **"class" :"arithmetic",**

... **"number": 12**

... **}**

**>>> function**

{'number': 12, 'name': 'operator', 'class': 'arithmetic'}

>>>

The keys are the strings “name”, “class”, and “number”, the field data are “operator”, “arithmetic”, and 12, and the curly braces are the syntax that allows you to define the dictionary.

The order of how the keys and their field data is presented is not necessarily the same order in which you defined them!

Here is a way of extracting a value from the table function we just created, and then changing one of the data values in it:

>>> **n = function.get("number")**

>>> **n**

12

>>> **function["class"]="logical"**

>>> **function**

{'number': 12, 'name': 'operator', 'class': 'logical'}

>>>

Here is a way that lets you sort the keys in a for loop using the sortedfunction. Python has two basic ways of sorting. The sorted function works to sort any iterable object, such as entries in a Python dictionary. The sort method is a list method that works on Python lists (there is no need to type in the comments):

>>> **K = {'x':1, 'y':2, 'z':3}** #creates the dictionary

>>> **K**

{'y':2, 'x':1, 'z':3} #not a sequence, thus comes back in any order

>>>**for key in sorted(K):**

... **print (key, '=', K[key])** #indent, and press Enter twice here!

...

x = 1

y = 2

z = 3

>>>

W19.3.12 Generators

A Python functional technique of program execution that harnesses the advanced programming methodologies of data flows, streams, and process pipelines, and preserves the state of the generation of output as the function executes stepwise, is called a *generator*. A generator produces a collection of output results only when method named next() in Python3.X) is called. The next() method is executed by the Python yield statement. When a large collection of data needs to be created on the fly, perhaps in single steps, at a particular time during program execution, the generator function is invoked. The next built-in function steps you through and generates the output.

Again, this technique is necessarily for more advanced programming application. The following example shows how to create and invoke a generator function (no need to type in the comments).

**Example W19.18**

>>> **def generadd(Q):**

... **for i in range(Q):**

... **yield i** #generates the next value

... **i += 1**

...

>>> **for i in generadd(4):** #whenever function is called, #the values are generated

... **print (i)**

...

0

1

2

3

>>> **z = generadd(6)**  #this passes 6 to generadd

>>> **z**  #this will show you the compiled generator object

<generator object generated at 0x284f60f4>

>>> **next(z)** #the **next** built-in steps you through and

#generates the value

0

>>> **next(z)**

1

>>> **next(z)**

2

>>> **next(z)**

3

>>> **next(z)**

4

>>> **next(z)**

5

>>> **next(z)**

Traceback(most recent call last):

File "<stdin?", line 1. in <module>

StopIteration

>>>

The following example shows how to implement recursive generator functions in a file named **regen.py** that you create with your favorite text editor. First create the file **regen.py** as shown, and then use Way3 (Import script mode) to do in-chapter In Chapter Exercises W19.25-27:

**Example W19.19**

**def abc():**

**a = deff()**

**for i in a:**

**yield i**

**yield 'abc'**

**def deff():**

**a = ijk()**

**for i in a:**

**yield i**

**yield 'deff'**

**def ijk():**

**for i in (1, 2, 3):**

**yield i**

**yield 'ijk'**

In-Chapter Exercises

25. Give the exact Python code that would bring the three functions from Example W19.19 into the Python interpreter, given that you must use Way 3 (Import script mode).

26. Give the exact Python code that would invoke the three functions from Example W19.19 on the Python command line.

27. Give the exact Python code that would allow you to step through the invocation of the three functions from Example W19.19, to generate its output results until you reach StopIteration. List the output generated at each step through the recursion.

W19.3.13 Coroutines

In the previous section, we introduced the advanced programming technique of using generator functions, which use yield to give output results. Python generator functions can also “consume” results using a yield statement. In addition, two new methods applied to generator objects, send() and close(), create a modular “framework” for objects that consume and give values. Generator functions that define these objects are called *coroutines*. Coroutines consume values using a yield statement on the right side of an expression, as follows:

value = (yield)

With this syntax, execution pauses at this statement until the object’s send method is invoked with an argument:

coroutine.send(data)

Then, execution resumes, with value being assigned to the value of data. To signal the end of a computation, we shut down a coroutine using the close() method. This raises a GeneratorExit exception inside the coroutine, which we can catch with a try/except clause.

The next example illustrates these concepts. It is a coroutine that prints strings that match a provided template pattern:

**Example W19.20**

>>>**def grepper(template):**

...  **print ('Searching for ' + template)**

... **try:**

... **while True:**

... **x = (yield)**

... **if template in x:**

... **print (x)**

... **except GeneratorExit:**

... **print ("Done")**

...

>>> **q = grepper("Pythonista")**

>>> **q.next()**

Pythonista

>>> **q.send("After doing this section, you will be known as a Pythonista")**

After doing this section, you will be known as a Pythonista

>>> **q.send("Not a very Pythonic answer")**

>>> **q.send("Python makes C look high maintenance and too complex")**

>>> **q.close()**

Done

>>>

When we call q.send a value, evaluation resumes inside the coroutine q at the statement line = (yield), where the sent value is assigned to the variable line. Evaluation continues inside q, printing out the line if it matches, going through the loop until it encounters the line = (yield) again. Then, evaluation pauses inside q and resumes where q.send was called.

We can chain functions that send() and functions that yield together to achieve complex behaviors, similar to streaming or pipelining, illustrated in earlier chapters in this book on shell programming. In the next example, the function read splits a string named text into words and sends each word to another coroutine.

**Example W19.21**

>>> **def read(text, next\_coroutine):**

... **for line in text.split():**

... **next\_coroutine.send(line)**

**... next\_coroutine.close()**

Each word is sent to the coroutine bound to next\_coroutine, causing next\_coroutine to start executing, and this function to pause and wait. It waits until next\_coroutine pauses, at which point the function resumes by sending the next word or exiting with Done.

If we join this function together in a pipeline with the function grepper defined in Example W19.17, we can create a program that prints out only the words that match a particular word.

>>> **text = "0110 1100 0101 1000 1010 0111 1111 0001 0110"**

>>> **found = grepper('01')**

>>> **found.next()**

Looking for 01

>>> **read(text, found)**

0110

0101

1010

0111

0001

0110

Done

>>>

The read function sends each word to the coroutine grepper, which prints out any input that matches its template pattern. Within the grepper coroutine, the line x = (yield) waits for each word sent, and it transfers control back to read when it is reached.

In-Chapter Exercises

28. (a) Put the code from Example W19.20 into a file (if you have not done so already), and using Way 3 (Import script mode), invoke the coroutine grepperon the template “1201”.

(b) Then search the following using that template: “0120 1201 1020”, “3012 3013 3212”, “12010203101213012”.

(c) What prints out on your screen when you use the proper commands, similarly to what is shown in the follow-up code to the function definition in Example W19.20?

29. (a) Put the code from Example W19.21 in a file (if you have not done so already), and using Way 3 (Import script mode), invoke the coroutine read on the string “Python is the most Pythonic enterprise a Pythonista can practice”.

(b) Then search for the string “Python” using the coroutines grepper and read.

(c) What prints out on your screen when you use the proper commands, similarly to what is shown in the follow-up code to the function definition in Example W19.21?

W19.3.14 Objects and Classes

OOP is a programming model that represents concepts as *objects* that have fields (attributes that describe the object) and associated procedures known as *methods*. Objects, which are usually instances of classes, are used to interact with one another to design applications and computer programs. Some examples of OOP languages are: Smalltalk, C++, C#, Java. Perl, Ruby, PHP, and Python.

Some common terms used in OOP are as follows:

**Class**: A user-defined model for an object that defines characteristics of any object in that class. The characteristics are data members (class variables and instance variables) and methods, accessed via dot notation.

**Class variable**: A variable that is shared by all instances of a class. Class variables are defined within a class but outside any of the class's methods.

**Data member**: A class variable or instance variable that holds data associated with a class and its objects.

**Instance variable**: A variable that is defined inside a method and belongs only to the current instance of a class.

**Inheritance**: The transfer of the characteristics of a class to other classes that are derived from it.

**Instance**: An individual object of a certain class. An object that belongs to a class Circle, for example, is an instance of the class Circle.

**Instantiation**: The creation of an instance of a class.

**Method**: A special kind of function that is defined in a class definition.

**Object**: A unique instance of a data structure defined by its class. An object can comprise both data members (class variables and instance variables) and methods.

The general form of a class definition is:

classClassName**:**

'Optional class documentation string'

class\_suite

The class has a documentation string, which can be accessed via ClassName.\_\_doc\_\_.

The class\_suite consists of all the component statements defining class members, data attributes, and methods.

The following lines of interactive code very simply illustrate the inheritance model of OOP classes and its hierarchic nature. Be sure to press **<Enter>** on the second line of code. There is no need to type in the comment line numbers.

>>> **class tab: pass** #Line 1

... #Line 2

>>> **tab.name = 'Mansoor Sarwar'** #Line 3

>>> **tab.age = 25** #Line 4

>>> **print (tab.name,tab.age)** #Line 5

Mansoor Sarwar 25

>>> **x = tab()** #Line 6

>>> **y = tab()** #Line 7

>>> **x.name** #Line 8

Mansoor Sarwar

>>> **y.name = 'Alan Turing'** #Line 9

>>> **tab.name, x.name, y.name** #Line 10

('Mansoor Sarwar', 'Mansoor Sarwar', 'Alan Turing')

>>>

For beginners, a line-by-line analysis and description of this code is as follows:

Line 1: Starting with the keyword class, you name it tab. You use the keyword pass to assign the class to an empty namespace object, that is, it has no class members, attributes or methods yet. A class is an object!

Line 2: Continue by pressing <Enter>.

Line 3: You now add an attribute called name to the class tab. The class tab has no instances yet!

Line 4: You assign another attribute called ageto the classtab.

Line 5: You print out the attributes of tab.

Line 6: You now assign an instance, named x, to the class tab, which is an empty instance.

Line 7: You now assign another instance, named y, to the class tab, which is another empty instance.

Line 8: The instance x inherits the attribute name from tab. You use the dot (.) operator to connect or refer to the instance x with name “Mansoor Sarwar” in the class tab.

Line 9: You now explicitly assign the instance y, with an attribute name,the value “Alan Turing”. You use the dot (.) operator to connect the instance y with name “Alan Turing”.

Line 10: You print out the name in tab, the name referred to in x inherited from tab, and the explicitly assigned name in y. Attribute references work through the mechanisms of inheritance, and attribute assignments work on the objects to which the assignment is done.

The following is a more involved example of creating a class, and then using some methods to manipulate the objects in that class. Type the following code into a file named **firstclass.py** using your favorite text editor:

**Example W19.22**

**#!/usr/bin/python3**

**class Structure:**

**'Common base class for all Python Structures'**

**StrucCount = 0**

**def \_\_init\_\_(s, name, number):**

**s.name = name**

**s.number= number**

**Structure.StrucCount += 1**

**def displayCount(s):**

**print ("Total Structures %d" % Structure.StrucCount)**

**def displayStructure(s):**

**print ("Name : ", s.name, ", Number : ", s.number)**

Then, at the Linux shell prompt, use Way 3 (Import Script mode) to run **firstclass.py**

On the Python command line, type the following (you can leave out the comments):

**>>> import firstclass**

**>>> Stru1 = firstclass.Structure("Arithmetic Operators", 17)** #creates

#the first object

**>>> Stru2 = firstclass.Structure("Logical Operators", 10)**

#creates the second object

**>>> Stru1.displayStructure()** #displays the first object

**Name : Arithmetic Operators,, Number : 17**

**>>> Stru2.displayStructure()** #displays the second object

**Name : Logical Operators,, Number : 10**

**>>> print ("Total Structures %d" % firstclass.Structure.StrucCount)** #prints total

**Total Structures 2**

**>>> Stru1.inst = 7 #creates a new attribute of Stru1**

**>>> hasattr(Stru1, 'inst') #checks object for attribute**

**True**

**>>> getattr(Stru1, 'inst')** #gets the value of the attribute

**7**

**>>> getattr(Stru1, 'name')** #gets the value of the attribute

**'Arithmetic Operators'**

There are three things to notice about this example:

1. The variable StrucCount is a class variable whose value is shared among all instances of this class. This variable can be accessed as Structure.StrucCount from inside the class or as firstclass.Structure.StrucCount outside the class.

2. The first class method, \_\_init\_\_(), is a special method, which is called a *class constructor* or *initialization* method. Python calls this method when you create a new instance of this class.

3. You declare other class methods like normal Python functions, with the exception that the first argument to each method is s. Python adds the s argument to the list for you; you do not need to include it when you call the methods.

W19.3.15 Exceptions

Before we begin our discussion of Python exceptions, it is worth noting that there are facilities that can help you to debug your program 1) before you even submit it to the interpreter, and 2) during the execution of the program. Usually these facilities are a part of a Linux Python integrated development environment (IDE), which we have *not* been using in this chapter to keep our Python tutorial here as plain and universal with respect to our reference Linux systems as possible.

A good example of one of these facilities is automatic indentation, available in a Python IDE editor. Another is an interactive step-by-step debugging tool such as PyDebug.

With that said, an exception, or unexpected end to a program, is a Python object that represents an error. To terminate the flow of execution of a program because of some exception the interpreter has found, Python has two kinds of exception-handling structure that can end the program. These structures are:

1. Exception handling that uses for example try:…except:…else:as shown in Example W19.15, and the standard exceptions—for example, StopIteration, as shown in Example W19.13.

2 Assertions, for example using the assert statement.

The general form of try:...except:...else: is:

try:

Some operations…

...

except ExceptionI:

If there is an Exception, do this…

else:

If there is no Exception then, do this…

The following is a simple exception test example, which opens a new file in the current working directory, writes some content to the file, and then exits normally.

**Example W19.23**

>>> **try:**

... **handler = open("datafile", "w")**

... **handler.write("This is a data file for testing exception handling!!")**

**... except IOError:**

**... print ("Error: Can\'t find file or write data")**

**... else:**

**... print ("File write successful")**

... **handler.close()**

...

File write successful

>>>

In-Chapter Exercises

30. List five other standard exceptions, what general class of exception they signal, and what source you used to obtain their names.

31. Modify the code of Example W19.23 so that it writes the integers 1, 2, 3, 4, 5, 6, 7, 8, 9, and 10 as a string list to the file named **datafile**.

W19.3.16 Modules, Global and Local Scope in Functions

At a certain level of abstraction, everything in Python is a module, even Python itself. A Python module is a container or package which holds all the hierarchical objects, statements, expressions, and other modular components we spoke about in the introduction to this chapter that are necessary to accomplish an intended task or subtask. Modules may contain function definitions, class operations, and variable assignments. Basically, there are only two types of Python module: a module written by you, or a module from some external library, like the Python standard library. The standard library that is built in contains over 200 modules, and there are many other module resources available online that a programmer can use so that she does not have to “reinvent the wheel,” so to speak.

A schematic diagram of the modular construction of a possible Python program is as follows:

Python shell ---> First Module.py ---> Second Module.py, etc. ---> Standard Library

>>> commands <---objects <---objects <---objects

results

The following example, taken from the code in ExampleW19.4, shows the composition and use of a module, which happens to be a function, that you create:

If you have not already done so, type the code from Example W19.4 into a file named **arith1.py** using your favorite text editor (or rename **math1.py** as **arith1.py**). Then type the following lines of Python code at the Python command line:

**Example W19.24**

>>> **import arith1**

>>> **A = arith1.add(3.5,4.2)**

>>> **B = A + arith1.subtract(14.78,20.45)**

>>> **C = arith1.multiply(B,5.0)**

>>> **D = arith1.divide(A,C)**

>>> **D**

0.11518324607329841

>>>

Notice these things about Example W19.24’s code:

1. You use import , rather than from…import.

2. The methods in **arith1** are addressed, or “touched”, by referencing the general syntactic form arith1.function\_name. For example, by applying the method arith1.add.

In-Chapter Exercises

32. After doing Example W19.24, what result do you get if you type **b** on the Python command line, and why?

33. After doing Example W19.24, type **del arith1** on the Python command line. Then type **D = arith1.divide(A,C)** and press <Enter> on the keyboard. What result do you get, and why?

34. Edit the file **arith1.py**, and comment out all of the return statements in the functions. Then, redo the commands shown in Example W19.24. What is D equal to, and why?

A simple example of a library module from the standard library that you can import and use to execute Linux operating system commands is given next. Type in the commands shown on the Python command line (the output results given in the example may differ from what you see on your screen, depending on the specific content in the targeted directory of your Linux system):

**Example W19.25**

>>> **import os**

>>> **File = os.popen('pwd')**

>>> **File.read(50)**

'/usr/home/bob\n'

>>> **for line in os.popen('ls -la f\*'): print(line.rstrip())** #Press Enter twice!

...

-rw-r--r-- 1 bob bob 42 Feb 14 23:07 func1.py

-rw-r--r-- 1 bob bob 222 Feb 14 23:08 func1.pyc

-rw-r--r-- 1 bob bob 27 Feb 14 21:54 func2.py

-rw-r--r-- 1 bob bob 207 Feb 14 21:47 func2.pyc

...

>>>

W19.4 Practical Examples

To begin this section, it would be helpful for you to read and try to understand the following two references in the Python online documentation for the release of Python you are using to get a better “top-down” overview of how Python is structured:

1. *The Python Language Reference -- https://docs.python.org/3/reference/*

2. Python standard library, particularly the **sys** and **os** modules.

In the previous sections of this chapter, we provided an overview of the Python language and its syntax via the writing and execution of small (1–25 line) script files and functions. In this section, we will detail some of Pythonʼs practical applications in real-world computer programming with larger (25–50 line) user-written modules, functions, and scripts.

As mentioned in Section W19.1, Python can be used to accomplish tasks revolving around shell scripting, systems programming, network and Internet scripting, database programming, systems administration scripting, GUI scripting, scientific and math programming, and data mining.

We will begin by writing script files in Python that accomplish what Linux shell scripts accomplish, with the goal of familiarizing you with Python.

W19.4.1 Another Way of Writing Shell Script Files

If you have not already done so, you should read and do the examples and In Chapter Exercises in Chapters 12 and 13 to review and get a better feel for basic and advanced shell scripting in a Linux environment. In this section, we do not go over the basics of shell scripting, but provide methods and practical examples (including the rewriting of shell scripts) of how to accomplish what Linux shell scripting accomplishes, but using Python language syntax and structure. The advantages of using Python are that it is a more robust and extensible language, with many more features and capabilities than any of the Linux scripting languages. One of these features is OOP, which Bash and Tcsh scripts are not capable of.

*W19.4.1.1 Rewriting Bash and Tcsh scripts*

We start with a simple example of a Bash shell script rewritten in Python. The Bash shell script to print out whether a certain directory path exists on our file system or not is given first, and then its Python equivalent is shown in Example W19.26. You should type in and run both of these code samples, and note the output on your system:

**Example W19.26**

Bash script code

**#!/bin/bash**

**if [ -d "/usr/bin" ] ; then**

**echo "/usr/bin is a directory"**

**else**

**echo "/usr/bin is not a directory"**

**fi**

Python code

**#!/usr/bin/python3**

**import os**

**if os.path.isdir("/usr/bin"):**

**print ("/usr/bin is a directory")**

**else:**

**print ("/usr/bin is not a directory")**

On our Linux Mint 18.2 system, using Python 3.5, we ran the Python code equivalent using a variation of the alternative method of Way 2 (Script mode). We named the Python equivalent **ex26.py**. We obtained the following output (remember to make the Python equivalent executable using the **chmod u+x** command):

$ **./ex26.py**

/usr/bin is a directory

$

The important thing to notice about the Python code is that:

1. A standard library module named os is imported at the top of the script file.

2. os.path is a nested module that provides directory and pathname tools in addition to those tools in the os standard library module.

In-Chapter Exercises

35. Give the exact syntax you used on your Linux shell command line to run the Bash shell script shown in Example W19.26.

36. Referring to the online documentation, what other os module from the standard library can be used to achieve the same thing as the Python code in Example W19.26?

37. Edit the Python code for Example W19.26 and substitute the path **/usr/bin/yyy** for **/usr/bin**. What output from the program do you get when you run it after making this change?

Another simple example of a Bash shell script converted to Python is as follows.

**Example W19.27**

Bash script code

**#!/bin/bash**

**echo "Enter input: \c"**

**read line**

**echo "You entered: $line"**

**echo "Enter another line: \c"**

**read word1 word2 word3**

**echo "The first word is: $word1:"**

**echo "The second word is: $word2:"**

**echo "The rest of the line is: $word3:"**

**exit 0**

Python code

**#!/usr/bin/python3**

**import sys**

**s = input("Enter input:")**

**print ("You entered:", s)**

**r = input("Enter another line:")**

**words = r.split(‘ ‘)**

**print ("The first word is:", words[0])**

**print ("The second word is:", words[1])**

**rest = (‘ ‘.join(words[2:]))**

**print ("The rest of the line is:", rest)**

**sys.exit() #normal exit status**

On our Linux Mint 18.2 system, using Python 3.5, we ran the Python code equivalent, using a variation of the alternative method of Way 2 (Script mode). We named the Python equivalent **ex27.py**. We obtained the following output, with the supplied text shown (remember to make the Python equivalent executable using the **chmod u+x** command):

$ **./ex27.py**

Enter input: **Linux rules the network computing world!**

You entered: Linux rules the network computing world!

Enter another line: **Linux rules the network computing world!**

The first word is: Linux

The second word is: rules

The rest of the line is: the network computing world!

$

In-Chapter Exercises

38. Give two examples of list indexing or slicing used in Example W19.27.

39. Give two examples of string methods from Example W19.27.

The following is another illustration of taking a Bash shell script and converting it to Python.

**Example W19.28**

Bash script code

**#!/bin/bash**

**echo "The command name is: $0."**

**echo "The number of command line arguments passed as parameters is: $#."**

**echo "The value of the command line arguments are: $1 $2 $3 $4 $5 $6 $7 $8 $9."**

**echo "Another way to display values of all the arguments: $@."**

**echo "Yet another way is: $\*."**

**exit 0**

Python code

**#!/usr/bin/python3**

**import sys**

**x = (sys.argv)**

**print ("The command name is: ", sys.argv[1])**

**print ("The number of command line arguments passed as parameters is: ", len(sys.argv[1: ])**

**print ("The value of the command line arguments are: ", x[1: ])**

**print ("Another way to display values of all the arguments: ", sys.argv[1: ])**

**print ("Yet another way is: ", sys.argv[slice(1,9)])**

**sys.exit ( )**

On our Linux Mint 18.2 system, using the default Bash and Python 3.5, we ran both the Example W19.28 Bash script code, and its Python code equivalent (using a variation of the alternative method of Way 2 (Script mode). We named the Bash version **ex28.bash**, and the Python equivalent **ex28.py**. We obtained the following output, with the argument lists shown (remember to make the Bash script file and its Python equivalent executable using the **chmod u+x** command):

$ **./ex28.bash a b c d e f g h i j k l m n**

The command name is: ./ex28.bash.

The number of command line arguments passed as parameters is: 14.

The value of the command line arguments are: a b c d e f g h i.

Another way to display values of all the arguments: a b c d e f g h i j k l m n.

Yet another way is: a b c d e f g h i j k l m n.

$

$ **./ex28.py a b c d e f g h i j k l m n**

The command name is: ./ex28.py

The number of command line arguments passed as parameters is: 14

The value of the command line arguments are: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']

Another way to display values of all the arguments: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n']

Yet another way is: ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']

$

Similar to the Bash shell in syntax and structure, the Tcsh shell has functional capabilities that can be implemented easily by Python. The following is a modified version of a of a script file, if\_demo2, that we presented in Chapter 14, whose syntactic structure and program functionality are converted to a Python script file that you should run using a variation of the alternative method of Way 2 (Script mode):

**Example W19.29**

Tcsh shell code

**#!/bin/tcsh**

**if ( ( $#argv == 0 ) || ( $#argv > 1 ) ) then**

**echo "Usage: $0 ordinary\_file"**

**exit 1**

**endif**

**if ( -f $1 ) then**

**set filename = $argv[1]**

**set fileinfo = `ls -il $filename`**

**set inode = $fileinfo[1]**

**set size = $fileinfo[6]**

**echo "File Name: $filename"**

**echo "Inode Number: $inode"**

**echo "Size (bytes): $size"**

**exit 0**

**else**

**echo "$0: argument must be an ordinary file"**

**exit 1**

**endif**

Python code

**#!/usr/bin/python3**

**import os**

**import sys**

**if len(sys.argv) == 1 or len(sys.argv) > 2: #check for no/too many args**

**print ("Usage: ", sys.argv[0], " ordinary file")**

**sys.exit(1)**

**if os.path.isfile(sys.argv[1]): #bingo, get stats**

**filename = sys.argv[1]**

**fileinfo = os.stat(filename)**

**print ("Filename inode size")**

**print (" ")**

**print (filename, fileinfo.st\_ino, fileinfo.st\_size)**

**sys.exit(0)**

**else: # argument must something else!**

**print (sys.argv[1], " argument must be an ordinary file")**

**sys.exit(1)**

On our Linux Mint 18.2 system, we ran the Python code equivalent using a variation of the alternative method of Way 2 (Script mode). We named the Python equivalent **ex29.py**. We obtained the following output, where there was a file named ex27.py in the current working directory, but no file named lab1 (remember to make the Bash script file and its Python equivalent executable using the **chmod u+x** command):

$ **./ex29.py**

Usage: ./ex29.py ordinary file

$ **./ex29.py lab1**

lab1 argument must be an ordinary file

$ **./ex29.py ex27.py**

Filename inode size

ex27.py 3541279 317

$

W19.4.2 Basic Web Server and User File Maintenance

A very useful and practical example of using built-in Python 3 capabilities on your system is the http.server module that comes standard with Python 3. The value of using this module for an ordinary user of a Linux system is that you can quickly and easily implement a system service that can be programmed, configured, and controlled by that ordinary user.

Another very useful and important aspect of an ordinary user’s interaction with a Linux system is how effectively you can maintain the files on your system. The Python standard library, and many user-written libraries and modules, can help you do this efficiently in Linux. You can utilize the extensive syntax and multi-paradigm programming capabilities of Python to go far beyond the capabilities of doing operating system and file maintenance available in any of the Linux shell programs. This section assumes that you have already read and done the In Chapter Exercises and problems in Chapters 4 through 9 that deal with file manipulation. User file maintenance consists of creating, saving, organizing, and deleting files on your system, in your own account. In particular, Chapter 17 describes and details how the same things can be done on a system-wide level, by the system administrator. The following sections in this chapter are a good preparation for what is shown in that chapter.

W19.4.2.1 Web Server Example

In this section we show an ordinary user how to create and use a web server using a built-in Python module.

**Example W19.30**

A Simple Web Server Implemented in Python

The http.server module that comes with Python is a simple HTTP server that

provides standard GET and HEAD request handlers. An advantage with the Python built-in HTTP server is that you don't have to install and configure anything. The only thing that you need is to have Python installed on your system, which it is by default on all of our representative Linux systems.

Step 1. To start an HTTP server on port 8000 (which is the default port), type the command:

$ **python3 -m http.server**

This will serve files and directories to a browser, whose URL is set to your IP address and port 8000, where the files and directories are located in the current working directory when you typed the above command.

You can also change the port to some other valid port number, for instance 8096, by typing this command:

$ **python3 -m http.server 8096**

Step 2. To share other files and directories on your network, or the Internet, in a terminal, cd into whichever directory you wish to have accessible via browsers and HTTP. For example:

$ **cd /home/bob/webex**

$ **python3 -m http.server**

Serving HTTP on 0.0.0.0 port 8000 ...

Step 3. Then in a web browser on your Linux system, put in either of the following two addresses:

http://your\_ip\_address:8000

or to the localhost:

http://127.0.0.1:8000

If you don't have an index.html file in the current working directory, then all files and directories in /home/bob/webex will be listed. If there’s an index.html file in webex (which in our case, there is), you will see the content of that index.html page displayed in the web browser. As long as the Python HTTP server is running, the terminal will update with messages as they are loaded from the server. These messages will be standard http logging information (GET and PUSH), 404 errors, IP addresses, dates, times, and a subset of the journal messages for the webex daemon example shown in this chapter.

*In-Chapter Exercise*

*40. Use the steps shown in Example W19.30 to implement and test a basic web server on your system.*

*W19.4.2.2 Backing up Your Files*

We will not go into the general necessity of backing up your files on your Linux system as a part of maintaining that computer system, because the reasons for that are obvious to even novice users.

According to Linux system professionals, there is an easy-to-remember and important set of considerations you must make when backing up the system as an ordinary user, and perhaps even as the system administrator. This set of considerations can be posed in simple question form as “How, What, Why, When, Where, and Who?” Some of the answers to these simple questions can be dovetailed together, and we give a selected list of example answers as follows:

“How” means on a local disk, to Dropbox, to a USB thumb drive manually, to another computer on your home network, automatically by *cron*, to another hard disk manually, totally, incrementally, to RAID, or any variant and combination of these.

“What” means just some of your personal files, all of them, only certain kinds of documents, your entire home directory, the whole disk drive, multiple disk drives, and so on.

“Why” means deciding on the relative importance of “What” you are backing up.

“When” means hourly, once a day, once a week, once a month, every time you save a particular file, and at what time exactly, like 3 a.m.

“Where” means very much the same thing as “How”.

“Who” means you personally, automatically by cron, the designated system administrator, Dropbox.com.

To give you a notion of a prudent strategy to deploy in backing up your own user files, the file that contains the words you are reading right now was archived in the following manner:

1. Saved at regular intervals to the hard drive on a local computer

2. Saved periodically to a USB thumb drive mounted on that local computer

3. Saved periodically to another hard drive on another computer attached to the local area network

4. Saved to Dropbox.com

We will use the rsync command to accomplish our backup strategies in this section. This command is similar to cp, except that it is more efficient and faster.

Most importantly, rsync “synchronizes” two files or directory structures so that changes in one are reflected in the rsync duplicate, either locally between drives, or remotely over a local area network (LAN) or the Internet.

It can copy locally or to/from another host over any remote shell, particularly ssh. It has a large number of options that control every aspect of its behavior and permit very flexible specification of the set of files to be copied. The rsync command finds files that need to be transferred using a “quick check” algorithm (by default) that looks for files that have changed in size or in last-modified time. We encourage you to consult the rsync manual page for more information.

The general forms of the rsync command are:

Local:

rsync [OPTION(S)...] SRC... [DEST]

Across a network:

Pull: rsync [OPTION(S)...] [USER@]HOST:SRC... [DEST]

Push: rsync [OPTION(S)...] SRC... [USER@]HOST:DEST

where OPTION(S) are the valid options for the rsync command, SRC is the source file or directory, and DEST is the destination path.

The next five examples will use Python standard library modules, and embed Bash shell commands in a Python “wrapper” (Linux shell command(s) embedded in Python code). The examples primarily use the python **os.system** module to:

1. Back up a single file on the hard disk to a mounted USB thumb drive

2. Back up a single directory beneath your home directory on the hard disk to a directory on a mounted USB thumb drive

3. Back up a single directory beneath your home directory on the hard disk to another network location on your local area network in Push mode

4. Back up a directory on the hard disk to a mounted USB thumb drive in a rolling, incremental scheme that creates “snapshots” of the source directory anytime the Python script is run

5. Customize a system command to show permissions of files in the current working directory that match a certain pattern

The following simple example shows you how to use Python to back up a single file on your hard disk to a USB thumb drive you mounted and attached to your system’s file system . It assumes you have an ordinary file in the current working directory named **ex28.bash**, and that the destination path on the USB thumbdrive is **/media/bob/Sony\_16AS1**.

**Example W19.31**

>>>**import os**

>>>**os.system('rsync -av ex28.bash /media/bob/Sony\_16SA1')**

sending incremental file list

ex28.bash

sent 403 bytes received 35 bytes 876.00 bytes/sec

total size is 298 speedup is 0.68

0

>>>

The following example shows you how to use Python to back up an entire directory on your hard disk to the USB thumb drive you have mounted and attached to your system. It assumes you have a directory under the current working directory named **W19**, and that the destination path on the USB thumb drive is **/media/bob/B405-01CE**.

**Example W19.32**

>>>**import os**

>>>**os.system('rsync -av W19 /media/bob/B405-01CE')**

sending incremental file list

W19/

W19/.~lock.W19\_new.docx#

W19/Chap\_W19.doc

W19/Chap\_W19.xls

W19/Fig\_W19\_1.png

W19/Screenshot from 2018-04-02 15-49-22.png

W19/Section\_W19\_4\_4.docx

W19/Section\_W19\_4\_5.docx

Output Truncated…

W19/python\_threads/thread4.py

sent 1,405,340 bytes received 1,024 bytes 562,545.60 bytes/sec

total size is 1,401,044 speedup is 1.00

0

>>>

The following example shows you how to use Python to back up an entire directory on your hard disk to a remote location on your local area network. It assumes:

1. You have the **ssh** daemon running on both your local and remote host, and that you have successfully logged into the remote host and previously exchanged keys between local and remote host.

2. You have a directory under the current working directory named **W19** with some files in it

3. Where a password is asked for, you type in your password on the remote host

4. That the destination path to the remote host is **bob@192.168.0.25:/home/bob**

**Example W19.33**

>>>**import os**

>>>**os.system('rsync -av -e ssh W19 bob@192.168.0.25:/home/bob/')**

bob@192.168.0.25's password:

sending incremental file list

W19/

W19/.~lock.W19\_new.docx#

W19/Chap\_W19.doc

W19/Chap\_W19.xls

W19/Fig\_W19\_1.png

W19/Screenshot from 2018-04-02 15-49-22.png

W19/Section\_W19\_4\_4.docx

W19/Section\_W19\_4\_5.docx

Output Truncated...

W19/python\_threads/thread4.py

sent 1,405,562 bytes received 1,024 bytes 216,397.85 bytes/sec

total size is 1,401,264 speedup is 1.00

0

>>>

In-Chapter Exercise

41. Repeat the operations shown in Examples W19.31 through W19.33, substituting file names and directory paths on your computer system and local network for those shown in the examples. When backing up files to a USB thumb drive is finished, be sure to unmount that drive before removing it from the USB connector on your machine.

The following example shows you how to create a rolling backup scheme of “snapshots” of a directory on your hard disk, and archive the contents of the directory to multiple (five) backup directories on the USB thumb drive you mounted and attached to your Linux system. It is assumed that the source directory, containing some files, already exists. Every time you execute this Python script, it recycles (in other words, deletes) the oldest (fifth) archived directory, and creates a new full backup with the rsync command:

**Example W19.34**

**#!/usr/bin/python3**

**import os**

**import shutil**

**target = "/media/bob/B405-01CE/"**

**i = 1**

**while i <= 5:**

**temp\_path = target + str(i) + "/"**

**if not os.path.exists (temp\_path):**

**try:**

**os.makedirs (temp\_path)**

**print ("Created " + temp\_path)**

**except:**

**print (" Could not create " + temp\_path)**

**i = i + 1**

**print ("Deleting the oldest archive")**

**shutil.rmtree (target + "5")**

**print ("Recycle the backups")**

**os.rename (target + "4", target + "5")**

**os.rename (target + "3", target + "4")**

**os.rename (target + "2", target + "3")**

**os.system('cp -a ' + target + "1" + " " + target + "2")**

**os.system('rsync -av /home/bob/python\_threads/' + " " + target + "1")**

When we ran the code from Example W19.34 on our Linux Mint 18.2 system for the first time, using the pathnames to the source directory for the backup, and the USB drive, as shown in the code, we got the following output-

$ **python3 ex34.py**

Created /media/bob/B405-01CE/1/

Created /media/bob/B405-01CE/2/

Created /media/bob/B405-01CE/3/

Created /media/bob/B405-01CE/4/

Created /media/bob/B405-01CE/5/

Deleting the oldest archive

Recycle the backups

sending incremental file list

./

thread1.py

thread2.py

thread3.py

thread4.py

sent 3,930 bytes received 95 bytes 8,050.00 bytes/sec

total size is 3,612 speedup is 0.90

$

Here is another example of carrying out simple system administration, in this case using Python 2.7.12. It customizes a shell command to show permissions set on files in the current working directory that match a certain pattern:

**Example W19.35**

**#!/usr/bin/python**

**import stat, sys, os, string, commands**

**try:**

**#Getting search pattern from user and assigning it to a list**

**pattern = raw\_input("Enter the file pattern to search for:\n")**

**#defining a 'find' string and assigning results to a variable**

**commandString = "find " + pattern**

**commandOutput = commands.getoutput(commandString)**

**findResults = string.split(commandOutput, "\n")**

**#output find results, along with permissions**

**print "Files:"**

**print commandOutput**

**print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"**

**for file in findResults:**

**mode=stat.S\_IMODE(os.lstat(file)[stat.ST\_MODE])**

**print "\nPermissions for file ", file, ":"**

**for level in "USR", "GRP", "OTH":**

**for perm in "R", "W", "X":**

**if mode & getattr(stat,"S\_I"+perm+level):**

**print level, " has ", perm, " permission"**

**else:**

**print level, " does NOT have ", perm, "permission"**

**except:**

**print "Error - check your input of file matching pattern"**

When we ran Example W19.35 (named exW19\_35.py) on our Linux Mint 18.2 system, using the input shown, we got the following results:

$ **python W19\_35.py**

Enter the file pattern to search for:

**\*.pem**

Files:

clone\_key.pem

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Permissions for file clone\_key.pem :

USR has R permission

USR does NOT have W permission

USR does NOT have X permission

GRP does NOT have R permission

GRP does NOT have W permission

GRP does NOT have X permission

OTH does NOT have R permission

OTH does NOT have W permission

OTH does NOT have X permission

$

We present a problem at the end of this chapter, Problem W19.15, that allows you to convert this Python 2.7 script file into a Python 3.X script file.

In summary, we used rsync to backup a file and directories from the hard drive to a USB thumb drive, to a remote host on the local network, and in a rolling scheme to a USB thumb drive. We have also illustrated, using Python 2.7.12, a simple Linux file maintenance command useful for system administration. In Chapters 17, and W26we explore more robust strategies and examples of how to back up user files and system files, and carry out common system administration chores using shell commands and Python code.

W19.4.3 Graphical User Interfaces with Python and tkinter Widgets

Up until this point in the chapter, we have interacted with Python in a text-based manner, where we typed commands into the Python shell, or into a text editor, and executed Python to view the output results as text. In this section, we will build a “widget”-based (widget is short for window gadget) GUI with Python, where we still create the Python code as text, but see the resulting output of the Python script in the form of widget graphics. We use tkinter as the GUI interface, or toolkit. tkinter is an abbreviation for the “TK interface”. “TK”, which is short for “toolkit”, is a platform independent, customizable, and configurable GUI library. The Python module tkinter allows Python programs to interface with the TK libraries.

To accomplish this, we assume the following:

1. That you are using a GUI environment to interact with your Linux system. For example, on our Linux system, Linux Mint 18.2, the default desktop environment when we installed the system was the Cinnamon Desktop. We had a GUI capability when we programmed Python GUI’s.

2. That you have Python 3.X on your Linux system, as we determined in Section W19.2. The command line sessions, In-Chapter Exercises, Examples, and Questions and Problems found in this, and in the following sections of the chapter, all require that you have at least this version of Python installed on your Linux system.

Our coverage of GUI construction with tkinter uses Python 3.5 on Debian-family systems, and Python 3.4 on CentOS 7, and the appropriate tkinter module for those versions of Python. Depending upon the default Python 3.X you are using, and given the specific Linux distribution you are executing it on, the tkinter installation instructions vary as shown in Section W19.4.3.1.

W19.4.3.1Installing tkinter

The standard widget GUI package that works with Python is named tkinter in Python 3.X. For you to add widget GUI components and functionality to Python, you must be able to import this component package into your Python session.

In this section, as much as is possible and practical given the Python GUI scripts we show, we add widgets from a “styled” widget set. These were available since Tk version 8.5, and are named ttk widgets.

You can easily test to see if your default Python system has tkinter available. On the Python command line, type:

**>>> import tkinter**

If you get an error message that the module does not exist, or some similar error message, then you must install the tkinter package. On a Debian-family Linux system, such as Linux Mint 18.2, use the apt command as follows:

$ **sudo apt-get install python3-tk**

On a CentOS 7 Linux system, such as CentOS 7.4, as the root user, use yum as follows:

[root@centos]# **yum install python34-tkinter**

W19.4.3.2 tkinter Toolkit Graphical User Interfaces

As shown in the chapters in the printed book, a GUI allows you to use a mouse or other pointing device (such as your finger on a tablet), rather than just the keyboard, to interact with programs that are running.

And, as implied in Section W19.4.3, generally when you use a CUI, such as during a Bash shell session, and you execute a command, the tasks that command accomplish do so in some structured order, either in a determinate (counting-wise) or indeterminate (logic-wise) manner. Then the command eventually terminates. But the Python GUI programs we show in this section create screen widgets that are displayed on screen, and then the Python script that generates the icons and widgets waits for the user to do something with them. This kind of programming structure is event-driven. GUI programs are event-driven.

A meaningful and useful Python-scripted GUI program always has some form of this basic content and structure:

1. Data of some sort is produced or already exists, for example, in the form of numbers, or database text. This data can be generated in a variety of ways. It could be generated by Python user-written modules placed before the GUI-constructing components in the Python script, which is the style of Python GUI programming shown in all of our examples. It could also be input interactively while the Python GUI script is running, it could use preexisting data generated beforehand by some other application, or it could even use a C program that generates data within the Python environment, while the Python script is executing.

2. Widgets are created, and are organized within some encompassing window frame, and then displayed on screen to the user. These widget objects are instances of master widget classes in tkinter, and contain GUI functions that are selected for, and specify, particular kinds of responses to user input, and possible data-generating application events.

3. There is a “hooking”, or tying together, of the instances from Step 2. that utilize “callback” GUI functions to access the data you want to display or otherwise modify. For example, you can use any “callable” Python object as a callback. These might be ordinary Python functions, bound methods, lambda expressions, and and any other callableobject.

This is the most critical, and therefore the most useful aspect, of Python GUI programming.

4. Steps 2. and 3. exist inside of an infinite event-driven loop, that processes user events. When a user event, such as a mouse button press, occurs, the event-driven loop invokes the function associated with a specific event.

5. Data is modified or produced with no apparent visible change on screen, or with graphical changes occuring on screen. This is the “request” step.

6. The loop continues to wait for events, and process them, until a possible termination event happens.

As we present in Section W19.4.3.3, tkinter widgets can be chosen and constructed to display and be hooked up to data using OOP, where the widgets are object instances of master classes of tkinter widgets. But the style and execution of the Python GUI script file itself can take on a procedural/imperative programming approach. We choose, as much as possible, to show the procedural/imperative approach because it is easier for beginners, particularly people who are not familiar with the OOP model of programming.

Python does not have core modules that implement GUI, event-driven-programming. GUI programming is implemented using imported modules, such as those from the tkinter toolkit.

To summerize the above steps., a user presses a mouse button to signify a graphical “pick” in a widget, within an encompassing tkinter-produced frame shown on-screen. That choice, or “event”, is recognized by and acted upon by the tkinter function controlling that widget. This event may generate a “request” for graphical service, and this request displays the data or graphics within the tkinter frame. We show a very similar situation in Chapter W28, in the X window System Event-Request Model.

W19.4.3.3 Basic Widget Construction Techniques

Widgets are designed images on the screen display, and they have a particular “style”, depending on the details of what information and kind of interactivity you want them to participate in. The “style” of a widget is controlled by many complex graphical components: the X window System, the display manager used, and the operating system itself. The tkinter module contains two versions of widgets: a default or “generic version,” which makes widgets look the same regardless of what windowing system, operating system, or display manager is running. tkinter also implements widgets that emulate a particular windowing system, operating system, or display managers “style”. Which tkinter modules you import determines which widget sets are available to generate events and callbacks. The standard way of doing this uses the module tk to access the “generic” widgets, and the module ttk to access the “styled” widgets. You must import the tk module because it allows you to create a “root”, or outer-most frame window. You optionally import the ttk module if you want the “styled” widgets.

Note, as is done in some of the following examples, that you can use both the tk and ttk widgets in a GUI script file.

To use the "generic" widgets, include the following import statement at the top of the script file-

**import tkinter as tk**

To use the "styled" widgets, include the following import statement at the top of the script file-

**from tkinter import ttk**

The following two tables list the standard, pre-defined widgets in the tkinter module.

The listing of widgets in Table W19.6 are used for user input. You can choose between the tk and ttk widgets. But sometimes you must use the tk version because the equivalent ttk versions don’t exist.

|  |  |
| --- | --- |
| Widget Name in tk and ttk | Function |
| tk.Button, ttk.Button | Display a button to execute some operation. |
| tk.Menu | Produce a top level pulldown or popup menu. |
| ttk.Menubutton | Display a popup or pulldown menu of buttons. |
| tk.OptionMenu | Display a popup menu, and a button to activate it. |
| tk.Entry, ttk.Entry | Enter one line of text in an object. |
| tk.Text | Display and edit formatted multi-line text. |
| tk.Checkbutton, ttk.Checkbutton | Display on-off, or True-False selections in an object. |
| tk.Radiobutton, ttk.Radiobutton | Display on-off selections in an object. |
| tk.Listbox | Display one or more alternatives from a list of choices. |
| ttk.Combobox | Display a text field with a pulldown list of values. |
| ttk.Notebook | Manages a collection of windows and displays a single one at a time. |
| tk.Scale, ttk.Scale | Allows selection a numerical value via a “slider” on a scale. |

Table W19.6 tkinter tk and ttk Input Widgets

The following widgets display information, but don’t allow any user interaction:

|  |  |
| --- | --- |
| Widget Name in tk and ttk | Function |
| tk.Label, ttk.Label | Display a fixed text string, or image. |
| tk.Message | Display a fixed multi-line text string. |
| ttk.Separator | Display a horizontal or vertical separator line. |
| ttk.Progressbar | Shows the relative progress of some operation. |
| ttk.Treeview | Display a hierarchy of items as a tree. |

Table W19.7 tkinter tk and ttk Information Display Widgets

When we wrote this book, we found out the version of the Tk toolkit by using the following commands in Python Version 3.5.2 running on our Linux Mint system:

$ **python3**

Python 3.5.2 (default, Nov 23 2017, 16:37:01)

[GCC 5.4.0 20160609] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> **import tkinter**

>>> **tkinter.TkVersion**

8.6

>>>

At the time we wrote this book, the best source of documentation for tkinter can be found at the following URL- https://docs.python.org/3/library/tk.html

W19.4.3.4 General Form of a Widget Call, and a Primary Example

As shown in Section W19.4.3.2, there are several parts to tkinter widget GUI script file construction, but the basic parts follow this outline:

The first part is generating data. The second part is constructing your widgets, using a set of universal constructor tools and the tkinter widget method set. The third part is “hooking” the data of the first part to the widget constructors of the second part, all within the context of an infinite event-loop.

In this section, we show the general form of widget instancing. We then show an outline of tkinter GUI scripting, and also a primary example of that.

tkinter widgets can be constructed to display and be hooked up to your Python application code, either purely using OOP, or they can be constructed using a procedural/imperative programming approach. As stated in Section W19.4.3.2, we choose the latter approach because it is easier for beginners. By the time you do all the examples we present here, you should be able to see that OOP is an essential, but not mandatory, part of tkinter GUI programming.

The general form of using a tkinter widget is as follows:

>>> **widget = Widget.method (master, option=value, option=value,...)**

where:

**widget** is the name assigned in your Python script to the particular instance you are creating and using

**Widget** is the master instance of a widget class in the tkinter module

**method** is the “operation” of instancing some master instance of the widget

**master** is the “container”, or frame, to which our instance of the widget belongs, or is attached to

**option** is a graphical entity or modifier that gives attributes to your particular instance

**value** is one of the characteristics that the option can take on

After importing the tkinter module, the first thing you need to do is create an object, which is an encompassing frame or window, for the display of your data. This is done by creating a Tk object with the following generic line of Python code:

window\_object\_name = tkinter.Tk()

Then you create widgets you have selected for their applicability to your purposes, and add them to the frames widget hierarchy.

For example, to create a button widget instance, you would call either the tk or the ttk Button method. You need to specify the window\_object\_name as the first argument:

cmd\_button = tk.Button(window\_object\_name, text="Some text.")

cmd\_button = ttk.Button(application\_window, text="Some text.")

The arguments needed to create each kind of widget varies, so you will need to refer to the Python documentation we specified in Section W19.4.3.2 for each specific widget type shown in brief in Table W19.7.

The following simple example shows a complete tkinter Python script, and the widget it creates. You should create (in a file with your choice of name), and then execute these six lines of code using Way 2 (Script mode). Line numbers are shown for reference in the following code.

**Example W19.36**

1 **import tkinter**

2 **from tkinter import ttk**

3 **w = tkinter.Tk()**

4 **w.title("Python GUI")**

5 **ttk.Label(w,text="My first tkinter gui window").grid(column=0, row=0)**

6 **w.mainloop()**

****

Figure W19.1 First Python GUI Display

To close the widget, just click on the “destroy window” button (in Figure W19.1, the X in the upper right-hand corner) in your style of GUI window in which the widget was created. You may have to expand the window to see all of the window manipulation buttons. The following universal traits of a tkinter widget GUI script file that are illustrated in Example W19.36 are as follows:

1. As specified in Section W19.4.3.2, tkinter programming is event driven, meaning you invoke tkinter and put it in a *wait state*, where it waits for an event like a mouse button click on the destroy window button, or a keyboard entry, and so forth. The widgets you create and which remain on screen generally only do so while tkinter is waiting for an event to happen, or until you destroy the window. You can also construct exit handling events in your script to close the widget and its window.

2. Lines 1 and 2 import all tkinter modules, and the ttk styled widgets, into the current session.

3. Line 3 creates the frame within which all widgets will exist, and assigns it a name w.

4. Line 4 gives a title, “Python GUI”, to the frame.

5. Line 5 instances the ttk Label method, assigns it a title “My first tkinter gui window, and then uses the grid geometry manager to position the label in the frame w. There are three geometry managers available in tkinter: pack, grid, and place. The grid geometry manager, which treats every window or frame as a table of rows and columns, gives you the greatest control over where widgets and their components are placed.

6. Line 6 starts off the event-driven loop that tkinter enters, and constructs the Label widget, with your modifying options, on screen.

In-Chapter Exercise

42. Install tkinter on your Linux system, and do Example W19.36. Then modify it to execute any other simple widget display you find interesting and useful.

W19.4.3.5 Hooking Tkinter Widgets to Applications in Python: Examples

Given the widget method set of available core widgets in tkinter, and armed with your knowledge of Python programming to this point, you are ready to do the following examples. The following simple example allows a user to add two real numbers that have been entered interactively. Line numbers and comments are for reference only, and are not executed.

**Example W19.37**

1 **import tkinter as tk**

2 **from tkinter import ttk**

3 **from functools import partial**

# This is the data generating module, which computes the sum.

5 **def add\_it(label\_result, n1, n2):**

6 **num1 = (n1.get())**

7 **num2 = (n2.get())**

8 **result = float(num1)+float(num2)**

9 **label\_result.config(text="Sum = %f" % result)**

10 **return**

# This “grids” the widget object where indicated, then returns it.

11 **def mkgrid(r, c, w):**

12 **w.grid(row=r, column=c, sticky='news')**

13 **return w**

14 **root = tk.Tk()**

15 **root.title('Real Number Adder')**

# The rest hooks the adder into the grid manager widgets.

16 **add1\_lab = mkgrid(0, 0, ttk.Label(root, text="addend 1",anchor='e'))**

17 **add2\_lab = mkgrid(1, 0, ttk.Label(root, text="addend 2",anchor='e'))**

18 **add1= mkgrid(0, 1, ttk.Entry(root))**

19 **add2= mkgrid(1, 1, ttk.Entry(root))**

20 **spacer = mkgrid(0, 2, ttk.Label(root, text=''))**

21 **labelResult = ttk.Label(root)**

22 **labelResult.grid(row=7, column=2)**

23 **add\_it = partial(add\_it, labelResult, add1, add2)**

24 **add\_but = mkgrid(1, 2, ttk.Button(root, text="Add them",command=add\_it))**

# Starts the root main event loop

25 **root.mainloop()**

A line-by-line description/explanation of Example W19.37 is as follows:

Lines 1. & 2. Import all tkinter modules, and the ttk styled widgets, into the current session.

Line 3. Imports the partial function module.

Lines 5. through 10. Defines the function add\_it that computes the real number sum.

Lines 11. through 13. Defines a function that will specify grid locations for the grid manager.

Line 14. Creates the frame within which all widgets will exist, and assigns it a name “root”.

Line 15. Adds a title, “Real Number Adder”, to the frame “root”.

Lines 16 & 17. Constructs 2 Labelwidgets, to designate the input locations for the addends.

Line 18. &19. Allows input of both addends, and constructs them using the Entry **w**idget.

Line 20. Places a spacer at grid position 0,2.

Lines 21. and 22. Establish a ttk label in the root window for the sum.

Line 23. Uses the Python partial function to call the add\_it function with values of add1 and add2.

Line 24. Defines a ttk button at grid location 1,2, that calls the function add\_it to find the sum when the button is “pressed” to generate an event.

Line 25. Starts the event loop, creating the root window and all widgets defined above.

Even though Example W19.37 uses procedural programming syntax and data abstraction, after having examined the line-by-line description/explanation of it, you should begin to see that tkinter GUI scripts are basically composed of OOP class instance objects. All of the methods applied to those instances come from the methods on the core widgets in tkinter.

The following example will construct a Fahrenheit-to-Celsius temperature conversion GUI with tkinter. A line-by-line description/explanation of the script follows the code (see Figure W19.2). Line numbers and comments are for reference only, and are not executed.

**Example W19.38**

1 **import tkinter as tk**

2 **from tkinter import ttk**

# This function computes the Celsius temperature from the Fahrenheit.

3 **def findcel():**

4 **famt = ftmp.get()**

5 **if famt == '':** #not double quote, 2 single quotes

6 **cent.configure(text='')**

7 **else:**

8 **famt = float(famt)**

9 **camt = (famt - 32) / 1.8**

# A method (configure) applied to cent to convert camt to a string (str(camt)).

10 **cent.configure(text=str(camt))**

# This grids the widget object where indicated, then returns it.

11 **def mkgrid(r, c, w):**

12 **w.grid(row=r, column=c, sticky='news')**

13  **return w**

14 **root = tk.Tk()**

15 **root.title('Temp Conversion')**

# The rest hooks the Temperatures into the grid manager widgets.

16 **flab = mkgrid(0, 0, ttk.Label(root, text="Fahrenheit Temperature",anchor='e'))**

17 **clab = mkgrid(1, 0, ttk.Label(root, text="Celsius Temperature",anchor='e'))**

18 **ftmp = mkgrid(0, 1, ttk.Entry(root))**

19 **cent = mkgrid(1, 1, ttk.Label(root, text="", relief='sunken',anchor='w'))**

20 **elab = mkgrid(0, 2, ttk.Label(root, text=''))**

21 **fbut = mkgrid(1, 2, ttk.Button(root, text="Compute Celsius",command=findcel))**

# Starts the root main event loop

22 **root.mainloop()**

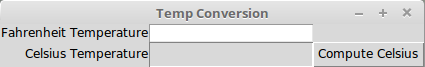
****

Figure W19.2 Tk Temp Conversion Display

Here is a line-by-line description/explanation of the code of Example W19.38.

Lines 1. & 2. Imports all tkinter modules, and the ttk styled widgets, into the current session.

Lines 3. through 10. Defines the function findcel, that does the temperature conversion from F to C. Note that lines 5. and 6. ensure that if an empty string is entered for famt, then cent is empty as well. Line 10. converts cent to text.

Lines 11. through 13. Defines a function that will specify grid locations for the grid manager.

Line 14. Creates the frame within which all widgets will exist, and assigns it a name “root”.

Line 15. Adds a title, “Temp Conversion”, to the frame “root”.

Lines 16. through 21. Define the labels and buttons necessary for the event generation and callbacks.

Line 22. Starts the main root event loop.

Even though the previous example is a procedural program design, you should be able to recognize from this line-by-line description/explanation that the underlying core widgets from tkinter are OOP classes that we have instanced as objects.

The example used methods on those classes, but the structure of our script file was still functional and declarative in nature.

In-Chapter Exercises

43. Do Examples W19.37 and W19.38 on your Linux system. Then modify them to accomplish interesting calculations, of your own choosing, that take input in a Tk GUI, and show output results in a similar fashion to the examples.

44. What is the difference between a Python method and a Python function?

45. In the grid geometry manager, where does the numbering of cells that widgets can be placed in begin, and how do the numbering indices evolve? For instance, in Example W19.38, what does grid position 0,2 mean?

46. What code would put a “quit” button in the grid cell 0,2 instead of a blank label?

W19.4.4 Multi-threaded Concurrency with Python

The first question you must ask yourself about how and why Linux functions the way it does is: Given the limited resources and nature of modern computer hardware, *how* does Linux maximize performance and efficiency for all users of a system? And how can user programs, such as Python, reflect this technique?

The answer for Linux is:

First, by breaking the hardware, that is, CPU(s), main memory, and peripheral memory, into multiple virtual machinery: in short, *virtualization*.

Then, by executing instructions *concurrently* on this virtualized machinery—this means not in any particular sequence, and perhaps even all at once.

Finally, by making sure that the data generated and stored in the file system is *persistent* over time.

Python, as a Linux tool to create user programs, can achieve concurrency with a mechanism called *threads*, and the concurrent execution of instructions by threads. This is very similar to Linux system programming with threads.

Similarly to the Linux system programming concurrency facility, Python threads give you the ability to run several programs concurrently, in a single process. When you create one or more threads in your Python program, they are executed concurrently, independently of each other, and they can share information among them because they are using the same resources of a single process.

These features make Python threads useful in creating Python applications for such things as network programming and the creation of GUI programs.

Python supports threads on Linux, and any other systems that uses the POSIX threads library (pthreads).

From the Python documentation:

“The Python interpreter is not fully thread-safe. In order to support multi-threaded Python programs, there’s a global lock, called the global interpreter lock or GIL, that must be held by the current thread before it can safely access Python objects. Without the lock, even the simplest operations could cause problems in a multi-threaded program: for example, when two threads simultaneously increment the reference count of the same object, the reference count could end up being incremented only once instead of twice.

Therefore, the rule exists that only the thread that has acquired the GIL may operate on Python objects or call Python/C API functions. In order to emulate concurrency of execution, the interpreter regularly tries to switch threads. The lock is also released around potentially blocking I/O operations like reading or writing a file, so that other Python threads can run in the meantime.”

The takeaway from the above, in terms of Python performance, is that Python threads *cannot* ordinarily take advantage of multiprocessor or multicore processor architectures. Thread switching can only occur between the execution of individual bytecodes in the interpreter. The frequency with which the interpreter checks for thread switching is set by the sys.setcheckinterval() function. By default, the interpreter checks for thread switching after every 100 bytecode instructions.

One of the many ways around this impasse is to use various Python extension modules. The coverage of these is beyond the scope of what we can illustrate here.

W19.4.4.1 Python Thread Examples Using Procedural/Imperative Programming

We show both the \_thread and threading modules in Python 3.5 in the examples we present in this section.

Unless you absolutely need OOP tools and capabilities for Python in general, and specifically in the threading module, the choice is largely a matter of programmer and programming team preference, and programming design goals/methodologies. We also cover some of the functions offered in the threading module in Section W19.4.5.

The basic \_thread module does not necessarily require that you program with OOP. It is very easy to use if you are accustomed to, and desire to, exclusively structure your programs with the procedural/imperative model. The procedural/imperative model is used by programming languages such as C.

The \_thread module provides low-level primitives for working with multiple threads of control (also called *lightweight processes* or *tasks*) that share their global data space. For synchronization, simple locks (also called *mutexes*) are provided within the classes and methods of this module.

W19.4.4.1.1 \_thread Functions Reference

Since the basic \_thread module is a bit simpler than the more advanced threading module covered later in the next subsection, we give examples of that first. This module provides a portable interface to whatever threading system is available in your platform: its interfaces work in the same way on any system with an installed pthreads POSIX threads implementation (including Linux and others). Python scripts that use the Python \_thread module work on all of these platforms without changing their source code.

For your reference, see the Python documentation page we provide at the book GitHub site, that details constants and functions for the \_thread module.

W19.4.4.1.2 Examples

Our first example allows you to experiment with a script that deploys the main \_thread interfaces. As shown in Section W19.4.4.1.1.

The script in Example W19.39 starts successive, one-at-a-time-only threads. Each new thread creation is triggered by a key typed at the controlling console, followed by pressing **<Enter>** , until you type an **x** at that console followed by pressing **<Enter>.** Following the code is a sample command line session that shows the execution of the script file in Python 3.5 on our representative Linux systems.

**Example W19.39**

**import \_thread**

**def child(tid):**

**print ('Started thread', tid)**

**def parent():**

**i = 0**

**while 1:**

**i += 1**

**\_thread.start\_new\_thread(child, (i,))**

**if raw\_input() == 'x': break**

**parent()**

$ **python3 Example\_W19\_39.py**

Started thread 1

**k <Enter>**

Started thread 2

**o <Enter>**

Started thread 3

**r <Enter>**

Started thread 4

**x <Enter>**

$

What exactly is going on in Example W19.39? A single thread is being started, and then it immediately dies, and the program loops indeterminately, allowing you to create successive new threads by pressing a key on the keyboard and then pressing **<Enter>**. Only two thread calls are made in this example: the import of the \_thread module and the call to the method **start\_new\_thread** thatcreates a new thread. This call takes a function (or other “callable”) object as a tuple argument, and starts a new thread to execute a call to the passed function with the passed arguments.

The following is another example that iterates to create new threads that exist simultaneously, in parallel:

**Example W19.40**

**import \_thread, time**

**def counter(myId, count):** # function that will run in each thread

**for i in range(count):**

**time.sleep(1)** # simulate useful code here

**print ('[%s] => %s' % (myId, i))**

**for i in range(5):** # call start\_new\_thread 5 times

\_**thread.start\_new\_thread(counter, (i, 3))** # loop the newest

# thread 3 times

**time.sleep(5)** # prevents exit from parent too early

**print ('Main thread exiting.')** # all threads are destroyed by default

When we execute this script file, we get:

$ **python3 Example14\_40.py**

[4] => 0

[2] => 0

[1] => 0

[3] => 0

[0] => 0

[0] => 1

[4] => 1

[1] => 1

[2] => 1

[3] => 1

[2] => 2

[4] => 2

[3] => 2

[0] => 2

[1] => 2

Main thread exiting.

$

What exactly is happening in Example W19.40? Five threads are being created and run simultaneously. Within each thread, the counter value from 0 to 4 is being printed at standard output. The time.sleep(1) method in the function counter is used to “simulate” code that might be used to do some system programming task(s).

In-Chapter Exercises

47. If each of the threads you start in Example W19.39 were to perform some operations, after what line in the given code would you put the lines of code that performed that work?

48. In the output of Example W19.40 shown, you will notice that the order in which the value of the **myId** variable that is printed is not always the same. Why is this true?

49. If you run Example16\_40.py a few times, is the order of the printed value of **myId** the same on each successive run of the program? Why or why not?

W19.4.4.2 Python Thread Example Using the OOP Model

The threading module constructs higher-level threading interfaces on top of the lower-level thread module. If you want your application to make better use of the computational resources of multi-core machines, you are advised to use the multiprocessing module. However, threading is still an appropriate model if you want to run multiple I/O-bound tasks simultaneously.

For your reference, see the Python documentation page we provide at the book GitHub site, that details constants and functions for the threading module.

W19.4.4.3 OOP GUIs and Producer–Consumer Model Threads

Threads are extremely important and integral to the Tk GUI toolkit, which was illustrated in Section W19.4.3. This also applies in general to GUI libraries such as Qt.

Since many of the functions of a GUI use synchronous I/O, any operation that can block or take a long time to complete must be spawned to run in parallel, so that the central GUI module (the main thread) is always running. Although such children can be run as processes, the efficiency and shared environment model of threads make them ideal for this role. Most GUI toolkits do not allow multiple threads to update the main thread in parallel; updates are best restricted to the main thread.

The two important points to be made about Python threads in a GUI are that the main thread handles all screen graphics updates and that GUI threads must obey the synchronization rules established for general thread concurrency.

All threads in a GUI generally follow what is called the *producer–consumer model*.

This is where one or more objects (the producers) are responsible for placing data into a buffer, and one or more objects (the consumers) are responsible for removing data from that buffer.

The drawbacks to this are as follows:

The producer(s) cannot add more data than the buffer can hold.

The consumers(s) cannot take from an empty buffer,

The actions of all objects *must* be synchronized.

We address more of the issues of the producer–consumer model in Section W19.4.5, and give an example Python program in that section to illustrate the model solution using condition variables.

W19.4.4.4 OOP Threads Example

The following is an example illustrating the basic methodology of using OOP and Python threads. We first describe, in blocks of code, what is happening in the Python code of the example. Then, we present the actual example code in its entirety. Finally, we show sample output when the code is run on the Linux command line.

It would be very instructive for a beginner to compare what the basic methodology and structure of the following OOP example is compared with the previous Python thread code examples presented.

The components of Example W19.41 are shown as blocks of code as follows (the blocks are indicated as comments on the line of code that begins the block).

Block 1. Import the Thread class from the threading module

Why do it this way? The Thread class of the threading module contains many useful methods that allow you to construct and manipulate threads. Using it avoids you having to define your own functions or methods to do the same operations. Not doing it this way would mean you would have to write lower-level system programming functions to accomplish thread creation and synchronization, and then, somehow, stitch that code together with higher-level Python functional programming code.

Block 2. Subclass your own thread, named Threader, by defining it as a child class based on the Thread class, and also define the constructor properties of it as function definitions.

Block 3. Define a .run method in the Threader class. This run method is always executed when we call the start method of any object in our Threaderclass.

The sleep function makes the thread inactive for a definite amount of time. This randomly-timed sleep will ensure that the code will *not* be executed so quickly that we will not be able to notice any changes.

Block 4. The most important block. Create three objects. Apply the .start method to each object, which, in turn, executes the .run method to each object.

You need to call the .join method built into the Thread class, and apply it to each object, or the program will terminate before the threads complete their execution.

**Example W19.41**

**from threading import Thread** #Block 1.

**from random import randint**

**import time**

**class Threader(Thread):**  #Block 2.

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

# The Constructor

**Thread.\_\_init\_\_(self)**

**self.val = val**

**def run(self):** #Block 3.

**for i in range(1, self.val):**

**print('Value %d in thread %s' % (i, self.getName()))**

**# Sleep for random time**

**GoToSleep = randint(1, 5)**

**print('%s sleeping fo %d seconds...' % (self.getName(), GoToSleep))**

**time.sleep(GoToSleep)**

**if \_\_name\_\_ == '\_\_main\_\_':**  #Block 4.

# Declare Threader class

**Threader\_Object1 = Threader(4)**

**Threader\_Object1.setName('Thread 1')**

**Threader\_Object2 = Threader(4)**

**Threader\_Object2.setName('Thread 2')**

**Threader\_Object3 = Threader(4)**

**Threader\_Object3.setName('Thread 3')**

# Run the threads!

**Threader\_Object1.start()**

**Threader\_Object2.start()**

**Threader\_Object3.start()**

# Wait . . .

**Threader\_Object1.join()**

**Threader\_Object2.join()**

**Threader\_Object3.join()**

#Exit . . .

**print('Main Terminating...')**

The output from the program is as follows, when run on our Linux command line:

$ **python Example\_W19\_41.py**

Value 1 in thread Thread 1

Value 1 in thread Thread 2

Thread 1 sleeping fo 1 seconds...

Value 1 in thread Thread 3

Thread 3 sleeping fo 1 seconds...

Thread 2 sleeping fo 3 seconds...

Value 2 in thread Thread 1

Value 2 in thread Thread 3

Thread 3 sleeping fo 5 seconds...

Thread 1 sleeping fo 5 seconds...

Value 2 in thread Thread 2

Thread 2 sleeping fo 3 seconds...

Value 3 in thread Thread 2

Thread 2 sleeping fo 5 seconds...

Value 3 in thread Thread 3

Thread 3 sleeping fo 1 seconds...

Value 3 in thread Thread 1

Thread 1 sleeping fo 5 seconds...

Main Terminating...

$

In-Chapter Exercise

50. If you wanted the threads to do some actual work, in what block of Example W19.41 and exactly where in that block would you put the Python code to accomplish that work?

W19.4.5 Talking Threads: The Producer–Consumer Problem Using the queue Module

In computing, the producer-consumer problem (sometimes called the bounded-buffer problem) is a classic example of the use of various approaches to the synchronization of the execution of multiple threads or processes. We addressed some of the issues involved with the producer–consumer model in Section W19.4.4. In this section, we give more details, and a worked example, to further illustrate this important computer science concept using Python.

The problem concerns two tasks, the producer process and the consumer process, that share a common, fixed-size queue, or buffer space. The producer “produces” a piece of data, puts it into the queue and starts producing more data. Simultaneously, the consumer is “consuming” the data, i.e., removing it from the queue.

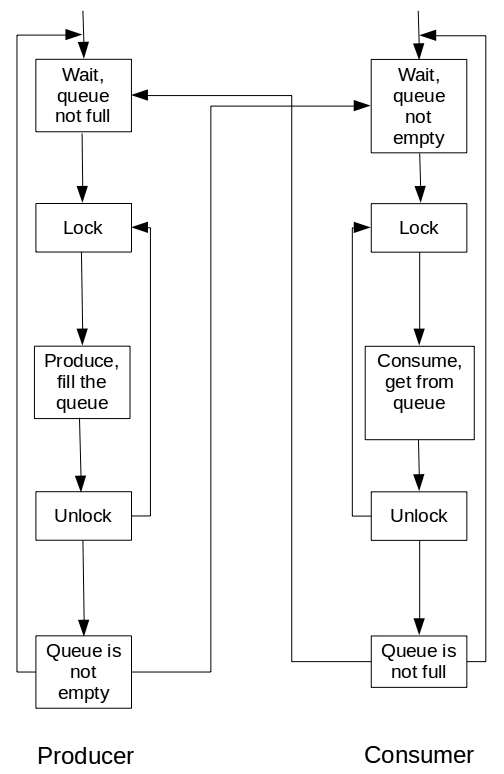
The solution to the problem, as we present it, is to make sure that the producer will not add data into the queue when it is full, and that the consumer will not try to remove data when the queue is empty. The next time the consumer removes an item from the queue, it notifies the producer, which starts to fill the queue again. In the same way, the consumer can become idle if it finds the queue is empty. The next time the producer puts data into the queue, it activates an idle consumer.

There are a variety of approaches to the solution in Python, for example, using locks, semaphores, event synchronization, condition objects or variables, barriers, and the queue module. The Python queue module has three classes that facilitate threading, and are only different in terms of retrieval order off the queue: Queue, LifoQueue, and PriorityQueue. It also incorporates a number of Python methods, which carry within them the “locking/releasing” mechanics that can be applied in order for you to work very efficiently with multi-threaded process applications. We choose to implement our example solution in Section W19.4.5.2 using the queue module, with its Queue class, and its Python methods. Section W19.4.5.1 gives a basic overview of the queue module, its classes and methods. We also give a hyperlink reference to the Python documentation on the queue module, in the Chapter W19 web references at the Github site for this book.

In all of the approaches to a solution in Python, the threads, as executing processes, “talk” to each other while they are actively producing or consuming data.

A summary of the solution is illustrated in Figure W19.3.

Figure W19.3 Producer/Consumer Model Solution



W19.4.5.1 The queue Module, Its Classes and Methods

The queue module in Python 3.X implements three OOP classes of queue, which differ only in the order in which the entries are retrieved. In a First In First Out (FIFO) queue, the first tasks put into the queue are the first taken out. In a Last In First Out queue, the most recently added entry is the first retrieved (similar to a “push-down” stack). In a priority queue, the entries are sorted, and the lowest-valued entry is retrieved first.

The critical feature of the queue module is that locks, and release mechanisms, that temporarily block or free competing threads, are implemented in the OOP “public” methods for the classes, without the programmer having to do it!

For your convenience, we present the queue module definitions of the following Queue classes, and their exception handling methods:

class queue.Queue(maxsize=0)

This is the FIFO constructor for a queue. The argument for maxsize is an integer that defines the largest number of items that can be placed in the queue. Placement of items on the queue will be blocked if the maxsize has been reached; queued items must be consumed to unblock. The queue size default is infinite.

class queue.LifoQueue(maxsize=0)

This is the LIFO constructor for a queue. The argument for maxsize is an integer that defines the largest number of items that can be placed in the queue. Placement of items on the queue will be blocked if the size has been reached; queued items must be consumed to unblock. The queue size default is infinite.

class queue.PriorityQueue(maxsize=0)

This is the constructor for a prioritzed queue. Placement of items on the queue will be blocked if the maxsize has been reached; queued items must be consumed to unblock. The queue size default is infinite.

Note that the lowest valued entries in the queue are retrieved first (the lowest valued entry is the one returned by sorted(list(entries))[0]). Typically entries are tuples of the form- (priority\_number, data).

exception queue.Empty

An exception raised when non-blocking .get() or .get\_nowait() is called on a Queue object which is empty.

exception queue.Full

An exception raised when non-blocking .put() or .put\_nowait() is called on a Queue object which is full.

The three Queue classes provide the public methods described in Table W19.8. Remember that a public method is accessible anywhere outside of the class.

|  |  |
| --- | --- |
| Methods and Arguments | Description |
| Queue.qsize() | Obtain the approximate size of the queue. No assurance that put() or get() will not block. |
| Queue.empty() | Tests the truth value of an empty queue. No assurance that put() or get() will not block. |
| Queue.full() | Tests the truth value of a full queue. No assurance that put() or get() will not block. |
| Queue.put(item, block=True, timeout=None) | Adds item into the queue. If block is true, and timeout is None, blocking occurs. If timeout is a positive number, block occurs at that number of seconds, and the Full exception is raised. If block is false, put item on the queue if a space is available. If no space is available raise a Full exception, and ignore the timeout. |
| Queue.put\_nowait(item) | Same as .put, with arguments item and block=false. |
| Queue.get(block=True, timeout=None) | Delete and return an item from the queue. If block is true, and timeout is None, block until an item is available. If timeout is a positive number, it blocks at most timeout seconds and raises the Empty exception if no item was available within that time. Otherwise (block is false), return an item if one is immediately available, else raise the Empty exception (timeout is ignored in that case). |
| Queue.get\_nowait() | Equal to .get(False) |
| Queue.task\_done() | Signals that a enqueued tasks is finished. |
| Queue.join() | Blocks until all items in the queue have been retrieved and processed. The count of unfinished tasks goes up or down depending on whether an item is added or taken off the queue. |

Table W19.8 queue Class Public Methods

W19.4.5.2 Examples of Using the queue Module Queue Classes

The three Examples we present in this section illustrate queue module classes and methods, and are used to construct a queue in a simple and basic way. Example W19.42 uses the LifoQueue class to implement an LIFO queue, similar to a stack. Example W19.43 uses the PriorityQueue class to implement a queue in which the order of retrieving the elements has to do with some characteristic of each element. Example W19.44 uses the Queue class to implement an FIFO queue.

**Example W19.42**

import queue

q = queue.LifoQueue()

for i in range(7):

q.put(i)

while not q.empty():

print(q.get(), end=' ')

$ **python3** **Example\_W19\_42.py**

6 5 4 3 2 1 0 $

**Example W19.43**

import queue as Q

q = Q.PriorityQueue()

q.put(15)

q.put(18)

q.put(4)

q.put(9)

while not q.empty():

print (q.get(),)

$ **python3 Example\_W19\_43.py**

4

9

15

18

$

**Example W19.44**

import queue

q = queue.Queue()

for i in range(8):

q.put(i)

while not q.empty():

print(q.get(), end=' ')

print()

$ **python3 Example\_W19\_44.py**

0

1

2

3

4

5

6

7

$

W19.4.5.3 An Example of the Queue Class Solution to the Producer-Consumer Model

The following Python code brings together many of the ideas and syntactic structures from this chapter.

This final Example illustrates a Python solution to the producer-consumer problem, using an FIFO with the queue.Queue class, in a multi-threaded program. It uses OOP that deploys the threading module and its methods to start two threads, Producer and Consumer.

An overview of how we use the Queue class to obtain a solution to the producer-consumer problem is as follows:

The Producer places a piece of data on the queue using the .put method. The utility and advantage the queue module confers is most evident here- .put locks the queue, checks to see if the queue is full, and calls an internal .wait() to pause the producer if the queue is full. The Consumer then uses the .get method to acquire the lock before removing data from the queue, and **.get** checks for an empty queue. If the queue is empty, the consumer is put in a wait state.

**.get()** and **.put()** also implement the notification logic to allow “talking” between Producer and Consumer threads.

What exactly is happening, step-by-step, in the code of Example W19.45? A brief explanation of the layout of the program in major blocks, and a description of the operational heart of the code is as follows:

Block 1: This is the initialization part of the program, where the threadingmodule, submodule Thread, and modules queue, time, and random are imported. In this block, we also instance a class of the queue.Queue class , and name it q\_buffer.

Block 2: This defines the function Producer as a class derived from the Thread module, and runs it as self. It specifies the range of integers that will be used as data “numbering”, using the variable numbers. It specifies, in an determinate, or logical repetition loop, how we will add data, as an “actual\_number”, to the queue. The Producer is put to sleep for a random amount of time.

The shared namespace for q\_buffer is established here using the **global** keyword.

Block 3: Very similar to Block 2, this defines the function Consumer as a class derived from the Thread module, and runs it as self. It specifies, in an determinate, or logical repetition loop, how we will retrieve data, in the form of an “actual\_number\_gotten”, from the queue. The Consumer is put to sleep for a random amount of time.

The shared namespace for q\_buffer is used here via the **global** keyword.

Block 4: Producer is started first, and then Consumer is started. The threads continue to be produced and consumed until a <Ctrl><Z> interrupt is sent to the running process from the keyboard standard input.

Trace the steps through the Blocks of code we present next in Example W19.45, in preparation for completion of the In-Chapter Exercises and Problems that follow:

**Example W19.45**

from threading import Thread #Block 1

import time

import random

import queue

q\_buffer = queue.Queue()

class Producer(Thread): #Block 2

def run(self):

numbers = range(5)

global q\_buffer

while True:

actual\_number = random.choice(numbers)

q\_buffer.put(actual\_number)

print ("Produced thread", actual\_number)

time.sleep(random.random())

class Consumer(Thread): #Block 3

def run(self):

global q\_buffer

while True:

actual\_number\_gotten = q\_buffer.get()

q\_buffer.task\_done()

print ("Consumed thread", actual\_number\_gotten)

time.sleep(random.random())

Producer().start() #Block 4

Consumer().start()

Output from the above example is as follows:

$ **python3 Example\_W19\_45.py**

Produced thread 4

Consumed thread 4

Produced thread 0

Produced thread 4

Consumed thread 0

Produced thread 1

Consumed thread 4

Consumed thread 1

Produced thread 4

Consumed thread 4

Produced thread 0

Consumed thread 0

Produced thread 1

Consumed thread 1

Produced thread 3

Consumed thread 3

Produced thread 2

Consumed thread 2

Produced thread 0

Consumed thread 0

Produced thread 2

[1]+ Stopped python3 ex16\_45\_new.py

$ **ps**

PID TTY TIME CMD

30985 pts/0 00:00:00 bash

31016 pts/0 00:00:00 python3

31023 pts/0 00:00:00 ps

$ **kill -9 31016**

$

In-Chapter Exercise (refer to Example W19.45)

51. As the output of running this program one time shows, the production and consumption of data to and from the buffer is sporadic. That is, sometimes data is produced and then immediately consumed, and sometimes data is produced and not immediately consumed. Also, you will notice that if you do several runs of the program, each run may yield different patterns of production and consumption. Why is this so?

Summary

We give a broad introduction to the Python programming language, using Python version 3. We illustrate all of its programming capabilities and syntactic structure, for the beginner, in the context of the three predominant computer programming paradigms. We show the details of doing a fresh install of Version 3.X on our representative Linux systems. We show all of Python’s basic syntax, including numbers and expressions, variables, statements, getting input from the user, functions, OOP in Python, modules, saving and executing Python scripts, string and sequence operations, and error handling. We also give many basic and practical examples, such as another way of writing shell script files, rewriting Bash and tcsh scripts, basic user file maintenance, backing up files, remote copying with rsync, and graphics using tkinter.

QUESTIONS, PROBLEMS, and PROJECTS

1. Type in the following Python code, with the indentation shown, and note what error messages you get. Then, after each error message, type in the proper indentation, until you can execute all eight lines of code.

**x= 23**

**if x==27:**

**print ("no go")**

**print ("why?")**

**elif x ==26:**

**print ("still a no go")**

**else:**

**print ("why?")**

2.

a. Take the code shown in Example W19.5 and convert it to a function script file that can be executed using Way 3 (Import script mode). Name the function script file **testcase.py**. Save it in the current working directory. The function should allow you to enter different values of x as an input argument each time it is invoked, and print out results on screen.

b. How is the function brought into and invoked in Python? Test it with several values of x.

3.

a. Take the code shown in Example W19.6 and convert it to a function script file that can be executed using Way 3 (Import script mode). Name it **nested.py,** and save it in the current working directory. The function should allow you to enter different values of w, y, and z as input arguments each time it is invoked, and print out the results on screen.

b. How is the function brought into and invoked in Python? Test it with several values of w, y, and z.

4. From the following psuedo-coded plan,

for i ← 1 to length(A)

j ← i

while j > 0 and A[j-1] > A[j]

swap A[j] and A[j-1]

j ← j - 1

write and test a script file, using Way 2 (Script mode), that:

a. Allows the user to pass a list individual numbers (real or integer) held in a list A in random order, as arguments to it, and

b. Uses for and while repetition structure(s) to sort the numbers from list A into ascending order, so their values can print out from low to high, left to right on the screen.

5. Create a function script file in Python that uses Way 3 (Import script mode)to execute it. Name it **factIter.py,** and have it calculate the factorial of an integer number using the while indeterminate repetition structure. Test it on the Python command line with several integer values to the script file.

6. Give a definition of list comprehensions, and how the comprehension functional component in Python works on lists, sets, and dictionaries.

7. Similar to Example W19.13, write a Python function script file that uses Way 3 (Import script mode) to execute. It must deploy the list function, and when invoked at the Python command line, allows you to input integer numbers for the indices of the matrix, and output the resulting random number matrix.

8. Convert the Python function script file you created in Problem W19.7 into a new Python function script file that writes the random number matrix to a file named **matrixout.txt**, similarly to what is done in Example W19.12.

9.

a. Convert the script file from Example W19.16 into a new Way 3 (Import script mode) function script file that obtains the filename as an input argument to the function.

b. Rename the data.txt file that Example W19.16’s script file worked on to **data2.txt**. Test your new function script file with the filename **data2.txt**.

c. What are the values of collection[2], collection[2][1], and collection[1][1]? How can you obtain these on the Python command line after your new function script file has run?

10. Using Way 2 (Script mode) to execute it, create a Python dictionary that represents a 3 × 3, two-dimensional matrix of real numbers of your choice. In the same script file, have it print out the elements of the matrix as:

row 1

row 2

row 3

W19.11 After doing Example W19.17, and using the Python standard library reference documentation, find and make a list of some of the basic set operations available (giving the basic syntax for each).

W19.12 Similarly to Examples W19.11 and W19.23, and using Way 3 (Import script mode):

a. Write a Python function script file that takes a filename as an argument, and

b. Uses a robust set of try:…except:…else: error handling statements in it to test the filename, and

c. Opens the file named in a., reads a collection of integers from the file into a list, and

d. Converts all of the list elements to integer numbers, and

e. Produces a new integer list of the squares of all the integers, and finally

f. Prints the list of the squares of all the integers.

Then, test the function for your error handling statements with erroneous file names, or names of files that are not in the current working directory.

W19.13 Modify the code in **arith1.py** from Example W19.24 so that the variable c is declared as a global variable in all four functions (add, subtract, multiply, and divide). For example:

**def add(a,b):**

**global c**

**c = a + b**

#NO RETURN STATEMENT!

Then:

a. Save the file as **arith2.py**.

b. Import **arith2**, and redo In Chapter Exercise W19.32.

c. Explain the results you get when you type >>>**arith2.c** on the Python command line at any particular point after you have imported and invoked the file **arith2** with some numeric arguments.

W19.14 Take the Bourne shell script named **cmdargs\_demo**, shown in Section 12.4 in the printed book, and convert it to Python 3 code that runs using Way 2 (Script mode). Your Python conversion should find the byte sizes of all ordinary (regular) and dot (.) files in the directory specified as an argument to the command, and add them up to print a total. It should skip over subdirectories. It should include the branching structure for error handling statements from the original Bourne shell script.

*Hint*: The following line of Python code adds up, or accumulates, the running total of file byte sizes of ordinary and dot files in the directory specified:

**x = sum(os.path.getsize(f) for f in os.listdir(directory) if os.path.isfile(f))**

W19.15 Following Example W19.34, substitute your own selected directory name on your Linux system as the source for backup, and back up all the files in that selected directory to a mounted USB thumb drive on your system in a rolling scheme of three directories. Then, make changes to the source files and directory on the hard drive, and run the script file again to see how the changes have been synchronized in the files and directory on the USB thumb drive.

Once you have tested this script file, use systemd to run it at a specified increment of time, such as daily at 5 AM, once every 6 hours, etc..

W19.16 You are tasked by your boss to convert a Python 2.7 system administration script file into a Python 3.X script file, to assist in doing sys admin at your company. She provides this Python 2.7 script file as the one seen in Example W19.35. Your Python 3.X script file must execute using Way 2 (Script mode). She also requests that you improve the output of search pattern results; therefore, you are not bound to format your output strictly in the way the Python 2.7 script file does, but your Python 3.X script file must supply at least the same information. She insists that your Python 3.X script file must also do appropriate exception handling; for example, it should not work on hidden files, or on files which are subdirectories of the current working directory.

What pattern-matching characters can you use with your Python 3 script file, such as the wildcard characters (sometimes called globbing patterns), etc.?

W19.17 Beginning with the tkinter script shown in Example W19.38, add Python and tkinter code to allow the user to do not only Fahrenheit-to-Celsius conversions, but also Celsius-to-Fahrenheit conversions. The layout of the widgets in the tkinter GUI for this problem can look similar to Figure W19.4.

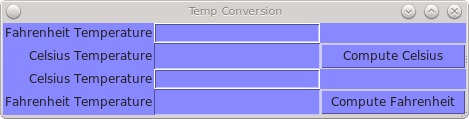


Figure W19.4 Tk Temp Conversion Cell Layout

W19.18 Add a single operating system call to the Example W19.39 code that executes the ps command. Every time you press <Enter> and start a new successive thread, that thread should execute the ps command with output to stdout.

W19.19 Add a single operating system call to the Example W19.40 code that executes the ping –c 5 google.com command. Every thread will then ping google.com “simultaneously.”

W19.20 Write a program using Python threads that pings 10 different hosts of your choice, simultaneously.

W19.21 Starting with the Python code from Example W19.45, rewrite the program so that an indeterminate number of values are produced by the producer and consumed by the consumer. Termination of the program can be achieved with <Ctrl-Z> if necessary.

W19.22 Modify the code of Example W19.45 so that it can achieve the following, in three separate Python scripts:

a) Start 2 producer thread that fill the buffer for 5 iterations each, and one consumer thread that iterates 10 times to consume all data in the buffer produced by the producer threads.

b) Start one producer thread and 3 consumer threads, so that all data produced is consumed.

c) Start three producer threads that fill the buffer for a number of iterations specified from within the script file interactively at run time each, but only have one consumer thread that executes until all produced data in the buffer is consumed.

Table W19.9 Python Syntax and Command Summary

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Interactive Help in Python Shell** | | | | | |
| help() Invoke interactive help | | | | | |
| help(*m*) Display help for module *m* | | | | | |
| help(*f*) Display help for function *f* | | | | | |
| dir(*m*) Display names in module *m* | | | | | |
| **Module Import** | | | | | |
| import *module\_name* | | | | | |
| from *module\_name* import *name,* ... | | | | | |
| from *module\_name* import \* | | | | | |
| **Common Data Types** | | | | | |
| **Type** | **Description** | | | **Literal Ex** | |
| int | 32--‐bit Integer | | | 3, -4 | |
| long | Integer > 32 bits | | | 101L | |
| float | Floating point number | | | 3.0, -6.55 | |
| complex | Complex number | | | 1.2J | |
| bool | Boolean | | | True, False | |
| str | Character sequence | | | 'Python' | |
| tuple | Immutable sequence | | | (2, 4, 7) | |
| list | Mutable sequence | | | [2, x, 3.1] | |
| dict | Mapping | | | { x:2, y5} | |
| **Common Syntax Structures** | | | | | |
| **Assignment Statement**  *var = exp* | | | | | |
| **Console Input/Output**  *var =* input( [*prompt*] )  *var* = raw\_input( [*prompt*] )  print *exp*[,] | | | | | |
| **Selection** | | | | | |
| if (*boolean\_exp*):  *stmt ...*  [elif (*boolean\_exp*):  *stmt ...*]  [else:  *stmt ...*] | | | | | |
| **Repetition** | | | | | |
| while (*boolean\_exp*):  *stmt* ***...*** | | | | | |
| **Iteration** | | | | | |
| for *var* in iterable*able\_object(sequence suite)*:  *stmt*  ***...*** | | | | | |
| **Function Definition** | | | | | |
| def *function\_name*( *parmameters* ):  *stmt* ***...*** | | | | | |
| **Function Call** | | | | | |
| *function\_name*( *arguments* ) | | | | | |
| **Class Definition** | | | | | |
| class *Class\_name* [ (*super\_class*) ]:  [ *class variables* ]  def *method\_name*( self, *parameters* ):  *stmt****...*** | | | | | |
| **Object Instantiation** | | | | | |
| *obj\_ref* = *Class\_name*( *arguments* ) | | | | | |
| **Method Invocation** | | | | | |
| *obj\_ref.method\_name*( *arguments* ) | | | | | |
| **Exception Handling** | | | | | |
| try:  *stmt ...*  except [*exception\_type*] [, *var*]:  *stmt ...* | | | | | |
| **Common Built-in Functions** | | | | | |
| **Function** | | | **Returns** | | |
| abs(*x*) | | | Absolute value of *x* | | |
| dict() | | | Empty dictionary, e.g.: d = dict() | | |
| float(*x*) | | | int or string *x* as float | | |
| id(*obj*) | | | memory addr of *obj* | | |
| int (*x*) | | | float or string *x* as int | | |
| len(*s*) | | | Number of items in sequence *s* | | |
| list() | | | Empty list, eg: m = list() | | |
| max(*s*) | | | Maximum value of items in *s* | | |
| min(*s*) | | | Minimum value of items in *s* | | |
| open(*f*) | | | Open filename *f* for input | | |
| ord(*c*) | | | ASCII code of *c* | | |
| pow(*x,y*) | | | x \*\* y | | |
| range(*x*) | | | A list of x ints 0 to *x* --‐ 1 | | |
| round(*x,n*) | | | float *x* rounded to *n* places | | |
| str(*obj*) | | | str representation of *obj* | | |
| sum(*s*) | | | Sum of numeric sequence *s* | | |
| tuple(*items*) | | | tuple of *items* | | |
| type(*obj*) | | | Data type of *obj* | | |
| **Common Math Module Functions** | | | | | |
| **Function** | | | **Returns (all float)** | | |
| ceil(*x*) | | | Smallest whole nbr >= *x* | | |
| cos(*x*) | | | Cosine of *x* radians | | |
| degrees(*x*) | | | *x* radians in degrees | | |
| radians(*x*) | | | *x* degrees in radians | | |
| exp(*x*) | | | e \*\* *x* | | |
| floor(*x*) | | | Largest whole nbr <= *x* | | |
| hypot(*x, y*) | | | sqrt(*x* \* *x* + *y* \* *y*) | | |
| log(*x* [, *base*]) | | | Log of *x* to *base* or natural log if *base* not given | | |
| pow(*x, y*) | | | x \*\* y | | |
| sin(*x*) | | | Sine of *x* radians | | |
| sqrt(*x*) | | | Positive square root of *x* | | |
| tan(*x*) | | | Tangent of *x* radians | | |
| pi | | | Math constant pi to 15 sig figs | | |
| e | | | Math constant e to 15 sig figs | | |
| **Common String Methods** | | | | |  |
| ***S*.method()** | | **Returns (str unless noted)** | | |  |
| capitalize | | *S* with first char uppercase | | |  |
| center(*w*) | | *S* centered in str *w* chars wide | | |  |
| count(*sub*) | | int nbr of non--‐overlapping occurrences of *sub* in *S* | | |  |
| find(*sub*) | | int index of first occurrence of *sub* in *S* or --‐1 if not found | | |  |
| isdigit() | | bool True if *S* is all digit chars, False otherwise | | |  |
| islower(), isupper() | | bool True if *S* is all lower/upper case chars, False otherwise | | |  |
| join(*seq*) | | All items in *seq* concatenated into a str, delimited by *S* | | |  |
| lower(), upper() | | Lower/upper case copy of *S* | | |  |
| lstrip()  rstrip() | | Copy of *S* with leading/ trailing whitespace removed, or both | | |  |
| split([*sep*]) | | List of tokens in *S*, delimited by *sep*; if *sep* not given, delimiter is any whitespace | | |  |
| **Formatting Numbers as Strings** | | | | |  |
| **Syntax:** format\_spec % numeric\_exp  format\_spec | | | | |  |
| width (optional): align in number of columns | | | | |  |
| specified; negative to left--‐align, precede with | | | | |  |
| 0 to zero--‐fill | | | | |  |
| precision(optional): show specified digits of | | | | |  |
| precision for floats; 6 is default | | | | |  |
| type(required): d (decimal int), f (float), s | | | | |  |
| (string), e (float 􀍴 exponential notation) | | | | |  |
| Examples for x = 123, y = 456.789  % x --‐> … 123  % x --‐> 000123  %8.2f % y --‐> … 456.79  “8.2e” %y--‐> 4.57e+02  “-8s”%y “Hello” ‐> Hello … | | | | |  |
| **Common List Methods** | | | | |  |
| ***L*.method()** | | **Result/Returns** | | |  |
| append(*obj*) | | Append *obj* to end of *L* | | |  |
| count(*obj*) | | Returns int nbr of occurrences of *obj* in *L* | | |  |
| index(*obj*) | | Returns index of first occurrence of *obj* in *L*; raises ValueError if *obj* not in *L* | | |  |
| pop([*index*]) | | Returns item at specified *index* or item at end of *L* if *index* not given*;* raisesIndexErrorif *L* is empty or *index* is out of range | | |  |
| remove(*obj*) | | Removes first occurrence of *objfrom L; raises ValueError if obj is not in L* | | |  |
| reverse() | | Reverses *L* in place | | |  |
| sort() | | Sorts *L* in place | | |  |
| **Common Tuple Methods** | | | | |  |
| ***T*.method()** | | **Returns** | | |  |
| count(*obj*) | | Returns nbr of occurrences of *obj* in *T* | | |  |
| index(*obj*) | | Returns index of first occurrenceof *obj* in *T*; raises ValueError if *obj* is not in *T* | | |  |
| **Common Dictionary Methods** | | | | |  |
| ***D*.method()** | | **Result/Returns** | | |  |
| clear() | | Remove all items from *D* | | |  |
| get(*k* [,*val*]) | | Return *D*[*k*] if *k* in *D*, else *val* | | |  |
| has\_key(*k*) | | Return True if *k* in *D*, else False | | |  |
| items() | | Return list of key-value pairs in *D*; each list item is two‐item tuple | | |  |
| keys() | | Return list of *D’s* keys | | |  |
| pop(*k*, [*val*]) | | Remove key *k*, return mapped value or *val* if *k* not in *D* | | |  |
| values() | | Return list of *D’*s values | | |  |
| **Common File Methods** | | | | |  |
| ***F*.method()** | | **Result/Returns** | | |  |
| read([*n*]) | | Return str of next *n* chars from *F*, or up to EOF if *n* not given | | |  |
| readline([*n*]) | | Return str up to next newline, or at most *n* chars if specified | | |  |
| readlines() | | Return list of all lines in *F*, where each item is a line | | |  |
| write(*s*) | | Write str *s* to *F* | | |  |
| writelines(*L*) | | Write all str in seq *L* to *F* | | |  |
| close() | | Closes the file | | |  |