Python 1

Getting Started

Welcome to the O'Reilly School of Technology's (OST) **Introduction to Python** course! We're happy you've chosen to learn Python programming with us.

Course Objectives

When you complete this course, you will be able to:

- store and manipulate user-input data using Python.
- implement basic Python decisions, iteration, sequence containers, sets, and dicts.
- read and write files using Python.
- define custom functions and call built-in Python functions.
- import modules and namespaces from the Python Standard Library.
- define classes and instantiate objects using Python's Class mechanism.
- handle exceptions and document code.
- build and debug an entire program written in Python.

In this course, you will learn the basics of programming with Python. Using the CodeRunner integrated learning environment, you'll learn about expressions, variables, conditionals, loops, lists, sets, dicts, functions, objects and exceptions.

Besides a browser and internet connection, all software is provided online by the O'Reilly School of Technology.

Learning with O'Reilly School of Technology Courses

As with every O'Reilly School of Technology course, we'll take a *user-active* approach to learning. This means that you (the user) will be active! You'll learn by doing, building live programs, testing them and experimenting with them—hands-on!

To learn a new skill or technology, you have to experiment. The more you experiment, the more you learn. Our system is designed to maximize experimentation and help you *learn to learn* a new skill.

We'll program as much as possible to be sure that the principles sink in and stay with you.

Each time we discuss a new concept, you'll put it into code and see what YOU can do with it. On occasion we'll even give you code that doesn't work, so you can see common mistakes and how to recover from them. Making mistakes is actually another good way to learn.

Above all, we want to help you to *learn to learn*. We give you the tools to take control of your own learning experience.

When you complete an OST course, you know the subject matter, *and* you know how to expand your knowledge, so you can handle changes like software and operating system updates.

Here are some tips for using O'Reilly School of Technology courses effectively:

- **Type the code.** Resist the temptation to cut and paste the example code we give you. Typing the code actually gives you a feel for the programming task. Then play around with the examples to find out what else you can make them do, and to check your understanding. It's highly unlikely you'll break anything by experimentation. If you *do* break something, that's an indication to us that we need to improve our system!

- **Take your time.** Learning takes time. Rushing can have negative effects on your progress. Slow down and let your brain absorb the new information thoroughly. Taking your time helps to maintain a relaxed, positive approach. It also gives you the chance to try new things and learn more than you otherwise would if you blew through all of the coursework too quickly.

- **Experiment.** Wander from the path often and explore the possibilities. We can't anticipate all of your questions and ideas, so it's up to you to experiment and create on your own. Your instructor will help if you go completely off the rails.

- **Accept guidance, but don't depend on it.** Try to solve problems on your own. Going from misunderstanding to understanding is the best way to acquire a new skill. Part of what you're learning is problem solving. Of course, you can always contact your instructor for hints when you need them.

- **Use all available resources!** In real-life problem-solving, you aren't bound by false limitations; in OST courses, you are free to use any resources at your disposal to solve problems you encounter: the Internet, reference books, and online help are all fair game.

- **Have fun!** Relax, keep practicing, and don't be afraid to make mistakes! Your instructor will keep you at it until you've mastered the skill. We want you to get that satisfied, "I'm so cool! I did it!" feeling. And you'll have some projects to show off when you're done.

Lesson Format

We'll try out lots of examples in each lesson. We'll have you write code, look at code, and edit existing code. The code will be presented in boxes that will indicate what needs to be done to the code inside.

Whenever you see white boxes like the one below, you'll *type* the contents into the editor window to try the example yourself. The CODE TO TYPE bar on top of the white box contains directions for you to follow:

CODE TO TYPE:

White boxes like this contain code for you to try out (type into a file to run).

If you have already written some of the code, new code for you to add looks like this.

If we want you to remove existing code, the code to remove ~~will look like this~~.

We may also include instructive comments that you don't need to type.

We may run programs and do some other activities in a terminal session in the operating system or other command-line environment. These will be shown like this:

INTERACTIVE SESSION:

The plain black text that we present in these INTERACTIVE boxes is provided by the system (not for you to type). The commands we want you to type look like this.

Code and information presented in a gray OBSERVE box is for you to *inspect* and *absorb*. This information is often color-coded, and followed by text explaining the code in detail:

OBSERVE:

Gray "Observe" boxes like this contain **information** (usually code specifics) for you to *observe*.

The paragraph(s) that follow may provide addition details on **information** that was highlighted in the Observe box.

We'll also set especially pertinent information apart in "Note" boxes:

> **Note** Notes provide information that is useful, but not absolutely necessary for performing the tasks at hand.

> **Tip** Tips provide information that might help make the tools easier for you to use, such as shortcut keys.

> **WARNING** Warnings provide information that can help prevent program crashes and data loss.

The CodeRunner Screen

This course is presented in CodeRunner, OST's self-contained environment. We'll discuss the details later, but here's a quick overview of the various areas of the screen:

File Browser

File ▾
- Home
  - .php_files
  - apache
  - apache_1.3.41

File Browser area:

Shows all of your files and folders. Navigate like you would in Windows Explorer or the Finder on a Mac.

- perl1
- perl1homework

Welcome | Courses | Messages | Overflow | **System Administration 5**   Help About Sign Out

System Administration 5

Refresh Course
- Lesson 1: Introduction to Linux System Security
  - Objective 1

Course Syllabus area:

To hide or view this panel while working on lessons, click the **<<** or **>>** button.

When you finish a lesson, click the **Quiz** and **Objective** links to do the homework.

- Lesson 5: Remote System Assessment: Investiga
  - Objective 1
  - Quiz 1
- Lesson 6: Local Assessment: Services

Grading Policy

Introduction to Linux System Security

A A HC

Appearance controls:

Use these as needed to change lesson text size or contrast.

Course Obje

After completing this course, you will:

- demonstrate your ability to identify systems security issues and solutions
- identify ways hacke
- find and fix weakne

Lesson Objectives

In this lesson you will:

- read about the O'Re
- navigate in the Code
- log into and out of the Linux Learning Environment

Lesson Content area:

Scroll up and down, and left to right, as needed. You might want to hide the File Browser and Course Syllabus areas while working so you can see more of this content.

Divider bars:

Click and drag to resize any panel.

untitled

1

Code Editor / Terminal Emulation area:

This is where you type the code and commands we give you in the lessons.

Note the icons at the top for various functions. We'll describe these as we use them.

These videos explain how to use CodeRunner:

File Management Demo

Code Editor Demo

Coursework Demo

A Bit of Python History

First, here's a little background information to introduce you to Python. No cringing please—this will be the most textbookish part of the course. We like to provide a little history for our students with a more philosophical and academic bent. Plus you'll have a better understanding of your programming tasks if you have a better idea about what makes Python tick.

The Python language was created by Guido van Rossum in the late 1980s. It was intended to be simple to use and easy to understand. The most interesting new feature of the language was its use of indentation to illustrate structure, similar to the way we use indentation in our everyday prose and written language.

Python was built to have a small "core," to keep it accessible, and a large library to make it versatile. Van Rossum was interested in networking. That interest prompted quick development of a useful set of network libraries for the language; many more libraries have been added since then.

Today, Python is used just about everywhere. Major users include YouTube, Google, Yahoo!, CERN, and NASA, and ITA—the company that produces the route search engine used by Orbitz, CheapTickets, travel agents—as well as many domestic and international airlines.

Python is an *interpreted* language, which means Python code isn't translated into the binary instructions that computers actually run. Instead, *bytecode* is created, and the interpreter uses that bytecode to tell it what to do. Python is also a *dynamic* language. This means that aspects of your program which become fixed early on in some languages, remain available for you to change in Python, even while your program is running.

In this course, we'll be using the latest version of Python (3.1). You may find that other people are using older versions. Fortunately, the differences between versions are minor. We'll go over the changes you'll need to be aware of in order to work with older versions as well.

Unix and CodeRunner

We use our CodeRunner Integrated Development Environment on the Unix operating system for this course. Unix and CodeRunner allow you to write Python programs on our computers so you don't need to worry about having any special software on your own computer. We have an entire OST course devoted to Unix if you'd like a deeper understanding of that operating system. For this course, we'll just cover the concepts you need to know, when you

need to know them.

In the next section, we'll finally enter some Python code and run it!

Programming in Python

A First Program

It is a tradition when learning a new language in computer programming to print the words "hello world" as a first example. Python can print "hello world" in a single line of code, so that doesn't make for a great example here. Instead, we'll look at a slightly more complicated example that not only prints "hello" and "goodbye," but also does a little calculation along the way.

Usually you'll enter a Python program in your favorite text editor and then run it by typing a command in a command *shell* (that's the term for one of those special programs whose job in life is to display a prompt and then let you type in a command that executes when you press Enter). On a Unix or Linux type of system (that includes OS X), the shell is bash, or tcsh, or some other -sh, and the editor is vim or Emacs or one of many other possibilities. On Windows, the shell is the DOS window (what you get by running "cmd") and the editor is Notepad or Wordpad or vim or, again, any of many other possibilities. (You can also run programs on Windows by double-clicking their icons, but then the usual input and output streams are not available.)

In this course we're going to use a completely integrated environment because we want the course to be about learning Python, not about learning a particular editor and operating system. Later in the course, there is a brief lesson on basic Unix commands and editors so that, when you're ready, you can transfer your knowledge of Python to the real world.

Let's get started on your first program! First, we'll create a folder to keep all of our Python stuff organized. In the left panel of your CodeRunner window, right-click **Home**, and select **New folder...** as shown:
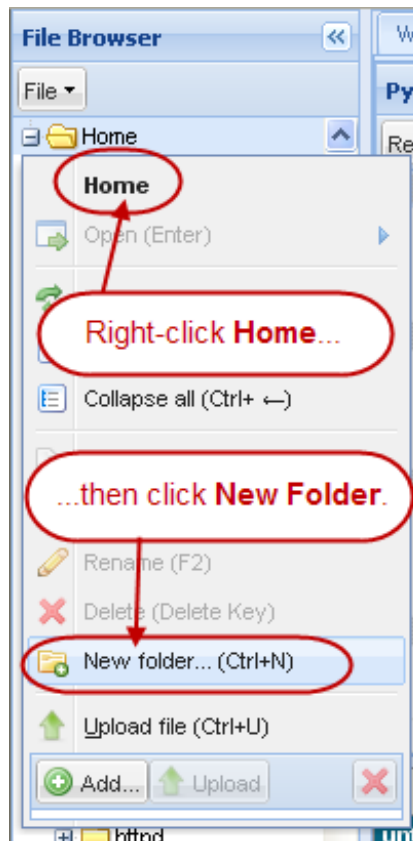
Enter the folder name **python1**:
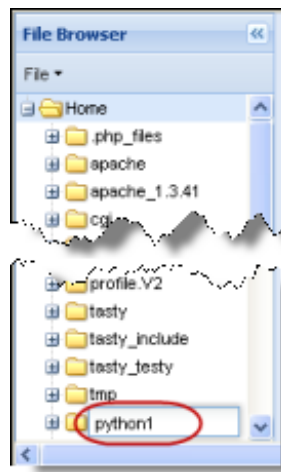
Now you should be able to see your **python1** folder listed in the **File Browser** on the left side of your screen:
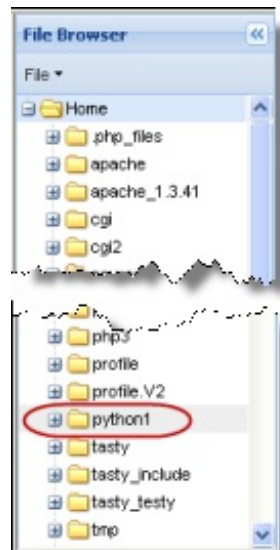
Now, in the editor window area below this lesson text, enter the code below:

CODE TO TYPE:

```
print("Hello World")
print("I have", 3 + 4, "bananas")
print("Goodbye, World")
```

In the editor menu bar, click the **Save** icon: ▯ (we'll show that icon from now on when we want you to save a file). Select your **/python1** folder and enter the file name **hello_world.py**:
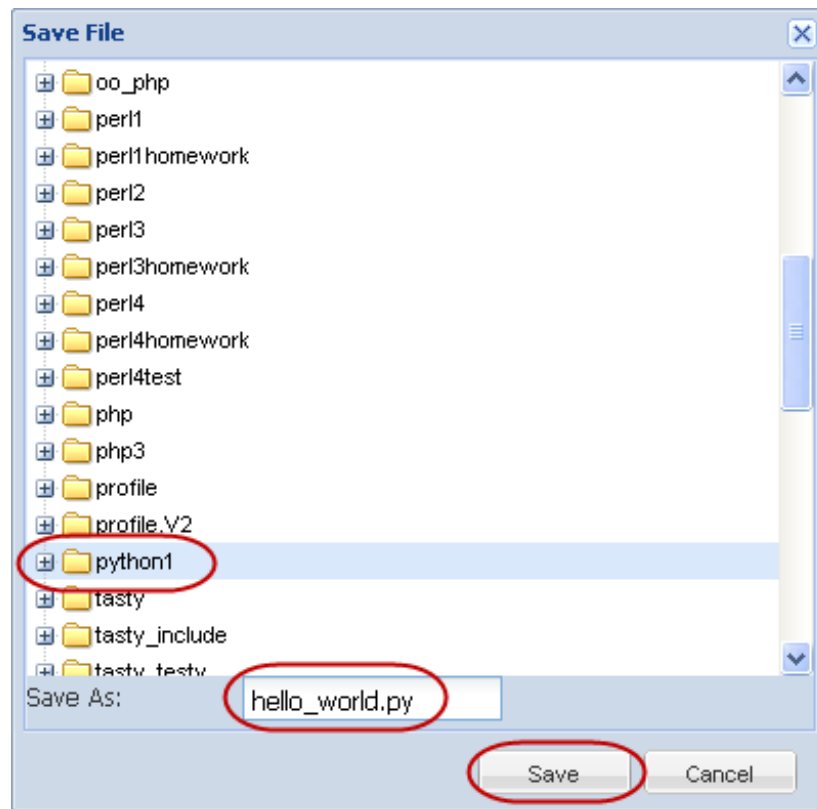
To run the program, open a Unix shell. Click the **New Terminal** icon at the top of the editor window and, when prompted, enter your OST username and password (the password won't appear as you type it):

> INTERACTIVE SESSION:
>
> ```
> cold1 login: username
> Password: yourpassword
> ```

Once logged in, you'll see a **cold1:~$** prompt. Change to your **/python1** folder and then run the program as shown:

> INTERACTIVE SESSION:
>
> ```
> cold1:~$ cd python1
> cold1:~/python1$ python3 ./hello_world.py
> Hello World
> I have 7 bananas
> Goodbye, World
> cold1:~/python1$
> ```

Congratulations! You're officially a Python programmer! Of course this program isn't very complex, but the interpreter has done the calculation you asked it to do. Pat yourself on the back! You're off to a strong start. Experiment with other calculations.

How did this work? Let's look a little closer:

> OBSERVE:
>
> ```
> cold1:~$ cd python1
> cold1:~/python1$ python3 ./hello_world.py
> ```

First we used the Unix changed directory (**cd**) command to change to the **python1** folder where our program is located. Then we ran **python3**, the current version of the Python interpreter, telling it to run the

hello_world.py program located in the current folder (**./**)

The Interactive Interpreter

In Python, you can run the interpreter in interactive mode when you want to try things out and see results printed right away. That instant feedback is really handy when you're learning a new language.

CodeRunner gives you access to interactive Python sessions—click the New Terminal icon, and in the Terminal window, type your OST login and password (again, it doesn't appear while you're typing it), and then type **python3**:

```
INTERACTIVE SESSION:

cold1 login: your OST login
Password: your OST password
cold1:~$ python3
Python 3.1.4 (default, Oct 13 2011, 10:08:29)
[GCC 4.4.5 20110214 (Red Hat 4.4.5-6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The prompt **>>>** indicates that the interpreter is ready for your input.

If you enter one of the lines from the program you just ran, the output will appear. This interactive interpreter window allows you to enter both statements and expressions (we'll cover those in detail later). Statements are executed pretty much as if they were part of a program; the expressions are evaluated and the resulting value is printed (as long as you're in interactive mode).

Type the commands below in the interpreter window *(Remember: when we say TYPE the code, do it. It's good for you!)*. The interpreter prints a result for each expression. (You'll see a different prompt after the fourth line. We'll talk about that in a minute):

```
INTERACTIVE SESSION:

>>> "hello" + " world"
'hello world'
>>> 'hello' + ' world'
'hello world'
>>> """hello""" + ''' world'''
'hello world'
>>> """hello
... world"""
'hello\nworld'
>>>
```

So, what happened here? The first three lines are all examples of *string concatenation*—a second string is appended to the first, giving a longer string as a result. Strings can have either single (') or double (") quotation marks around them, and either one quotation mark or three at the beginning and end of the string. Use exactly the same combination at both ends.

The last expression shows an important difference between the one-quotation mark and the three-quotation mark forms. A string given in one-quotation mark form *must* begin and end on the same line. Three-quotation mark strings can spread across more than one line.

The fourth example actually does extend across two lines, so the interpreter changed its prompt from **>>>** to **…** (an ellipsis) after you entered the first line. The ellipsis lets you know that you've got an incomplete statement or expression, and the interpreter is waiting for you to finish it. When you completed the string with the second line of input, the interpreter then printed the value of the two-line expression, and returned the normal **>>>** prompt. You can see that the line feed between **hello** and **world** is represented by **\n**, which is

known in Python as a string *escape sequence*.

Data in Python

In Python there are various types of data you can manipulate. The simplest are strings. There are also various numeric data types: integers, floats, and complex numbers. Let's see how to write those values in your programs.

String Representations

We've seen that Python has several ways of representing strings. For regular strings, we use either of the one-quotation mark forms. Use three-quotation mark strings if, for example, the value you need to represent contains newlines, or contains quotation marks itself. The interpreter represents certain characters using escape sequences. You can put escape sequences into your strings to insert certain literal or non-printing characters. Here's a list of the most common sequences:

Escape Sequence	Is translated into
\"	Double quote
\'	Single quote (apostrophe)
\\	Backslash
\r	Carriage return
\n	Line feed (newline)
\{newline}	Ignores the newline, allowing you to run a string across multiple program lines
\0*nn*	Character whose value in octal is *nn*
\x*nn*	Character whose value in hexadecimal is *nn*

You can build a really long string using triple-quotation mark strings and escaping the newlines, or by placing several different strings one after the other in your source code. Usually you'll extend those types of statements across multiple lines using parentheses; the interpreter will assume a statement or expression is incomplete if it runs into unmatched parentheses. Continue the interpreter session or start a new one and try these commands:

```
INTERACTIVE SESSION:

>>> """One\
... Two\
... Three"""
'OneTwoThree'
>>> ("One" "Two" "Three")
'OneTwoThree'
>>> 'OneTwoThree'
'OneTwoThree'
```

The interpreter should print the same value back after you enter each of the three strings. The first string you entered spans three lines, but only printed out one.

Numbers in Python

In Python, numbers are represented as you might expect. Integers are strings of digits. The digits can be preceded by a minus sign (-) for negative numbers. There is no limit on integer values in Python, although the larger they get, the longer it takes you to do anything with them!

> **Note** In Python, you *cannot* use commas to separate groups of digits like you sometimes do in text documents.

A floating-point number is made up of an integer followed by a decimal point and a fractional part. You may also use *exponential notation* if you like, by placing the letter **E** after the integer.

Complex numbers generally consist of a real part and an imaginary part that's followed by a **J**; the real part is

separated from the imaginary part by a plus or minus sign. The imaginary number followed by the **J** can comprise a complex number in Python as well. (For you mathematicians wondering why **i** wasn't used, this is standard engineering notation. The rest of us can just carry on.)

Let's try some of this stuff out. If you don't have an interpreter session open, click the **New Terminal** icon and type **python3** again. Then, try these numbers:

```
INTERACTIVE SESSION:

>>> 1
>>> -1000
>>> 12.34
>>> 1.234E2
>>> 1+2j
>>> 1j
```

Huh. it seems the interpreter doesn't always print a value the way you enter it. Floating point numbers aren't always exact, though the interpreter gets as close as possible. Although the errors are relatively small, you want to keep them from accumulating too much in long strings of calculations. (More on that later.) If some of this isn't quite clear to you yet, don't worry. We're just getting started. We'll be talking about it all lots more and you'll have many chances to try things out and ask questions.

Program 2: Printing Simple Python Expressions

You've seen that you can concatenate strings by adding them together. There are many more operations you can perform on your numbers in Python:

Symbol	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
**	Exponentiation

A Few Sample Expressions

Create another program file in the editor window:

```
CODE TO TYPE:
print("""--------------------
Some Calculations
--------------------""")
print(314159E-5)
print(10**6, 1j**2)
print(3 + 2 * 4, 1 / 3)
print("-" * 20)
print((3.14159 * 16) ** 2)
print(3.14159 * 16 ** 2)
print(20 * "-")
print("--------------------\nEnd of Calculations\n--------------------")
```

☐ Save it in your **/python1** folder as **calculations.py** and run it in the Terminal tab:

```
cold1:~/python1$ python3 ./calculations.py
-------------------
Some Calculations
-------------------
3.14159
1000000 (-1+0j)
11 0.333333333333
-------------------
2526.6144583935998
804.24704
-------------------
-------------------
End of Calculations
-------------------
cold1:~/python1$
```

Take a minute to ponder. Think deeply and make sure you understand all of your results before going further. For example, you might wonder, *why does 3 + 2 * 4 give 11, and not 20?* Hmm... something to think about! Feel free to talk with your instructor if you are in any way befuddled.

First Hurdle Cleared

Phew! That was a whole lot of introduction there. Thanks for sticking with me. Keep it up, you're doing great so far. See you at the next lesson!

Entering and Storing Data

Lesson Objectives

When you complete this lesson, you will be able to:

- give data meaningful names.
- extract data from the user.
- store data.

Welcome back. Let's get right to it, shall we? In Python, *explicit* is better than *implicit*. The interpreter won't try to convert the string "3.14159" into a number for you implicitly—if you try to add that string to the integer 1, you'll get an error message:

```
>>> "3.14159" + 1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>> 1 + "3.14159"
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
>>>
```

We'll talk about that more later in the lesson; for now, just let those terms marinate in your mind. In the last lesson, you saw how to represent string and numeric data in Python programs, and use the **print()** function to send expression values to the user. Now we'll look at how to store data and how to extract data *from* the user. Because interactive user input arrives to us in string form, we'll also need to be able to convert strings into other data types.

Binding Values to Names

Most programming languages let you name your data. Giving meaningful names to data makes your code easier to read and helps you to recall its purpose. It also allows you to run the same code with different data values. And most importantly, giving your data meaningful names means you can refer to the same piece of data at different places in your program: you can *store* data and then use it later.

In Python, a value is given a name with the *assignment statement*. In its simplest form, the *assignment statement* consists of a name, an equals sign, and a value. The value can be a single data item or an expression.

Click [icon] to start a new interactive session, and type these Python statements:

```
INTERACTIVE SESSION:

cold1:~$ python3
>>> r = 32
>>> pi = 3.14159
>>> area = pi * r ** 2
>>> print(area)
3216.98816
>>> item = { 'link': "http://holdenweb.com", 'value': 99.99 }
>>> targetURL = item['link']
>>> print(targetURL)
http://holdenweb.com
>>> lst = range(5)
>>> print(lst)
range(0, 5)
>>> r = r + 1
>>> print(r)
33
```

These *assignment* statements all have pretty much the same format: **name = value**. They don't represent mathematical equations, they are *instructions to the computer*. The statements are telling the computer to associate the value on the right side of the equals sign with the name on the left side of it. Once a name is bound to a value, it stays that way unless you change it.

When you read a statement like **r = r + 1**, be sure to read it like a programmer, not a math student! For us, it means to take the value *currently associated* with the name **r**, add one to it, and then associate this *new* value with the name. So if the value of r was **1112** before the statement was executed, it would be **1113** afterward.

Names in Python

Every programming language has rules about which names are acceptable. In Python, the interpreter requires every name to begin with a letter (upper- or lower-case) or the underscore character. The rest of the name can be made up of letters, digits, or underscores. So, **i**, **_private**, **CamelCase**, and **very_long_name_127** are all valid names. But **12_name** isn't valid, because it begins with a digit. **my-name** is also invalid, because it contains a hyphen.

Namespaces and Object Space

Values in Python are stored in memory allocated from a heap (also known as "object space"). The heap is an expandable storage space. *Namespaces* hold names, which refer to values (objects in object space). Memory usage in Python is conveniently automatic. When you bind a name to a value with an assignment statement, that binding takes place in the "current namespace." In a complex Python program, namespaces are created and destroyed continually.
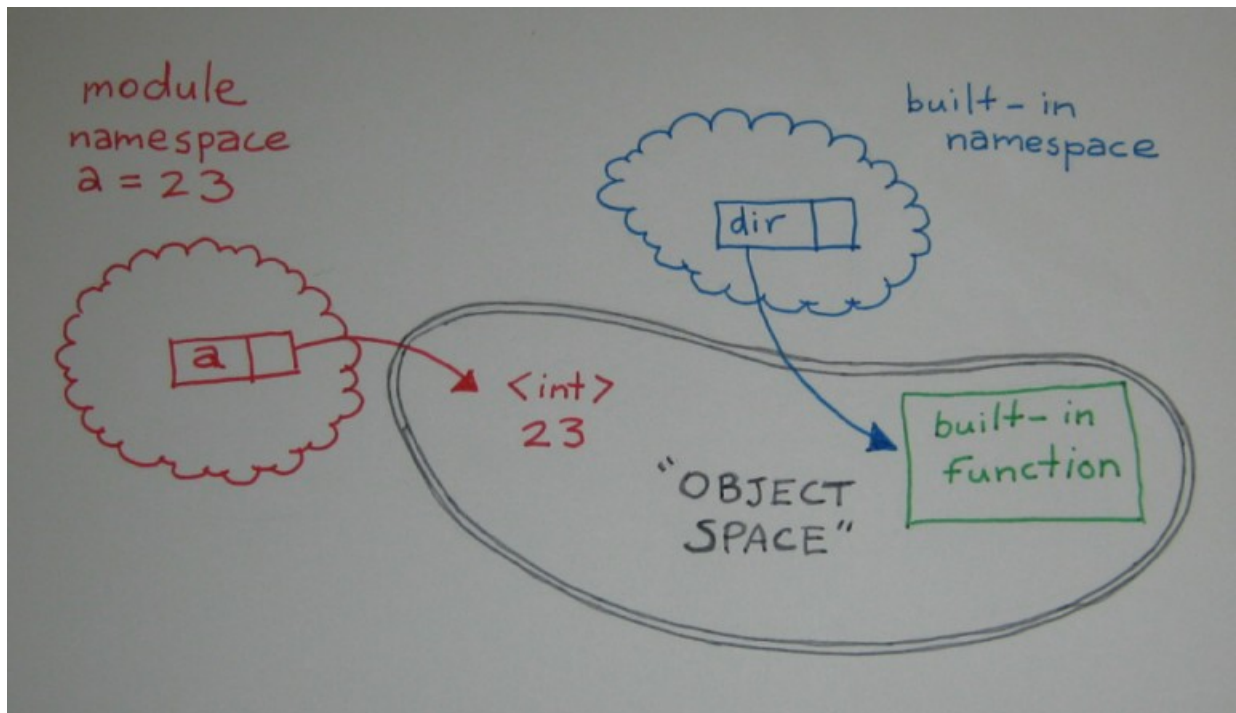
Each Python file you create is a *module*. Each module has its own namespace (often called the *global namespace*). An assignment statement at module level affects the module's global namespace. When the interpreter needs the value associated with a specific name, it looks for the name in a predefined list of places. For module-level code, there are only two namespaces to consider: the module global namespace and the built-in namespace that holds Python's essential functions. You'll learn to write functions and classes later when we create instances of classes. Every time you call a function or create a new class or instance, the Python interpreter creates a new namespace. That namespace becomes unavailable when the related object is destroyed.

Start the interactive interpreter window again and try these commands (remember, the **>>>** is a prompt, not something you have to type):

```
cold1:~$ python3
>>> a = 23
>>> dir()
['__builtins__', '__doc__', '__name__', '__package__', 'a']
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__', '__delatt
r__', '__divmod__',
'__doc__', '__eq__', '__float__', '__floor__', '__floordiv__', '__format__', '__
ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__', '__int__', '__i
nvert__', '__le__',
'__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '_
_or__', '__pos__',
'__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__',
'__rlshift__', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rr
shift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__', '__str__',
'__sub__', '__subclasshook__',
'__truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator',
'imag', 'numerator', 'real']
```

Note You may see different results.

Consider these questions and answers as they relate to the code you just typed:

Q. In which namespace was a value bound to **a**?
A. The module global namespace of the interactive session.

Q. In which namespace did the interpreter locate the **dir()** function?
A. The built-in namespace.

Q. Which namespace does **dir()** report on when called with no argument?
A. The module global namespace.

More Python Syntax Basics

Line Continuations

Under normal circumstances, each line of your Python program is a single statement. The exceptions are when a line is explicitly continued by the addition of a backslash, or when a line ends before an opening paired delimiter (curly bracket, parenthesis, or square bracket) is closed. When you enter statements and expressions in the interactive interpreter, you normally see a **>>>** prompt, indicating that the interpreter is waiting for you to enter a new statement or expression. If you see a **…** prompt instead, it means the interpreter does not regard the current statement or expression as complete. There are different ways to lay out a Python assignment. These next few assignments all bind the value 927 to the name "a." Type these commands in an interactive session:

```
cold1:~$ python3
>>> z = 100
>>> a = (3 + z) * 9
>>> print(a)
927
>>> a = \
...     (3 + z) * 9
>>> print(a)
927
>>> a = (
...     (3 + z)
...     * 9
...     )
>>> print(a)
927
```

Multiple Statements on One Line

Although multiple statements on a single line *can* be separated by semicolons, we don't recommend it. As you'll discover down the road, leading spaces are significant! Python uses leading space to mark blocks of code, so if you start a command line with a space, the command generally will fail with an syntax error.

Let's try a few more examples in the interactive interpreter window (if you closed it, click [icon] to start a new interactive session):

```
cold1:~$ python3
>>> a = 1
>>> z = 2
>>> print(a, z)
1 2
>>> a = 1; z = 2
>>> print(a, z)
1 2
>>> a, z = 1, 2
>>> print(a, z)
1 2
```

In our first example, we have a different single assignment statement. Next, those same two statements appear, separated by a semicolon. Finally, there is an example of what is called an *unpacking assignment*. This has a comma-separated list of names on the left and another list of values on the right. Each value is bound to the corresponding name.

Indentation

In the programs you've written so far, all statements have started in the first column of the line. Statements can be indented when they are the object of one of Python's *compound statements*. A set of statements at the same indentation level (including any code indented within a statement) form a block, also called a *suite*. We'll look more closely at suites when we discuss compound statements in future lessons. For now, just be sure to start your lines without any leading spaces.

Comments

In a Python program text, the "#" character (pound sign, octothorp, hash mark, call it what you will) introduces a comment. The comment runs to the end of the line—it is disregarded by the interpreter. Comments should only occur where whitespace is legal (for readability). Comments help other programmers to make sense of your program, so include them often. As your skill level increases, your comments may be less detailed, but your code should always be easy to read for both intent (the desired result of the code) and implementation (the way the code accomplishes the intent). Use comments as necessary to keep your code readable!

Docstrings

Any Python expression is a valid statement (though statements are *never* expressions). A string on its own, as the first statement of various Python constructs (like module, function, class, and method), is interpreted by many tools as documentation. Using a three-quote string allows you to add lots of documentation to your programs. Use docstrings extensively to document your code. Later examples will show you some practical docstring content. For now, let's try a new program. Type this code in the editor window below:

```
CODE TO TYPE:

#
# This is a program that prints its own docstring
#
"""print_docstring.py prints its own docstring
(the string that's the first executable statement,
which in this case extends to three lines)."""
print(__doc__)
```

☐ Save the program in your **/python1** folder as **print_docstring.py**, and go to an interactive session to change to the /python1 folder and run it:

```
INTERACTIVE SESSION:

cold1:~$ cd python1
cold1:~/python1$ python3 ./print_docstring.py
print_docstring.py prints its own docstring
(the string that's the first executable statement,
which in this case extends to three lines).
cold1:~/python1$
```

> **Note** **__doc__ in the Module Namespace**: In the code above, the interpreter resolves the name **__doc__** by looking in the module namespace. The name is always present, but if the module has no docstring, it is set to the special value **None**.

Now what happens if we remove the docstring—what happens when the print statement runs? Turn the string into an assignment statement by putting "x = " at the beginning of the first line after the comments, as shown:

```
CODE TO TYPE:

#
# This is a program that prints its own docstring
#
x="""print_docstring.py prints its own docstring
(the string that's the first executable statement,
which in this case extends to three lines)."""
print(__doc__)
```

Save and run it. Can you think of any other interesting variations on this program? Go ahead and try a few of your own experiments!

Using String Methods: Case Conversion

In the example below, replace each comment with a Python expression that returns the value described.

Do not use any literal strings—write expressions using methods of s only! For example: **s.capitalize()**.

To see a list of the methods of a string, use **dir("")** in the interactive interpreter.

Type in this code:

```
#
# case_convert.py
#
s = "this Is a Simple string"
slower =  # s converted to lower case <--
supper =  # s converted to UPPER CASE <--
stitle =  # s converted to Title Case <--
print(s, slower, supper, stitle, sep="\n")
```

☐ Save it in your **/python1** folder as **case_convert.py** and run it.

Test your program on various strings by modifying the assignment statement. Play with using the dir("string") function to investigate string methods. For example, try s.capitalize(), s.islower(), s.swapcase()...

It's a little tedious to have to edit the program each time you want to see what happens with a new value, right? Next we will look at a way of allowing the user to provide the strings that our program operates on and avoid all that extra work!

Reading and Converting User Input

The input() Function

To read data entered (interactively) by the user, you use the **input()** function. If you provide **input()** with a string argument, that string (and only that string) will be printed as a prompt, immediately before reading an input string from the user. Once the user types their input (ending it by pressing **Enter**), the function returns the user's input (less the Enter) as a string. Unlike the lines you will read from files, user input has no trailing newline. This is fine, but if you need a number from the user, you must perform some sort of *conversion*. You also need to handle any errors that may arise from your attempts to convert (we'll let that slide for now though and push onward!)

A couple of lines of input are shown in the following screen shot. Notice that the **input()** function always returns a string—even when the user actually types in a number:

```
>>> input("Talk to me: ")
Talk to me: My name is Steve
'My name is Steve'
>>> name = input("Enter name: ")
Enter name: Steve
>>> name*5
'SteveSteveSteveSteveSteve'
>>> x = input("Enter a number: ")
Enter a number: 333
>>> x
'333'
>>>
```

Type Conversions

As we mentioned earlier, in Python, "explicit is better than implicit," so we cannot add a string (even a string whose value is a valid number) to a number. Instead, we have to explicitly convert the string first. The **int()** function takes a single string as an argument, and returns the integer represented by the string (or raises an exception). The **float()** function is similar, but takes any valid string representation of a floating-point number instead (again raising an exception if the string cannot be converted).

You'll need an interactive terminal now. Remember, if you don't have one available, just click the icon. Type in these commands interactively:

```
cold1:~$ python3
>>> n = int(input("Enter a number: "))
Enter a number: 33
>>> x = float(input("Another number: "))
Another number: 45.67
>>> n, x
(33, 45.67)
>>> y = float(input("Final number: "))
Final number: abc.def
Traceback (most recent call last):
  File "<console>", line 1, in <module>
ValueError: could not convert string to float: abc.def>>>
```

Feel free to try other inputs. Observe, too, that the floating-point number system used on computers cannot express 45.67 exactly, though it gets pretty close. This usually only happens with floating-point numbers, not integers. If you haven't programmed before, just remember these "rounding errors" make arithmetic slightly inexact, so be sure they don't make a difference to your results. They can sometimes add up surprisingly quickly. In the last of the three cases above, the user is entering text that cannot be converted into a number. So Python calls the action to a halt with an *exception traceback* that tells you what happened. (Pretty cool, huh?)

Because the observations were made in an interactive interpreter after the traceback, you see another **>>>** prompt. If an unhandled exception occurs when running a program, the program run is terminated. But this isn't always your desired result. Fortunately, there are ways you can handle these exceptions and avoid program termination. For now, let's just type carefully when we need to provide numeric input!

Calculating with Stored Values

Okay! Let's put all this together in a short sample program that asks for the height, width, and depth of a room, and calculates the surface area of the walls. It'll give you an idea of how real code is written.

Type this code in the editor window below:

```python
#
# wall_area.py
#
h = float(input("Room height: "))
w = float(input("Room width : "))
d = float(input("Room depth : "))
area = 2 * (h * (w + d))
print("Area of walls:", area)
```

□ Save it in your **/python1** folder as **wall_area.py**, and run it in an interactive session a few times, using different inputs:

```
cold1:~/python1$ python3 ./wall_area.py
Room height: 12
Room width : 14
Room depth : 32
Area of walls: 1104.0
cold1:~/python1$
```

What happens if you give the program a non-numeric input? (Never fear. We'll show you how to deal with those circumstances later.)

Getting It Done

We're covering a lot of material in these early lessons, and we still have a long way to go. You're doing really well so far—stick with it—see you in the next lesson!

Making Decisions: The if Statement

Lesson Objectives

When you complete this lesson, you will be able to:

- take different actions depending on whether a specific *condition* is true.
- compare and evaluate two values using the comparison operator.
- determine whether one string appears inside another, using the **in** test.
- test for several different conditions.
- use the **elif** keyword instead of **else … if.**.
- use the **or** operator between the conditions.

So far, we've covered basic types of statements in Python—the assignment statement and the statement that calls a function. The function we've used most often is **print()**, which has the handy side effect of printing out the value of one or more expressions.

One of the most common needs in programming is to make a decision and act on it—for example, doing one thing if a customer answers Yes to a question, and another if they answer No. The **if** statement gives us this power, allowing us to take different actions depending on whether a specific *condition* is true.

Conditions in Python

In order to be able to make a decision, you need to evaluate some condition. The conditions we compare most frequently in Python are of two values, using the comparison operator. You can compare for various kinds of equality or inequality:

Operator	True when
a == b	a and b have the same value
a != b	a and b do not have the same value
a < b	a's value is less than b's
a <= b	a's value is less than or equal to b's
a > b	a's value is greater than b's
a >= b	a's value is greater than or equal to b's

Comparing numbers is pretty intuitive, but keep in mind that you can't use these operators to compare complex numbers. The operands are two-dimensional, so they can't be laid out in a straight line; simple comparisons like that aren't valid. Instead, you must compare the absolute values of complex numbers, using the **abs()** function.

Comparing strings is fairly straightforward as long as you can alphabetize a list of items. The characters in strings have a defined order, sometimes called the *collation sequence*. Let's suppose we want to compare strings **a** and **b**. The interpreter looks at the first character of each string. If the first character of **a** occurs earlier in the collation sequence than the first character of **b**, then **a** is less than **b**. If the first character in **a** is greater than the first character in **b**, then **a** is greater than **b**. If this initial attempt to compare the strings is inconclusive, then the next characters in the sequence of the strings are compared until a determination is made.

If the end of one of the strings is reached and additional characters still remain in the other, then the longer of the two strings is greater. If both strings have exactly the same characters in them, they are considered equal. You may see the term "lexical comparison" used to describe this method of comparing strings. Start a new terminal session and verify the following results:

```
cold1:~$ python3
>>> a = 23.0
>>> b = 22
>>> a == b
False
>>> a != b
True
>>> a < b
False
>>> a <= b
False
>>> a > b
True
>>> a >= b
True
>>> p1 = "Python"
>>> p2 = "Perl"
>>> p1 == p2
False
>>> p1 != p2
True
>>> p1 < p2
False
>>> p1 <= p2
False
>>> p1 > p2
True
>>> p1 >= p2
True
>>> "this+" > "this"
True
>>> "that" == "that"
True
>>> "That" == "that"
False
>>> "That".upper() == "thAT".upper()
True
>>>
```

The last tests show that string comparisons are case-sensitive. If you want to avoid case-sensitivity, use the **upper()** or **lower()** method to convert both strings to the same case.

In addition, you can determine whether one string appears inside another, using the **in** test:

```
cold1:~$ python3
>>> x = "nan"
>>> s = "Banana"
>>> x in s
True
```

The result of the expression **x in s** is True when the substring **x** appears somewhere inside the string **s**. You can also test to find out whether a string is a member of a list or a tuple (a tuple is a sequence or ordered list, of finite length); **x in lt** is true if **x** is an element of **lt**, whether **lt** is a list or a tuple.

Also, strings have several methods for you to use to determine whether the string has specific characteristics. The most important ones are shown in this table:

Method Example	True when …
s.startswith(x)	String s starts with the substring x

s.endswith(x)	String s ends with the substring x
s.isalnum()	All characters in s are alphanumeric and there is at least one character
s.isalpha()	All characters in s are alphabetic and there is at least one character
s.isdigit()	All characters in s are digits and there is at least one character
s.islower()	All cased characters in s are lowercase and there is at least one cased character
s.isupper()	All cased characters in s are uppercase and there is at least one cased character

All of these conditions can be tested individually or, as we'll see later, in combination. You can use the **if** statement to choose whether or not to execute one or more statements by testing a condition and executing the statement **if** the condition is true. You can also choose which sets of statements to execute.

Making Decisions: if Statements

Expressions that the Python interpreter will evaluate as True or False (also called *conditions*) can be used to modify the actions of your program, using a basic **if** statement.

The **if** statement begins with the keyword **if**, followed by an expression, and ends with a colon. This line is always followed by an indented suite—one or more statements with an indentation level greater than that of the **if** line. If the condition is true, then the indented suite is executed. All the statements in the suite must be indented to exactly the same level. In the Python world, we use four additional spaces for each new indentation level.

Okay, let's program!

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Detect any mention of Python in the user's input."""

uin = input("Please enter a sentence: ")
if "python" in uin.lower():
    print("You mentioned Python.")
```

> **Note**   By default, the CodeRunner editor and your web browser likely display tab characters as taking up eight spaces. It is proper Python programming practice to indent four spaces. As such, we recommend that you indent code using the space bar rather than the Tab key. Alternatively, if you are familiar with a Linux text editor, you are welcome to use emacs or vim, which are available on the OST server.

Save it in your **/python1** folder as **find_python.py** and run it:

INTERACTIVE SESSION:

```
cold1:~$ cd python1
cold1:~/python1$ ./find_python.py
Please enter a sentence: My python is a happy snake.
You mentioned Python.
cold1:~/python1$ ./find_python.py
Please enter a sentence: There's a pylon in the road.
cold1:~/python1$
```

Run the program several times to verify that when the string "python" is present in your input, the program prints "You mentioned Python." and when "python" is NOT present, it does not. Make sure you test in all circumstances.

> **Note**   Because we added **#!/usr/local/bin/python3** at the beginning of the program, we didn't need to call **python3** specifically in the interactive session. We'll use this from now on in our programs, but it might not work on other systems.

Choosing Between Alternatives: the else Clause

The basic **if** statement allows you to choose whether to execute an indented suite made up of one or more statements. If you want to execute one set of actions if the condition is true and execute another set if it is false, add an **else** clause to the **if** statement. The **else** clause follows the first indented suite, and is followed by the indented suite to execute if the **if** condition is false. When the condition is true, the first suite is executed; when it is false, the second suite is executed. Modify **find_python.py** as shown:

```
#!/usr/local/bin/python3
"""Detect any mention of Python in the user's input."""

uin = input("Please enter a sentence: ")
if "python" in uin.lower():
    print("You mentioned Python.")
else:
    print("Didn't see Python there.")
```

☐ Save and run it. Test your program several times, using both types of input. When your program includes alternative behaviors, it's important to test all the possible paths.

Multiple Choice Decisions

Sometimes a decision isn't as simple as choosing between A or B. You may need to test for several different conditions, then take an action on the first condition that's true. In this case, using **if … else** repeatedly gives rise to a small problem. Chained if ... else statements move code to the right. **else** adds a level of indentation, so if we have a long chain of tests, the code moves over towards the right margin, which can make your code difficult to read. That's nothing we can't handle, though! Take a look:

OBSERVE:

```
if (condition 1):
    suite 1
else:
    if (condition 2):
        suite 2
    else:
        if (condition 3):
            suite 3
        else:
            ...
```

To overcome this, Python has the **elif** keyword, which you can use instead of **else … if**. Because a single **elif** incorporates the functions of both the **else** and the **if** statements, **elif** does not introduce an additional level of indentation:

OBSERVE:

```
if (condition 1):
    suite 1
elif (condition 2):
    suite 2
elif (condition 3):
    suite 3
else:
    ...
```

Both of our examples do the same thing, but the second one is easier to read, and presents the chain of choices much more clearly. The **else** clause at the end is optional; if it's included, then the suite under it will be executed **if** none of the conditions are true. Without an **else** clause, the program won't do anything at all if none of the conditions are true (it will just continue on the line after suite 3).

Now suppose we want to analyze a user's input to detect different programming languages, and respond if we don't find any of our languages mentioned. Modify your program so that it uses **elif** to select among the alternatives. Edit

find_python.py again so it looks like this:

```
#!/usr/local/bin/python3
"""Detect any mention of several languages in the user's input."""

uin = input("Please enter a sentence: ")
if "python" in uin.lower():
    print("You mentioned Python.")
elif "perl" in uin.lower():
    print("Aha, a Perl user!")
elif "ruby" in uin.lower():
    print("So you use Ruby, then?")
else:
    print("Didn't see any languages there.")
```

☐ Save and run the program. Test your results a few times. The first three times, mention one of the target languages; the fourth time don't mention a language at all. Now let's ponder a few questions together: What happens if our input contains two languages? Does the program detect them both? Why or why not?

Combining Conditions: 'and' and 'or'

Sometimes you want to take a particular action only when several conditions are true. You could do this by putting one **if** inside another, or you could use the **and** operator between the conditions. Similarly, if you want your program to execute a particular action when at least one of several conditions is true, you could use the **or** operator between the conditions.

Let's test the **and** and **or** operations interactively:

```
cold1:~$ python3
>>> s = "ABC"
>>> if s.isupper() and s.startswith("A"):
...     print("s is upper case starting with A")
...
s is upper case starting with A
>>>> s = "BBC"
>>> if s.isupper() and s.startswith("A"):
...     print("s is upper case starting with A")
...
(Nothing prints.)
>>> if 1 == 2 or s.endswith("C"):
...     print("Impossible happened or s ends with C")
...
Impossible happened or s ends with C
>>>
```

If two conditions are joined by **and**, the result is true only if *both* conditions are true. If two conditions are joined by **or**, the result is true if *either* condition is true, so even though 1 can never be equal to 2, in the second example, the condition was still true.

Testing for a Range of Values: Chaining Comparisons

Comparison operators have a special feature; they can be "chained." Instead of writing **a < b and b < c**, you can write **a < b < c**. Although there are slight differences between the way the Python interpreter evaluates the two expressions, for now you can regard them as equivalent.

Here are some of the other tests you can create with **if** statements. This program uses the **while** statement. Create this new program as shown:

```
#!/usr/local/bin/python3
target = 63
guess = 0

while guess != target:
    guess = int(input("Guess an integer: "))
    if guess > target:
        print ("Too high...")
    elif guess < target:
        print ("Too low...")
    else:
        print ("Just right!")
```

☐ Save it in your **/python1** folder as **guesser.py** and run it, entering a few guesses. For every guess you make, the program reports whether your guess is too high or too low. With every guess, you close in on the target number. Below is the output for a typical run of the program:

```
cold1:~/python1$ ./guesser.py
Guess an integer: 22
Too low...
Guess an integer: 88
Too high...
Guess an integer: 50
Too low...
Guess an integer: 67
Too high...
Guess an integer: 58
Too low...
Guess an integer: 63
Just right!
```

Wrapping It Up

You're looking good so far. But there's plenty more to learn still!

In the next lesson, we'll look at how we can write more powerful programs using *loops*. See you there!

Iteration: For and While Loops

Lesson Objectives

When you complete this lesson, you will be able to:

- use a **for** statement to loop over the characters in a string you enter, and an **if** statement to determine whether each character is a vowel.
- execute a **break** during a loop to terminate the loop immediately.
-

Glad to see you here again! We've covered a couple of basic "object" types in Python—strings and numbers. We've also encountered one of Python's sequence types—strings. Other Python sequence types may contain more than just characters. But before we try to understand Python's container objects, let's take a look at loops.

Modern computers execute millions or even billions of instructions per second. If we had to write out every instruction, it would take a lifetime to write less than a second's worth of code. Fortunately, we have loops at our disposal. Loops tell the computer to execute the same sequence of actions over and over. Through the use of loops, we can tell the computer to repeat the same operations on different pieces of data, without typing the instructions again each time.

At the end of the last lesson we worked with a program that contained a **while** loop. Using a **while** loop, we can apply the same logic repeatedly—in this case, until a specific condition is true. Similarly, **for** loops allow you to repeat the same actions on each of a number of things. We'll take a closer look at **for** loops first.

A Basic For Loop

Suppose you wanted to count the number of vowels in a string. How would you approach this task? Traditionally, you would set a counter to zero, then loop over the characters in the string, adding one to the counter when a vowel character was found. After all the characters in the string were processed, the counter would contain the total number of vowels in the string.

Let's try an example that uses a **for** statement to loop over the characters in a string you enter, and an **if** statement to determine whether each character is a vowel. Type this code in the editor window:

> CODE TO TYPE:
>
> ```python
> #!/usr/local/bin/python3
> """Counts the vowels in a user input string."""
>
> s = input("Enter any string: ")
> vcount = 0
> for c in s:
>     if c in "aeiouAIEOU":
>         vcount += 1
>     print("C is ", c, "; Vowel count:", vcount)
> ```

☐ Save it in your **/python1** folder as **vowel_counter.py** and run it in the Terminal tab. The "Enter any string:" prompt appears. The program counts the number of vowels in any string you enter and returns a total.

The **for** statement is followed by an indented suite (in this case, a single **if** statement). When the **for** statement executes, the name **s** is bound to a string. For each character in the string, the interpreter executes the indented suite with the name **c**, bound to the current character. So, suppose you entered "the." The first time through the loop, **c** would be "t." The second time, it would be "h," and the third time, it would be "e." Each time around, the **if** statement checks to see whether **c** contains a vowel. If it does, we add 1 to the **vcount** counter; otherwise nothing happens because there is no **else** clause for the **if**. After all characters have been processed, **vcount** contains a count of the vowels within the input string.

We had the program print every time it went through the loop so you could see how it works. Now, let's change our program to print only when it finishes reading the input string. To do that, we simply unindent the print statement so that it falls outside of the loop. We'll also remove the ~~code that prints the value of "c"~~:

```
#!/usr/local/bin/python3
"""Counts the vowels in a user input string."""

s = input("Enter any string: ")
vcount = 0
for c in s:
    if c in "aeiouAIEOU":
        vcount += 1
print("C is ", c, "; Vowel count:", vcount)
```

☐ Save and run it again to see the difference.

Breaking Out of a Loop

The **for** loop is useful for processing each element in a sequence. We'll look at more complex sequences in the next lesson, but for now we'll just use strings.

Suppose you wanted to know where the first space appears in a string. One way to find out would be to loop through the string, counting characters until you found a space. But once you found it, how would you stop counting? You could set a flag to tell your computer to stop counting after you found the space and completed the loop, but then why bother completing the loop? It would be more efficient to stop looking at characters once you found the first space. To do that, we use the **break** statement. If you execute a **break** during a loop, the loop terminates immediately.

Let's write a program that prints out the position where the first space appears in a string. Enter this code:

```
#!/usr/local/bin/python3
"""Program to locate the first space in the input string."""

s = input("Please enter a string: ")
pos = 0
for c in s:
    if c == " ":
        break
    pos += 1
print("First space occurred at position", pos)
```

☐ Save it in your **/python1** folder as **space_finder.py** and run it:

```
cold1:~$ cd python1
cold1:~/python1$ ./space_finder.py
Please enter a string: Space, the final frontier.
First space occurred at position 6
```

The count (**pos**) starts at 0 because that's the first position of a string or any other Python sequence. Each time through the loop, we test to see if the current character (c) is a space (" "). If it is, we hit the **break**, exit the loop, and print the value from **pos**; otherwise we add 1 to **pos** and the loop continues. Be sure you get things in the right order! Incrementing the count before testing and terminating the loop would cause what's known as an "off by 1 error."

But what does the program do if there's no space in the input? I'm glad you asked! It prints out a result as though a space followed the input string, because the loop terminates after it has inspected each character in the string. Check it out in your program by running it with an input containing no spaces.

We need separate logic to verify that there really is a space in the string. Python loops come with such extra logic built in, in the shape of the optional **else** clause. This clause is placed at the same indentation level as the **for** or **while** loop that it matches, and (just as with the **if** statement) is followed by an indented suite of one or more statements. This suite is *only* executed if the loop terminates normally. If the loop ends because a **break** statement is executed, then the interpreter just skips over the **else** suite. In the case of the space detection program, we execute the **break** when we detect a space, so an **else** clause on the **for** loop would only run if there were no spaces in the input.

We need to modify our code a bit more. In the first version, the **print** was located at the end of the loop, where it always runs no matter what the outcome of the testing. Now we want it to be part of the suite *guarded* by the **if** statement, so it only runs when a space is found. Modify your space_finder.py file as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Program to locate the first space in the input string."""

s = input("Please enter a string: ")
pos = 0
for c in s:
    if c == " ":
        print("First space occurred at position", pos)
        break
    pos += 1
else:
    print("No spaces in that string")
print("First space occurred at position", pos)
```

▢ Save and run it with a string that contains a space and then with a string that doesn't. The program runs just fine either way.

As your programs become more complex, you will find that there are sometimes different ways to express the same logic. In those cases, you should "do the simplest thing that works." For example, in the body of the loop, we *could* have put the statement that increments the counter in the suite of an **else** clause. We chose not to use an **else** because if the expression **c == " "** tests as true, the **break** statement will guarantee that **pos** isn't incremented (by immediately exiting from the loop) before the assignment statement is executed.

While Loops

The **for** loop is useful when you want to apply the same logic to each member of a sequence. But sometimes (like in the guessing game at the end of the last lesson) you don't have a finite sequence; you want actions to be repeated until some condition is true.

Suppose we want to split a string into words. Defining words by the spaces between them, we can locate the first space with a **for** loop. Now we can modify the string each time we find a word (by re-binding the name of the string to a new string with the word removed) until there are no more words left. That's the idea behind the next program. Create a new file as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Program to split a sentence into words."""

s = input("Please enter a sentence: ")
while True:
    pos = 0
    for c in s:
        if c == " ":
            print(s[:pos])
            s = s[pos+1:]
            break
        pos += 1
    else:
        print(s)
        break
```

▢ Save it in your **/python1** folder as **sentence_splitter.py** and run it. The **while True** clause in this program sets up a loop that will keep running until logic in the **if/else** suites causes a break. When you see **while True** in a program, either the programmer has included a **break** statement to terminate the loop, or the program is designed for continuous operation—*or* the programmer has made a terrible mistake and the program will never stop running! In this case, it's the former: the **break** to terminate the **while** loop is inside the **for** loop's **else** clause. Run the program and enter a sentence. You should see each word in the sentence on a separate line.

Of course this program isn't perfect—very few programs are! Try entering a sentence where the words are separated by multiple spaces. The program prints empty lines, corresponding to the "empty words" between the spaces. We can fix that, though. One way would be to remove leading spaces before going into the **for** loop each time. Let's modify our sentence_splitter.py to allow multiple spaces between words. Edit the program as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Program to split a sentence into words."""

s = input("Please enter a sentence: ")
while True:
    while s.startswith(" "):
        s = s[1:]
    pos = 0
    for c in s:
        if c == " ":
            print(s[:pos])
            s = s[pos+1:]
            break
        pos += 1
    else:
        print(s)
        break
```

▢ Save and run it.

When you run your updated program, you can enter as many spaces as you like between the words and still get one word per line in the output. Can you figure out how you might use **or** to ignore extra tabs between words? What part of the program would you need to change to treat tabs as completely equivalent to spaces? (Hint: you would have to accept sentences with just tabs between the words.)

Terminating the Current Iteration

The **break** statement can be used to terminate either a **for** or a **while** loop. There is another statement you can use to terminates only the current iteration of a loop, moving on to the next iteration immediately.

In the final example for this lesson, we'll process lines of input from the user. The user will indicate the end of their input by entering a blank line (simply pressing the **Enter** key), but we want them to be able to add comments to their input by entering lines beginning with the **#** character. These lines shouldn't be processed; they are just there to inform the reader. (Python also accepts comments—the **#** character tells the interpreter to ignore everything else up to the end of the line). We aren't especially concerned with the processing that's done on each line, so in this example we'll just use the **len()** function to print.

A comment should be indicated by the first printable character. Create a new file as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Demonstrating the continue statement."""

while True:
    s = input("Enter a line (or Enter to quit): ")
    if not s:
        break
    if s.startswith("#"):
        continue
    print("Length", len(s))
```

▢ Save it in your **/python1** folder as **length_counter.py** and run it. Enter several lines, including at least one comment line that begins with "#." Comment lines are processed differently from regular lines because of the **continue** statement, which immediately causes the program to loop and ask for another input. There are other ways

you could have achieved the same result.

Feel the Power

Once you understand how looping logic works, you're well on the way to comprehending the real power of computers. Looping allows you to tell the computer to repeat the same set of instructions again and again and again...

We must confess, there are easier ways to perform the tasks that you programmed in this lesson, but we want you to understand what's going on behind the scenes first. Start or switch to an interactive session and enter the following expressions:

```
INTERACTIVE SESSION:

>>> "spaces are our friends".find(" ")
6
>>> "What\tis a   word?".split()
['What', 'is', 'a', 'word?']
>>>
```

The **find()** string method locates a given character inside the string (it finds the first occurrence of the string passed as its argument). The **split()** string method, when called without arguments, splits the string up into its constituent words, which are assumed to be separated by one or more white space characters. The strings inside the square brackets constitute a *list*. We'll be looking at those (and their cousins, the tuples) in the next lesson.

Well done. Good for you for sticking with it! Now you have a grasp of a lot of the basics, you'll be able to take on more complex programming challenges. See you at the next lesson!

Sequence Containers: Lists and Tuples

Lesson Objectives

When you complete this lesson, you will be able to:

- write Lists and Tuples.
- access the individual elements in a sequence with *indexing*.
- index and slice elements.
- modify lists.
- use a third component of the slice specification: the *stride*.
- determine whether a sequence contains a specific value, using the **in** keyword.

Welcome back, my favorite student! So far, we've covered basic object types in Python, such as strings and numbers. Now we're ready to look at Python's "container" objects, starting with lists and tuples. Both lists and tuples are *sequence* types. Because strings are sequence types as well, much of what we learn here can be applied to strings as well.

Sequence types have a specific *order*, so it's easy to spot a string's first and last characters. Similarly, lists and tuples present elements in a particular order. Each *element* of a sequence is numbered. The numbering always starts at zero. You refer to an individual element by following the sequence name with a number in square brackets [ ].

Lists and Tuples

Python uses both lists *and* tuples. In general, tuples are used when the position of an element indicates something about its capability, and lists are used to hold elements that will be treated in the same manner. Python doesn't enforce these constraints, though; the only hard and fast rule for using lists and tuples is *don't use tuples if you want to change the sequence*. Tuples are primarily for use when you need a non-modifiable sequence.

Writing Lists and Tuples

Sometimes you'll want to write the contents of a list right inside of your code. To do that, write a comma-separated list of the element values, surrounded by square brackets [ ].

Tuples are usually written as a comma-separated list of values surrounded by parentheses ( ) rather than brackets [ ], but in many cases the parentheses () are optional. (Okay, I will refrain from including illustrations of brackets [ ] and parentheses ( ) now. You get the picture, right?) The interactive interpreter always puts parentheses around a tuple when asked to display its representation, and we recommend you do the same, in order to facilitate readability.

Let's look at some sequences in action. Start an interactive terminal session and type:

```
cold1:~$ python3
>>> lst1 = [1, 3, 5]
>>> lst2 = [2, 4, 6]
>>> tup1 = (9, 7, 5)
>>> tup2 = (8, 6, 4)
>>> dir(lst1)
['__add__', '__class__', '__contains__', ... , 'append', 'count', 'extend',
'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
>>> dir(tup1)
['__add__', '__class__', '__contains__', ... , 'count', 'index']
>>> lst1+lst2
[1, 3, 5, 2, 4, 6]
>>> tup1+tup2
(9, 7, 5, 8, 6, 4)
>>> lst1+tup1
Traceback (most recent call last):
  File "<console>", line 1, in <module>
TypeError: can only concatenate list (not "tuple") to list
>>> clist = [lst1, lst2, tup1, tup2]
>>> print(clist)
[[1, 3, 5], [2, 4, 6], (9, 7, 5), (8, 6, 4)]
```

The results of the calls to the dir() function show that a list has methods that a tuple doesn't have. We abbreviated the output by removing many of the special names beginning with a double underscore. We'll learn more about those methods later, but for now we'll focus on the regular methods.

You can put whatever you like in a list. Usually you'll enter simple values like strings and numbers. The **clist** list in the example above contains two other lists and two tuples.

Accessing Sequence Values

Once you have created a sequence, you can access its individual elements using *indexing*. To index a single element from a sequence, you follow it with a numeric value in square brackets (remember—the first element is numbered zero!). You can also take *slices* from the sequence, creating a new and usually smaller sequence. To slice a sequence, you write two numeric values, separated by a colon (:), inside the square brackets. If you omit the first value, the new sequence starts with the first element of the sliced sequence. If you omit the last value, the new sequence ends with its last element. Continue the interactive terminal session as shown:

```
>>> clist = [1, (2, 3, 4), "a", "Bright", "c"]
>>> clist [1]
(2, 3, 4)
>>> clist[1][1]
3
>>> clist[3][1:3]
'ri'
>>> stuff = clist[2:4]
>>> stuff
['a', 'Bright']
>>> stuff2 = clist[2:5]
>>> stuff2
['a', 'Bright', 'c']
>>> stuff[0]
'a'
>>> "Strings are sequences too"[:7]
'Strings'
```

Indexing and slicing are fundamental operations in Python, so make sure that you understand why each expression evaluates the way it does. Be aware that when you slice a sequence, the second index isn't the index of the last

element in the slice. This is actually very useful. It would be confusing if **clist[2:4]** didn't give you a list 2 elements long, so element 4 isn't included in that slice. So, to get the fourth element in our slice, we referenced the nonexistent element 5. Because strings are also sequences, we can chop strings up without too much difficulty.

Modifying Lists

Although strings and tuples are also sequences, they are *immutable*. Once created, they can't be changed (although you can still index and slice them to extract individual elements or sub-sequences). Lists, however, can be changed. In the same way that you can bind a new value to a name with an assignment, you can also bind a new value to an element of a list. Let's check out one way you can modify a list. Start or continue the interactive terminal session as shown:

```
INTERACTIVE SESSION:

>>> stuff = [1, (2, 3, 4), "a", "Bright", "c"]
>>> stuff[1] = "Not a tuple"
>>> stuff
[1, 'Not a tuple', 'a', 'Bright', 'c']
>>> stuff[0] = 0
>>> stuff[3] = 'b'
>>> stuff
[0, 'Not a tuple', 'a', 'b', 'c']
>>> stuff[2:4]
['a', 'b']
>>> stuff[2:4] = [1, 2, 3]
>>> stuff
[0, 'Not a tuple', 1, 2, 3, 'c']
>>>
```

So far, we've just been just replacing single elements of the list. It's also possible to replace a slice. When you do that, make sure that you also assign another sequence. Any sequence will do—a list, tuple, or string. If you assign a string to a slice, each character in the string becomes a new element of the list. Try experimenting with these possibilities.

Because you can replace any slice of a list, you can delete the slice by assigning an empty sequence to it. But there are less labor-intensive ways to replace a slice of a list. Python's **del** statement was designed especially for deleting things. You can use it on a single element or a slice. If you know that a list contains a certain value, but you don't know the value's index, you can use the list's **remove()** method to delete it from the list. If the same value occurs more than once, only the first occurrence is deleted. Let's give it a try. Type the commands below as shown:

```
INTERACTIVE SESSION:

>>> dlist = ['a', 'b', 'c', '1', '2', 1, 2, 3]
>>> dlist[6]
2
>>> del dlist[6]
>>> dlist
['a', 'b', 'c', '1', '2', 1, 3]
>>> dlist[:3]
['a', 'b', 'c']
>>> del dlist[:3]
>>> dlist
['1', '2', 1, 3]
>>> dlist.remove(1)
>>> dlist
['1', '2', 3]
```

As we saw in an earlier example, we can add elements to a list. Another way to include more elements is to use the list's **append()** method. You call the method and give it a new element to be appended to the list. It's also possible to insert elements at a specific position, and again there are two ways to do this. The simplest way is to use the list's **insert()** method, which you call with a position (index value) and a value to be inserted. Or you could also assign the new value to an empty slice—any slice with the same value for the lower and upper indexes is bound to be empty. Let's experiment with adding new elements to a list. Type the commands below as shown:

INTERACTIVE SESSION:

```
>>> elist = [] # The empty list
>>> elist.append('a')
>>> elist
['a']
>>> elist.append('b')
>>> elist
['a', 'b']
>>> elist.append((1, 2, 3))
>>> elist
['a', 'b', (1, 2, 3)]
>>> len(elist)
3
>>> elist[1:1]
[]
>>> elist[1:1] = ["new second element"]
>>> elist
['a', 'new second element', 'b', (1, 2, 3)]
>>> elist.insert(3, "4th")
>>> elist
['a', 'new second element', 'b', '4th', (1, 2, 3)]
>>> len(elist)
5
```

One of the limitations we run into with slice assignment is that the replacement must be a sequence, so we usually append or insert it. If you have a sequence of elements that you want to insert, keep in mind that slice assignment requires much less code than most other techniques.

Note

If you call a list's **append()** method with a sequence argument (like you did with **elist.append((1, 2, 3))** in the example above), that entire sequence becomes the last element of the list.

Slices with a Stride: Skipping Sequences

A slice specifies a subsequence of a sequence. Now suppose you don't want to include every element, but instead you want to use every second or third element. The easiest way to do this would be to use a third component of the slice specification: the *stride*. The stride specifies how many elements to skip in the slice before extracting the next element. The stride is separated from the first two components with a colon; so your slice specification is like [first:last:stride].

When you specify only two slice components, by default the stride is 1; it takes every element in the slice. A stride of 2 takes every second element, and so on. Stride values can be negative as well as positive. Slicing always works by setting the index to the first slicing component and then increasing the index by the stride value, until the index reaches or goes past the second slicing component. When the stride is negative, the first slicing component must be higher than the second. Type the commands below as shown:

```
>>> alf = "abcdefghijklmnopqrstuvwxyz"
>>> alf[2:13]
'cdefghijklm'
>>> alf[2:13:2]
'cegikm'
>>> alf[2:13:-2]
''
>>> alf[13:2:-2]
'nljhfd'
>>> alf[13:2]
''
>>> alf[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

One way to get the reverse of a sequence is to slice the whole thing by omitting the first and second slice components and then use a stride of -1. So, if you want to replace a list with its reverse, rather than use the list's reverse() method lst.reverse(), you can write lst = lst[::-1]. Python sequences are nothing if not versatile!

Other Functions and Methods to Use with Sequences

Sometimes you'll have a string that you want to break up into a list of words. The **split()** method helps you to do just that. If you call the string without any arguments, it will split the string. If you call it with one argument (for example, a space " "), it will use that argument as a separator, returning a list of the strings that appear between its occurrences.

If you give a second argument, it should be an integer. This informs the interpreter of the maximum number of times to recognize the separator, which limits the number of elements in the returned list.

To get the sum of the numbers in a sequence, pass the sequence to the **sum()** function as an argument. The interpreter will raise an exception if there is a non-numeric element in the sequence. To get the length of any sequence, use the **len()** function.

To find the number of times a particular element appears in a list or tuple, use the **count()** method, with the element value you are seeking as the argument.

Testing for Presence in a Sequence

To determine whether a sequence contains a specific value, use the **in** keyword, which returns either **True** or **False**. Sequences also have an **index()** method that will return the lowest index value at which a given element occurs. You have to be careful using **index()** though, as it will raise an exception if the element isn't present. To avoid the exception, you can use an **if** test to ensure that the value is present, but it's better just to handle the exception, and avoid doing the search twice. We'll cover the **if** statement and exception handling in detail later.

Manipulating Lists and Tuples

Create a new program file as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Simpler program to list the words of a string."""

s = input("Enter your string: ")
words = s.strip().split()
for word in words:
    print(word)
```

☐ Save it in your **/python1** folder as **better_sentence_splitter.py** and run it. This code performs the same tasks as the one we wrote in the last lesson, but it uses features built into the Python language. Now type in a string that contains some white space, press **Enter**, and examine the result. You should see the list of words, printed one per line.

```
s = input("Enter your string: ")
words = s.strip().split()
for word in words:
    print(word)
```

This code applies the **strip()** method to string **s**, which returns a string with no leading or trailing white space. The **split()** method is then applied to the already stripped string, returning a list of the words. The **for** loop iterates over the list, printing each word on a separate line.

Now let's do something a little more complex with lists. We'll take a long piece of text and find out how many lines, words, and characters it contains. To determine the number of characters, use the **len()** method. We count the lines by splitting the text to get a list of them. Finally, we split each line into words and accumulate a total by adding the number of words in each line together.

Create a new file in the editor window as shown:

```
#!/usr/local/bin/python3
"""Count the words, lines and characters in a chunk of text."""

gettysburg = """\
Four score and seven years ago our
fathers brought forth on this continent,
a new nation, conceived in Liberty, and
dedicated to the proposition that
all men are created equal.

Now we are engaged in a great civil war,
testing whether that nation, or
any nation so conceived and so dedicated,
can long endure. We are met on
a great battle-field of that war. We have
come to dedicate a portion of that
field, as a final resting place for those
who here gave their lives that that
nation might live. It is altogether
fitting and proper that we should do this."""

charct = len(gettysburg)

lines = gettysburg.split("\n")
linect = len(lines)

wordct = 0
for line in lines:
    words = line.split()
    wordct += len(words)

print("The text contains", linect, "lines,", wordct, "words, and", charct, "characters.
")
```

☐ Save it in your **/python1** folder as **paragraph_stats.py** and run it. If you typed in exactly the same input text, you'll see **The text contains 16 lines, 102 words, and 557 characters.**

> **Note**   Some operating systems may give different results; for example, Unix records a newline as one character, while Windows records it as two.

Okay, now let's modify our program to keep a count of word lengths, so we know how many one-letter, two-letter, and three-letter words there are, and so on. Modify your paragraph_stats.py file as shown:

```python
#!/usr/local/bin/python3
"""Count the words, lines and characters in a chunk of text."""

gettysburg = """\
Four score and seven years ago our
fathers brought forth on this continent,
a new nation, conceived in Liberty, and
dedicated to the proposition that
all men are created equal.

Now we are engaged in a great civil war,
testing whether that nation, or
any nation so conceived and so dedicated,
can long endure. We are met on
a great battle-field of that war. We have
come to dedicate a portion of that
field, as a final resting place for those
who here gave their lives that that
nation might live. It is altogether
fitting and proper that we should do this."""

lengthct = [0]*20 # a list of 20 zeroes
charct = len(gettysburg)

lines = gettysburg.split("\n")
linect = len(lines)

wordct = 0
for line in lines:
    words = line.split()
    wordct += len(words)
    for word in words:
        lengthct[len(word)] += 1

print("The text contains", linect, "lines,", wordct, "words, and", charct, "characters.
")
for i, ct in enumerate(lengthct):
    if ct:
        print("Length", i, ":", ct)
```
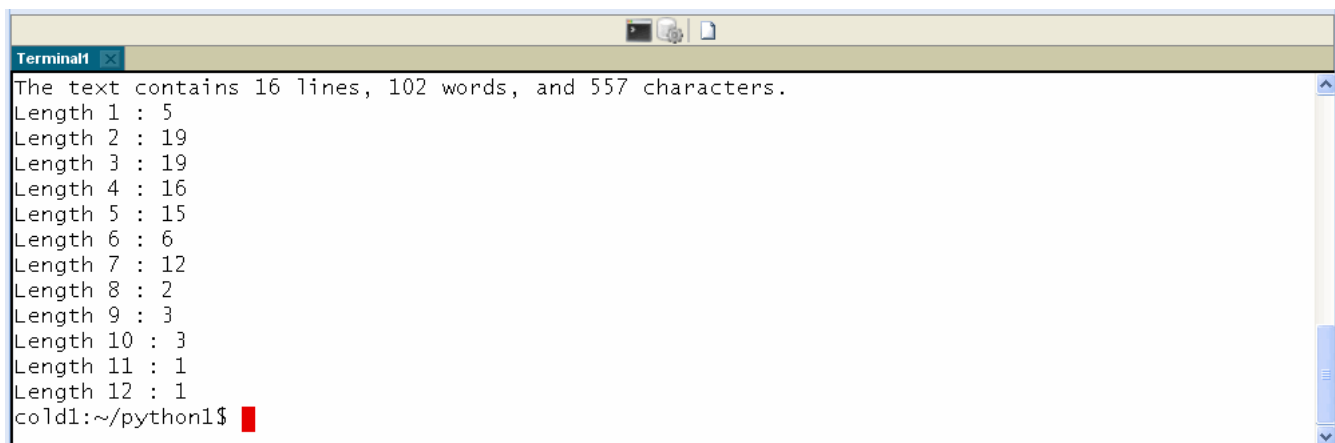
In the new program, we begin by creating a list of counts. The idea is that the count of *n*-letter words will be kept in **lengthct[*n*]**, and we assume that no word will be longer than 19 characters. Sometimes that kind of assumption can be dangerous, but for now, for experimentation's sake, we'll just go with it. In the loop that processes each line, we have added a loop to iterate over the words. The length of each word is used as an index into the **lengthct** list, and that element is incremented by one (they all start at zero). Finally, when the text has been fully processed, there is a bit more code used to output the count of words of each length. The only real wrinkle here is the **if** statement that omits those lengths for which there aren't any words.

☐ Save and run it. Your output will look like this:

```
The text contains 16 lines, 102 words, and 557 characters.
Length 1 : 5
Length 2 : 19
Length 3 : 19
Length 4 : 16
Length 5 : 15
Length 6 : 6
Length 7 : 12
Length 8 : 2
Length 9 : 3
Length 10 : 3
Length 11 : 1
Length 12 : 1
cold1:~/python1$
```

So far, so good. Go on and experiment some more. Modify the text so it contains a word of twenty characters or more (like "deinstitutionalizing"). What happens when you run the program? How could you make the program work again? Can you think of a way you might modify the program to keep a count of the individual words, so you could see how many times each word was used? Using only sequences, this is pretty difficult, but not impossible.

It Slices, It Dices...

You've learned quite a bit about Python's sequence types and just how useful they can be. Next, we'll check out Python's *mapping* types.

Phew. This isn't easy, but you're doing really well. Keep it up, and I'll see you in the next lesson!

Sets and Dicts

Lesson Objectives

When you complete this lesson, you will be able to:

- create sets.
- use sets and dicts.

Good to have you back! We'll go over some background information first and then get to work on some examples. Lists and tuples are versatile data structures, but they have one fundamental property that you just can't get around: elements are retrieved by number. That's fine when you're working with elements that are numbered, but on occasion you'll want to be less specific. That's when *sets* and *dicts* come in handy.

Sets are similar to lists. You can use the **in** keyword to find out whether a particular value appears as an element within a list or a set. The interpreter finds members within lists by checking each element of the list, one after the other, until it finds the value it's looking for, or gets to the end of the list. The interpreter finds elements in sets using a much faster method "under the hood" than the linear scan used for lists.

Using a list is fine when your program contains just a few elements, but the number may grow over time, particularly when the data is being stored in a file or a database. As the number of elements in your program grows, program performance becomes increasingly slow. That could cause problems. In those cases, it's better to use a set in the first place.

The same value can appear multiple times in a list, but in a set, a value can appear only once. When you "add" an element to a set that contains that particular element already, the set remains the same. Because of this feature, you can't predict the order in which the set elements will occur if you loop over them with a **for** loop. When you add an element, the order may change completely. In other words, although sets are *collections* or *containers*, sets aren't *sequences*. There is no concept of "position" for set elements. Conversely, in a list, you can determine the position of a given element, using the **index()** method.

Dicts are similar to lists as well. A dict stores values that can be retrieved by indexing, but the index values in a dict don't need to be numerical. All of this will make more sense after we work through a few examples.

Creating Sets

You write a set as a comma-separated list of elements inside braces { }—for example, you'd type the first three natural numbers as **{1, 2, 3}**. You can also use Python's built-in **set()** function. This is usually called with a single sequence argument, and creates a set that contains all the sequence's elements.

Python includes two separate data types for handling sets. As with lists and tuples, you build regular sets, then you can add or remove elements. You can also build frozen sets, which stay the same once you have created them and raise an exception at any attempt to change them. A set is an unordered collection of items with no duplicate elements; lists are ordered and sets are not. Set objects also support various operations like union, intersection, and difference. If you're not familiar with all this stuff yet, don't panic! We'll go over all of it in detail here and in later lessons.

Working with Sets

Okay, now it's time to get friendly with set operations! Start an interactive terminal session and enter the following commands:

```
cold1:~$ python3
>>> {1, 2, 3, 1, 2, 3, 1, 2, 3, 1}
{1, 2, 3}
>>> vowels1 = {"a", "e", "i", "o", "u"}
>>> vowels2 = set("aieou")
>>> vowels1 == vowels2
True
>>> languages = {"perl", "python", "c++", "ruby"}
>>> languages.add("php")
>>> languages
{'python', 'php', 'ruby', 'c++', 'perl'}
>>> "perl" in languages
True
>>> "java" in languages
False
>>> {'python', 'ruby'} < languages
True
>>> set("the quick brown fox") & vowels1
{'i', 'u', 'e', 'o'}
>>> vowels1 - set("the quick brown fox")
{'a'}
>>> set("the quick brown fox") - vowels1
{' ', 'c', 'b', 'f', 'h', 'k', 'n', 'q', 'r', 't', 'w', 'x'}
>>>
```

Note   In the result from **set("the quick brown fox") & vowels**, the duplicate elements are eliminated.

The examples above used integers, characters, and strings, but most Python objects can be elements of a set. You can compute the *intersection* of two sets using the **&** operator, and the *difference* between two sets with the **-** operator. There are a number of other operations you can perform on sets as well. Many, but not all, of the operations can be performed using either operators or a method call on one of the sets.

Assume that **s** and **t** are sets in the following table:

Operation	Method Call	Returns
x in s	-	True if **x** is an element of set **s**.
s <= t	**s.issubset(t)**	True if every element of **s** is also an element of **t**.
s < t	-	True if every element of **s** is also an element of **t** but there is also an element of **t** that is not in **s**.
s >= t	**s.issuperset(t)**	True if every element of **t** is also an element of **s**.
s > t	-	True if every element of **t** is also an element of **s** but there is also an element of **s** that is not in **t**.
-	**s.isdisjoint(t)**	True if **s** and **t** have no element in common.
s | t	**s.union(t)**	The set containing all elements of **s** and all elements of **t**.
s & t	**s.intersection(t)**	The set containing all elements that are in both **s** and **t**.
s - t	**s.difference(t)**	The set containing all elements that are in **s** but not in **t**.
s ^ t	**s.symmetric_difference(t)**	The set containing all elements that are in **s** or **t** but not in both.
s |= t	**s.update(t)**	**None**, but adds all elements of **t** to **s**.
s &= t	**s.intersection_update(t)**	**None**, but leaves **s** containing only elements that originally belonged to both **t** and **s**.
s -= t	**s.difference_update(t)**	**None**, but removes any elements of **t** from **s**.
s ^= t	**s.symmetric_difference_update(t)**	**None**, but leaves **s** containing all elements that belong to **t** or **s** but not both.

Let's use a set to keep track of how many different words there are in a given piece of text. Create a new file as shown:

```python
#!/usr/local/bin/python3
"""Count the number of different words in a text."""

text = """\
Baa, baa, black sheep,
Have you any wool?
Yes sir, yes sir,
Three bags full;
One for the master,
And one for the dame,
And one for the little boy
Who lives down the lane."""

for punc in ",?;.":
    text = text.replace(punc, "")
print(text)
words = set(text.lower().split())
print("There are", len(words), "distinct words in the text.")
```

☐ Save it in your **/python1** folder as **word_counter.py** and run it.

This is a classic problem we can run into when working with text in our programs. Python lets you solve it in a unique way. First, it uses a **for** loop to remove all the punctuation (punc) characters (,?;.) from the string, replacing each one with an empty string (""). Next, it prints the text so you can confirm that the punctuation has been removed. Finally, it converts the text to lower-case, splits the text at each run of white space, and creates a set from the resulting list.

Python removes the punctuation to ensure that only words are present in the text. "Baa" is not the same as "Baa," (with a comma), so the punctuation must be removed. The text is converted to lower case before splitting so that, for example, "One" and "one" will not be treated as unique words. A set cannot contain duplicate entries. The number of elements in the set (given by the **len()** function) is comprised of the number of *different* words in the text.

To see another application of sets, let's write a program that compares two inputs and prints out the words they have in common and various other pieces of information. Type the code below as shown:

```python
#!/usr/local/bin/python3
"""Find matching words in two input lines."""

words1 = set(input("Sentence 1: ").lower().split())
words2 = set(input("Sentence 2: ").lower().split())
print("Words in both strings", words1 & words2)
print("Unique to sentence 1:", words1 - words2)
print("Unique to sentence 2:", words2 - words1)
```

☐ Save it in your **/python1** folder as **word_matcher.py** and run it, and then enter two different sentences with some words in common, as shown:

```
cold1:~$ cd python1
cold1:~/python1$ ./word_matcher.py
Sentence 1: Four score and seven years ago
Sentence 2: Four and twenty blackbirds were baked in a pie
Words in both strings {'and', 'four'}
Unique to sentence 1: {'ago', 'seven', 'score', 'years'}
Unique to sentence 2: {'a', 'blackbirds', 'in', 'pie', 'twenty', 'were', 'baked'}
cold1:~/python1$
```

The program prints the sets of words, telling you which are common to both sentences and which are unique to each sentence. Because the sets are not sorted, the program prints them in unpredictable order. To overcome this issue, modify the program to make use of Python's **sorted()** function. Edit your code as shown in blue:

```python
#!/usr/local/bin/python3
"""Find matching words in two input lines."""

words1 = set(input("Sentence 1: ").lower().split())
words2 = set(input("Sentence 2: ").lower().split())
print("Words in both strings", sorted(words1 & words2))
print("Unique to sentence 1:", sorted(words1 - words2))
print("Unique to sentence 2:", sorted(words2 - words1))
```

☐ Save and run it, and enter the same two sentences. The output of the first version of this program printed sets as its results, but this modified version prints lists. When applied to a set, the **sorted()** function sorts the elements of the set into a list. This displays our results in a predictable (alphabetical) order.

Working with Dicts

The dict is a useful structure for storing values against arbitrary keys. Let's take a look. Start an interactive interpreter session and type the commands shown below:

INTERACTIVE SESSION:

```python
>>> d = {'Steve': 'Python', 'Peter': 'Perl', 'Rob': 'Ruby'}
>>> d['Rob']
'Ruby'
>>> d['Peter'] = "C#"
>>> d
{'Steve': 'Python', 'Peter': 'C#', 'Rob': 'Ruby'}
>>> d['Peter']
'C#'
>>> del d['Peter']
>>> d
{'Steve': 'Python', 'Rob': 'Ruby'}
>>> d['Guido'] = 'Python'
>>> d
{'Steve': 'Python', 'Rob': 'Ruby', 'Guido': 'Python'}
>>> d.keys()
dict_keys(['Steve', 'Guido', 'Rob'])
>>> for k in d.keys():
...    print(k)
...
Steve
Rob
Guido
>>> for k in d.items():
...    print(k)
...
('Steve', 'Python')
('Rob', 'Ruby')
('Guido', 'Python')
>>> d[(1, 2)] = "Tuple"
>>> d[1] = "Integer"
>>> d
{(1, 2): 'Tuple', 1: 'Integer', 'Rob': 'Ruby', 'Steve': 'Python', 'Guido': 'Python'}
>>> d[1]
'Integer'
>>> d[1.0] = "Hello there"
>>> d[1+0j]
'Hello there'
```

Here you can see some of the most important aspects of dict behavior. Dict literals use braces { } like sets do, but each element is represented by a key, followed by a colon and the value associated with that key. In the example above, you can see strings, numbers and tuples being used as keys. There are some types of object you can't use as keys, but let's not worry about that just yet. We've got enough to wrap our brains around for now!

In addition to creating dicts with a literal representation, you can also add new key-value pairs, and replace the value associated with an existing key, using assignment statements. If you assign to an existing key in the dict, then the assigned value replaces the previously associated value. If no value is associated with the key (in other words, if the key does not currently exist in the dict), then the key is added and the assigned value is associated with the key.

Numeric keys receive slightly different treatment. You might expect that **d[1]**, **d[1.0]**, and **d[1+0j]** would refer to different values in the dict, but those three keys are all numerically equal, and so assigning to **d[1.0]** overwrites the value assigned to **d[1]**, and the same value can be retrieved by referencing **d[1+0j]**.

You can also see in our example that dicts have a **keys()** method that returns the keys of the dict. This is known in Python as an *iterator*. We'll look at iterators in some detail in a later course, but for the moment all you need to know is that you can iterate over it, and each time around the loop, you get another key from the dict. The same is true of the dict's **items()** method, only this iterator yields key-value pairs rather than the keys from the dict.

The dict is a flexible object type. You can perform the following operations on a dict **d**:

Expression	Description
d[k]	Returns the item from **d** associated with key **k**, raising a KeyError exception if k is not present.
len(d)	Returns the number of items in the dict.
del d[k]	Removes **d[k]** from **d**, raising a KeyError exception if k is not present.
k in d	Returns True if **d** has a key **k**; otherwise returns False.
k not in d	Returns True if **d** does not have a key **k**; otherwise returns False.
d.get(k, default)	Returns the value of **d[k]** if that key exists; otherwise returns **default** (if the default value is not given, returns None rather than raising a KeyError exception).
d.update(other)	Updates the dict, overwriting any existing keys that appear in **other**, which can either be another dict or a sequence of key-value pairs.

Remember, you learn more by experimenting. Play around with a dict or two in an interactive console until you are comfortable with the way they work.

Applying Dicts: Counting Words

Now that you know how dicts work, let's apply the concept to a classic text processing problem: counting the occurrences of words within a text. In a previous exercise, we put the words from a piece of text into a set, but there was no way to associate a count with each word—all we can do with a set is detect whether an item is present.

Because the dict is able to associate a value with the key, we can use each word as a key in the dict and have the associated value be the number of times the word appears in the text. Open your **word_counter.py** program, click the Save As (🖫) icon, and save it in the **/python1** folder as a new file named **word_frequency.py**. Then, edit it as shown:

```
#!/usr/local/bin/python3
"""Count the frequency of each word in a text."""

text = """\
Baa, baa, black sheep,
Have you any wool?
Yes sir, yes sir,
Three bags full;
One for the master,
And one for the dame,
And one for the little boy
Who lives down the lane."""

for punc in ",?;.":
    text = text.replace(punc, "")
freq = {}
for word in text.lower().split():
    if word in freq:
        freq[word] += 1
    else:
        freq[word] = 1
for word in sorted(freq.keys()):
    print(word, freq[word])
```

☐ Save and run it. You'll see output showing the number of times each word appears in the text. Word splitting works in the same way as before, but now, each time a word is examined, the program checks to find out whether the word has appeared before. If it has not, then a new entry is made in the dict with a value of one. If it has (if it is already found in the **freq** dict), the current count is incremented.

A slight modification to the program allows us to dispense with the **if** statement. Edit **word_frequency.py** as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Count the frequency of each word in a text."""

text = """\
Baa, baa, black sheep,
Have you any wool?
Yes sir, yes sir,
Three bags full;
One for the master,
And one for the dame,
And one for the little boy
Who lives down the lane."""

for punc in ",?;.":
    text = text.replace(punc, "")
freq = {}
for word in text.lower().split():
    freq[word] = freq.get(word, 0)+1
for word in sorted(freq.keys()):
    print(word, freq[word])
```

☐ Save and run it. You see the same results as before. This version of the program uses the same statement to update the count, even if the word has been seen before. It uses the **get()** method with a default value of zero to retrieve the existing count, so if the word hasn't been seen before, the assignment inserts a value of one against the new key.

A More Complex Application: Word Pair Frequencies

In the final example of this lesson, we'll do a slightly more complex counting task. For each word in the input, we will keep a count of the number of times it was followed by each of the other words that immediately follow it in the text. This involves keeping a dict for each word. The keys of this second dict will be the words that immediately follow the original word. The values will be the number of times that particular word followed the original word.

With your word_frequency.py program open in the editor window, click the Save As ( ) icon and save the file in your **/python1** folder as **pair_frequency.py**. Edit the new program as shown:

```
CODE TO TYPE:

#!/usr/local/bin/python3
"""Count the frequency of each word in a text."""

text = """\
Baa, baa, black sheep,
Have you any wool?
Yes sir, yes sir,
Three bags full;
One for the master,
And one for the dame,
And one for the little boy
Who lives down the lane."""

for punc in ",?;.":
    text = text.replace(punc, "")
words = {}
textwords = text.lower().split()
firstword = textwords[0]
for nextword in textwords[1:]:
    if firstword not in words:
        words[firstword] = {}
    words[firstword][nextword] = words[firstword].get(nextword, 0)+1
    firstword = nextword
for word in sorted(words.keys()):
    d = words[word]
    for word2 in sorted(d.keys()):
        print(word, ":", word2, d[word2])
```

☐ Save and run it.

Since we have to process words in pairs, we set **firstword** to be the first word in the text. The loop then loops over the *rest* of the text (**textwords[1:]**), assigning the word to **nextword**. At the end of each pass through the loop, **nextword** is assigned to **firstword**, so that at the start of each iteration we have a consecutive pair of words in **firstword** and **nextword**.

The program makes sure that there is an entry for the first word in the **words** dict: each entry starts out as an empty dict, which will be used to store the number of occurrences of individual *following* words. Then it uses the same technique that the second version of word_frequency.py did to update the count of the *following* word. The printing of the output has become a little more complex, because each word requires you to print out each *following* word. So in the output phase, we have *nested* loops: one loop inside another.

Nice Work!

You've just added sets and dicts to your programming tool kit—no easy feat! Excellent! In the next lesson, we'll focus on output, and ways to control the format of the output produced by your programs.

I like what I'm seeing so far! Keep it up and see you in the next lesson...

String Formatting

Lesson Objectives

When you complete this lesson, you will be able to:

- use the format() Method.
- create a string to be used as a format, then call the string's **format()** method to produce a formatted value or set of values.
- use the *field name* to tell the formatting engine which value is to be formatted.
- specify a field's type, specifications, and alignment.
- create a simple listing program.

Up until now your programs created output using the **print()** function, which simply sends the output to the screen without embellishment. But sometimes you'll need your output to be formatted in a particular way. Python has some really convenient formatting features that can help with that.

The format() Method

To produce a formatted value or set of values, you create a string to be used as a format, then call the string's **format()** method. The format string can be made up of literal text (which becomes part of the formatted string) and *replacement fields*. The replacement fields are surrounded by curly brackets (**{ }**). You provide the values you want to format as arguments to the method. The values are interpolated into the format string to produce a formatted string, which becomes the result of the method call. The replacement fields contain either the position or the name of the argument whose value should be used (you'll learn more about named arguments later).

Let's look at some of the capabilities that this offers. Start an interactive session and enter the following commands to see how formatting works:

```
INTERACTIVE SESSION:

cold1:~$: python3
>>> "{2}, {1}, and {0}".format("George", "Paul", "John")
'John, Paul, and George'
>>> "{who} is a smart {what}".format(what='cookie', who='Sylvia')
'Sylvia is a smart cookie'
>>> "The fifth element of the first argument is {0[5]}".format(
...     ["Dallas", "Zorg", "Cornelius", "Ruby", "Billy", "Leelo"])
'The fifth element of the first argument is Leelo'
>>> d = {'Cher': "Sarkisian", 'Sonny': "Bono"}
>>> "Sonny's surname is {0[Sonny]}".format(d)
"Sonny's surname is Bono"
>>> "Cher's surname is {lookup[Cher]}".format(lookup=d)
"Cher's surname is Sarkisian"
>>> for first, last in d.items():
...     print("{0:10} {1:10}".format(first, last))
...
Cher       Sarkisian
Sonny      Bono
>>> fmt = "{0:>6} = {0:>#16b} = {0:#06x}"
>>> for i in 1, 23, 456, 7890:
...    print(fmt.format(i))
...
     1 =               0b1 = 0x0001
    23 =            0b10111 = 0x0017
   456 =         0b111001000 = 0x01c8
  7890 =   0b1111011010010 = 0x1ed2
```

You can see that Python has some very powerful string-formatting capabilities. Now we need to understand the rules and ways to write formats that will give us the output we want. We'll go over these rules in the next few sections.

The curly brackets play a vital role in formatting strings. Each sequence of characters surrounded by a pair of curly brackets is replaced by some representation of an argument to the **format()** method.

Function Arguments

The **format()** method, like all Python functions, can be called with two types of argument. The first type, and the one you are most familiar with, is called *positional*, because it is identified by the position it occupies in the argument list. The second type is called *keyword*; it's preceded by a name and an equals sign.

If a call has any positional arguments, they must always appear before any keyword arguments. Thus, **"…".format(a, b, k1=c, k2=d)** is legal, but **"…".format(k1=c, k2=d, a, b)** is not (it will be flagged as a syntax error by the interpreter).

The arguments to the **format()** method call are the values to be formatted. The format string on which the method is called specifies how the values are to be represented, by including replacement fields. Other text in the format string (that does not appear between curly brackets) is simply copied to the output literally.

Note	To include actual curly brackets in the output, simply put two curly brackets together, **{{** or **}}**. These doubled curly brackets can never occur in a replacement field, and so they are treated specially.

Format Field Names

The first part of the replacement field, immediately following the opening curly bracket, is the *field name*. This tells the formatting engine which value is to be formatted. The field name begins with either a number, which specifies a positional argument to the **format()** method, or a name, which specifies a named argument. This can be followed by extra information that allows you to index the selected argument (which will presumably be an indexable object such as a list, tuple, or dict) or access one of its attributes.

Example Field Name	Meaning
1	The second positional argument
name	The keyword argument called **name**
0.**attr**	The **attr** attribute of the first positional argument
2[0]	Element 0 of the third positional argument (which must be a list, tuple, or dict)
test[**key**]	The element associated with key **key** in the keyword argument named **test**

These features alone can get you pretty far. Let's experiment now and get more comfortable programming by writing a slightly unusual program. Usually we expect to provide variable data to a program and format its results in a standard way. This time we'll provide you with standard data and let you enter format specifications that will select specific elements for display.

Type the following code as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Accept format strings from the user and format fixed data."""
i = 42
r = 31.97
c = 2.2 + 3.3j
s = "String"
lst = ["zero", "one", "two", "three", "four", "five"]
dct = {"Jim": "Dandy",
    "Stella": "DuBois",
    1: "integer"}
while True:
    fmt = input("Format string: ")
    if not fmt:
        break
    fms = "{"+fmt+"}"
    print("Format:", fms, "output:", fms.format(i, r, c, s, e=lst, f=dct))
```

☐ Save it in your **/python1** folder as **formatting.py** and run it; verify that you get the answers shown for the inputs

given in the following table:

Input	Output	Explanation
0	{0} : 42	First positional argument
1	{1} : 31.97	Second positional argument
2.imag	{2.imag} : 3.3	The **imag** attribute of the third positional argument
{{3}}	{{{3}}} : {String}	A left curly bracket (specified by **{{**) followed by the fourth positional argument, followed by a right curly bracket (specified by **}}**)
e[0]	{e[0]} : zero	Element zero of the keyword argument named **e**
f[Stella]	{f[Stella]} : DuBois	The element of the keyword argument named **f** indexed by the string **"Stella"**
f[1]	integer	The element of the keyword argument named **f**, indexed by the integer **1**

To exit the program, press **Enter**. The program takes whatever you enter, wraps it inside curly brackets, and uses the constructed string as a formatting string against four positional arguments and two keyword arguments. The **print()** call will only output a single value, but you can vary the format to get all kinds of results.

If you don't understand the results, your instructor can help cast some light on the topic.

Format Specifications

The formatting mechanism has some pretty sophisticated ways to *select* what is formatted. Now let's see about actually *formatting* the selected value. We do that by following the field name with a colon and a *format specification*. This can include details about the filling mechanism to be used, how the output is to be aligned in the field, how to treat the signs of numbers, how wide the field should be, how many digits of precision to allow, or what type of conversion should be performed on the selected value.

The various components of the format specification must appear in a prescribed order. No component is required.

Padding and Alignment

Padding clears an area around the content (inside the border) of an element. You don't need to specify a padding character, but if you do specify padding, you must specify the field's alignment as well. There are four different characters that you can use to specify the field's alignment. If the alignment specifier is preceded by some other character, that character is used to pad the field to the requested width; otherwise the space character is used. The alignment options are:

Alignment Option	Meaning
<	The field is left-aligned in the available space, with any padding to its right. This is the default when no alignment is specified.
>	The field is right-aligned in the available space, with any padding to its left.
=	(Valid only for numeric types). Forces the padding to be placed after the sign but before any digits. This can be used to print padded numeric values with the signs all aligned above each other. Pad characters are typically "0" or "*".
^	The field is centered within the available space. Padding characters will be added on the left and right.

No padding is required if the value occupies the whole width of the field. If no width is specified, this will always be the case, and no padding will ever be inserted.

Sign

As you may have guessed, we don't specify signs for non-numeric values. The interpreter would raise a **ValueError** exception if it found such a sign specification. There are three ways we can use signs:

Option	Meaning
+	Insert a **+** sign for positive values, a **-** sign for negative values.

-	Insert a **-** sign for negative values, no sign for positive values.
space	Insert a **-** sign for negative values, a space for positive values

Base Indicator

The base indication can only be requested for integers whose values are being displayed in hexadecimal, octal, or binary (simmer down, we're going to talk about this stuff more in a few minutes). To request it, include a hash mark (**#**) in the format specification. When a base indicator is requested, binary numbers are preceded by **0b**, octal numbers by **0o** and hexadecimal numbers by **0x**.

Digit Separator

To use commas as thousands separators (for example, 9,999,999), insert a comma in the format specification. This may restrict your programs' portability, as some locales use a comma as a decimal point and a period as a thousands separator. To keep your code as portable as possible, use locale-dependent types of specifications (more on this in a few minutes).

Field Width

The field width is a decimal integer specifying the total width of the output generated by the format specifier. As a special case, if the field width begins with a zero character ('0'), it is treated as a shorthand for a pad character of '0' and a fill type of '=' (zeroes between the sign and the digits). This is illegal for non-numeric values and will raise a **ValueError** exception under those circumstances.

Precision

Precision is specified as a period followed by a decimal number. For numeric values, this indicates how many significant digits to display:

```
INTERACTIVE SESSION:

>>> "{0:15.5}".format(987.654)
'         987.65'
>>> "{0:15.5}".format(98765.432)
'     9.8765e+04'
```

For other types of values, it indicates how many characters will be used from the field content.

Field Type

Last of all comes a letter that dictates which type of value should be formatted. For string values, the letter can be omitted, or can be **s**. All numeric types can also be formatted with a field type of **s**, in which case the resulting value before alignment and truncation (yeah, I said it: *truncation*—aka limiting the number of digits right of the decimal point) is the same as that produced by applying the built-in **str()** conversion. Complex number values cannot be formatted in the same way as real and integer values; instead, you must format the real and imaginary parts separately. You can access these parts using the **.real** and **.imag** attribute qualifiers in the field names. Integer and long values can be formatted with these field types:

Type	Field Type
b	Binary: formats the number in base 2.
c	Character: converts the number to the corresponding Unicode character.
d	Decimal: formats the integer in base 10.
o	Octal: formats the integer in base 8.
x	Hexadecimal: formats the number in base 16, using lower-case letters a through f for the digits from 10 to 15.
X	Hexadecimal: like **x**, but uses upper-case letters.
n	Like **d**, but uses the locale settings to determine the decimal point and thousands separator characters.
No code	Treated the same as **d**.

Floating-point and decimal values use a separate set of type codes:

Type	Field Type
e	Exponential notation: formats in scientific notation using **e** to indicate the exponent.
E	Same as **e** but uses an upper-case exponent indicator.
f	Fixed-point. Displays the number as a fixed-point number, using "nan" to represent "not a number" and "inf" to represent infinity.
F	Same as **f** but upper-case: uses "NAN" and "INF."
g	General format. Uses fixed-point format unless the number is too large, in which case it uses exponent notation with lower-case indicators.
G	Like **g** but uses upper-case indicators.
n	Like **g** but uses the current locale settings to determine decimal point and thousands separators.
%	Multiplies the number by 100 and displays in **f** format followed by a percent sign.
No code	Treated similarly to **g** except that it always produces at least one digit after the decimal point and by default uses a precision of 12.

Variable-Width Fields

The field width and the precision are numeric values. If you want these values to be reliant on program data, you can pass the width and precision as arguments to the **format()** method and then use a nested field name inside the format specification. This nested field name (which must refer to an integer value) is substituted for the field width or precision as the formatting takes place. So, for example, **"{0:{1}.{2}f}".format(1234.5678, 18, 3)** displays the number 1234.5678 to three decimal places in a field 18 characters wide.

Let's try a few examples. Start up an interactive session and enter the commands shown:

```
INTERACTIVE SESSION:

>>> "{0:010.4f}".format(-123.456)
'-0123.4560'
>>> "{0:+010.4f}".format(-123.456)
'-0123.4560'
>>> for i in 1, 2, 3, 4, 5:
...     "{0:10.{1}f}".format(123.456, i)
...
'     123.5'
'    123.46'
'   123.456'
'  123.4560'
' 123.45600'
>>> n = {'value': 987.654, 'width': 15, 'precision': 5}
>>> "{0[value]:{0[width]}.{0[precision]}}".format(n)
'        987.65'
```

The numerical rounding is always correct. And by using dict access, you can carry the value, field width, and precision (along with other values you might need) all within a single object.

A Simple Listing Program

This example program lists the names, ages, and weights of a number of individuals. Currently the data is stored as a list of tuples. We'll list the data using formatting statements. Enter this code in the editor window:

```
#!/usr/local/bin/python3
"""Produce a listing of people's names, ages and weights."""
data = [
        ("Steve", 59, 202),
        ("Dorothy", 49, 156),
        ("Simon", 39, 155),
        ("David", 61, 135)]
for row in data:
    print("{0[0]:<12s} {0[1]:4d} {0[2]:4d}".format(row))
```

☐ Save it in your **/python1** folder as **person_list.py** and run it. While this program works, the correspondence between related data items seems a little obscure. Modify the program as shown below to extract the individual items from the row and pass them as separate arguments to the **format()** call:

```
#!/usr/local/bin/python3
"""Produce a listing of people's names, ages and weights."""
data = [
        ("Steve", 59, 202),
        ("Dorothy", 49, 156),
        ("Simon", 39, 155),
        ("David", 61, 135)]
for name, age, weight in data:
    print("{0:<12s} {1:4d} {2:4d}".format(name, age, weight))
```

☐ Save and run it again. The results are the same, but which code do you think is easier to read?

Okay, now let's make the name field wider. We'll use the period as a pad character to help the reader follow the line from the name to the age and weight. Modify the program a third time—add a padding character before the alignment indication and increase the field width:

```
#!/usr/local/bin/python3
#!/usr/local/bin/python3
"""Produce a listing of people's names, ages and weights."""
data = [
        ("Steve", 59, 202),
        ("Dorothy", 49, 156),
        ("Simon", 39, 155),
        ("David", 61, 135)]
for name, age, weight in data:
    print("{0:.<30s} {1:4d} {2:4d}".format(name, age, weight))
```

☐ Save and run it again.

Check You Out!

You've learned so much about Python's formatting features! They can really help make the output from your programs readable and usable.

In the next lesson, we'll return to functions and talk about the role of keyword arguments and parameters like the ones we used with the string **format()** method in this lesson.

See you there!

More About Looping

Lesson Objectives

When you complete this lesson, you will be able to:

- use the range() function with loops.
- use the enumerate() function.
- use the While loop when you need to validate user input.
- use loops to add, retrieve, and delete data.
- create a dictionary using an enumerated loop.
- combine the pieces of code and programming skills you learn in this lesson to make a program that prepares a list of invitations.

Earlier, you learned about **for** and **while** loops. The **for** loop repeats the loop body once for each element of a container. The **while** loop repeats the loop body continuously, testing before each execution whether some condition is true; it stops only when the condition becomes false (or a **break** statement is executed, which terminates any loop). In this lesson, we'll show you some other cool things you can do with loops.

Fun with the range() function.

Can you say 'Python is fun' 1,000 times fast? You might not be able to, but Python can!

```
INTERACTIVE SESSION:

>>> for i in range(1000):
...     print("Python is fun!")
...
Python is fun!
Python is fun!
[... 996 lines omitted ...]
Python is fun!
Python is fun!
>>> range(1000)
range(0, 1000)
>>> type(range(1000))
<class 'range'>
```

You just printed **Python is fun!** one thousand times, using the **range()** function, which generates arithmetic progressions. When we ask the interpreter to print out the result of a call to **range(1000)**, it doesn't print out a list or a tuple, as you might expect. In fact, **range()** returns a special type of object known as a *range* object. You can iterate over this object just like you can iterate over a list [0, 1, 2, ..., 998, 999]. Using the object is different from using a list because it produces the numbers one by one as needed. This saves time and storage space that would be needed to construct a list instead.

The last example printed a string constant. In this next example, we'll print out some numbers:

```
INTERACTIVE SESSION:

>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

Did you notice that the **4** did not print? The range starts at zero, so the given end point is never part of the generated

sequence. This may be confusing at first, but the interpreter has good reason for doing it like that. You'll see in this next example:

```
INTERACTIVE SESSION:

>>> names = ['John', 'Paul', 'George', 'Ringo']
>>> for i in range(3):
...     print(names[i])
...
John
Paul
George
```

(Nothing personal, Ringo; we just wanted to make a point.) Generally, if you want to print only the names, you wouldn't loop over the indexes and use them to select the appropriate list elements. Instead you would loop over the list directly. If you ever see code like **for i in range(len(something))**, that normally indicates what's sometimes called a *code smell*. I know, funny, right? It's code that works, but still stinks a little. Code smells are usually an indication that something needs to be changed.

By now you can see that **range()** is a bit like indexing—it starts counting at zero (unless you tell it to start somewhere else) and goes on until just before it gets to the end value. What if you want a range of numbers that starts at 5 and ends at 7? You give **range()** two arguments instead of one:

```
INTERACTIVE SESSION:

>>> for i in range(5, 8):
...     print(i)
...
5
6
7
```

Remember the *stride* we used with lists? We can do it in **range()** as well. Add a third argument as shown:

```
INTERACTIVE SESSION:

>>> for i in range(10, 40, 10):
...     print(i)
...
10
20
30
```

You can also use a negative stride if you want a numerically descending sequence. In the next example, you'll see that again, the sequence stops before it actually reaches the final value:

```
INTERACTIVE SESSION:

>>> for i in range(10, -30, -10):
...     print(i)
...
10
0
-10
-20
```

Using the enumerate() function

The range function is really useful and powerful. But what if you need to step through a set of numbers by tens *and* track which iteration you are in? For example, when counting by tens:

```
0 10
1 20
2 30
3 40
4 50
```

We can do that, right? We'll provide a counter variable and increase it with each iteration:

INTERACTIVE SESSION:

```
>>> c = 0
>>> for i in range(10, 60, 10):
...    print(c, i)
...    c += 1
...
0 10
1 20
2 30
3 40
4 50
```

This method works, but Python gives us a better way: the function **enumerate()**. Like **range()**, it generates a sequence of values, but in this case, the values are tuples, each containing two elements. The first element is a counter that starts at zero, and the second element is the current item from the sequence that was given as an argument to **enumerate()**. In a **for** loop, you can use a tuple of two names to receive the elements, similar to the unpacking assignments we used earlier. In the example below, **i** is the index and **e** is the element from the sequence:

INTERACTIVE SESSION:

```
>>> for i, e in enumerate(range(10, 60, 10)):
...     print(i, e)
0 10
1 20
2 30
3 40
4 50
```

Now we'll take a look at two ways to print out a numbered list of names. There's more than one way to do it; Python has an older way of formatting, not deprecated, still works, based on the C language printf. Then it has a new "formatting mini-language" more like C#, introduced with Python 3 but also back-ported to 2.6 and above, using numbers in {curly brackets} to identify objects to format, and the .format() function we learned about in the last lesson. The format() version has more bells and whistles and makes it easier to do certain things. Also, one could argue it's cleaner in not requiring a separate operator. Here's an example using the old way:

INTERACTIVE SESSION:

```
>>> names = ['John', 'Paul', 'George']
>>> for i, name in enumerate(names):
...     print('%s. %s' % (i+1, name))
...
1. John
2. Paul
3. George
```

Now, the same thing using the new way of formatting:

```
>>> for i, name in enumerate(names):
...     print('{0}. {1}'.format(i+1, name))
...
1. John
2. Paul
3. George
```

Same results, different methods. We'll usually use the .format method in this course, but you're likely to encounter the %s method in the real world, so we'll use it occasionally.

> **Note** In the above examples, we add one to the count because, although Python counts from zero, we humans normally prefer to start at one.

A More Complex While Loop Example

Suppose you want to print a list of all the factorials under 1000. In mathematics, N factorial is written as "N!" A factorial is calculated by multiplying successive numbers together:

OBSERVE: Factorial Calculations

```
1! = 1 = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 6
4! = 1 x 2 x 3 x 4 = 24
```

I'm about to get mathematical and academic on you for a minute here, so settle in. The textbook definition of *n factorial* (as long as n is a non-negative integer) is the product of all positive integers less than or equal to n. Factorials are used in calculus, combinatorics, and probability theory.

You *could* figure out all factorials under 1000 by figuring out the pattern of calculations manually, but that would get pretty tedious with larger sets, don't you think? In order to avoid that agony and its inherent potential for errors, let's use a **while** loop to resolve the calculation instead. Create a new file in the editor window as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Print all factorials less than 1000."""

c = 0
f = 1
while (f < 1000):
    print(f)
    c += 1
    f = 1
    for n in range(c, 0, -1):
        f = f * n
```

☐ Save it in your **/python1** folder as **factorial.py** and run it. The program prints all the factorials under 1000.

There are actually *two* loops here. The first (or *outer*) loop uses the c variable to simply count upwards. The second (or *inner*) loop generates the factorial, based on the value of the counter.

Each iteration of the *outer* loop increments our counter variable **c** by 1, copies that to **n**, and resets the factorial variable **f** to 1. The *inner* loop does its work by taking the value of **n**, which is simply a copy of the counter, and multiplying that repeatedly against the factorial variable.

So, if you already know **N!**, then you can produce **(N+1)!** (the next value in the sequence) by multiplying **N!** by **N+1**. You can make this program even more efficient by avoiding the second loop, since the second loop would be run for each

factorial. This saves a lot of work. Give it a try. Edit the code as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Print all factorials less than 1000."""

c = 1
f = 1
while (f < 1000):
    print(f)
    c += 1
    f *= c
```

☐ Save and run it. The program produces the same sequence of values, but it does not repeat work unnecessarily. This becomes more important as your programs expand.

While Loops and User Input Validation

We can use the While loop when we need to validate user input. It lets us return the user back to the prompt until they provide a valid response. To implement this feature, we create an infinite loop that can only be broken by correct action by the user.

Suppose we want to force the user to provide a yes or no response. Create a new file in the editor window as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Validate user input"""

while True:
    s = input("Type 'yes' or 'no':")
    if s == 'yes':
        break
    if s == 'no':
        break
    print("Wrong! Try again.")
print(s)
```

☐ Save it in your **/python1** folder as **validate_input.py** and run it. The console asks you to type yes or no. Instead, type **spam** and press **Enter**. The console responds with **Wrong! Try again.**. If you enter anything besides **yes** or **no**, you'll get the same response. When you finally enter **yes** or **no**, you break the While loop and your entry is printed.

The problem with this program is that it doesn't adapt itself well to more options. For example, if you need to add **maybe** as a possible response, that involves adding two lines of code and modifying a third. With your

validate_input.py in the editor window, click the Save As icon to save it in your **/python1** folder as **better_validate_input.py**, and edit it as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Validate user input"""

valid_inputs = ['yes', 'no', 'maybe']
input_query_string = 'Type %s: ' % ' or '.join(valid_inputs)
while True:
    s = input(input_query_string)
    if s in valid_inputs:
        break
    print("Wrong! Try again.")
print(s)
```

☐ Save and run it. In this new program, the **valid_inputs** list variable is used to build the string that queries the user for

input. It's also used to validate the user's input. So in order to add an option, you can just replace **valid_inputs = ['yes', 'no', 'maybe']** with **valid_inputs = ['yes', 'no', 'maybe',** *'another option'***]**.

More sophisticated user interface validation **while** loops are used in applications such as operating systems and web site registration systems.

Dicts and Loops

Earlier, we learned about a useful construct for handling data called *Dicts*. In this next set of examples, we'll use loops to add, retrieve, and delete data. Eventually we'll combine everything into one large example to handle invitations to a party.

In the first example, you'll create a dictionary using the words of the phrase **Python is awesome**, using an enumerated loop to do all the hard work:

```
INTERACTIVE SESSION:

>>> data = {}
>>> for index, word in enumerate('Python is awesome'.split(' ')):
...     data[index] = word
...
>>> print(data)
{0: 'Python', 1: 'is', 2: 'awesome'}
```

Keep this interactive session open. First you created an empty dict, then enumerated over a list containing the words split out of the "Python is awesome" string. With each execution of the loop body, you added the index of the loop as a dict key, using the word as the value of the dict element. You can do this over any list of data, from a list of words to lines of text in a file.

Our next example uses the **items()** method for retrieving data from a dict. **items()** returns a generator object, which then produces two-element tuples of keys and their corresponding values from the dict:

```
INTERACTIVE SESSION:

>>> data.items()
dict_items([(0, 'Python'), (1, 'is'), (2, 'awesome')])
>>> for element in data.items():
...     print(element)
...
(0, 'Python')
(1, 'is')
(2, 'awesome')
>>> for key, value in data.items():
...     print(key, value)
...
0 Python
1 is
2 awesome
```

The last two commands you entered are extremely useful, because they allow you to access all the data in a dict quickly. This is a very common pattern in working with dicts in Python. The dict's **items()** method produces (key, value) pairs, and the **for** loop unpacks the tuples and binds them to **key** and **value**, respectively.

Of course, there will be times when you'll need to remove key/value pairs from a dict. Suppose you had a dict whose keys were words, and you wanted to remove all *noise words* (words that are not normally indexed, such as 'is' and 'at'). You can use a loop to accomplish this task:

```
>>> noise = ['is', 'at']
>>> data
{0: 'Python', 1: 'is', 2: 'awesome'}
>>> for key, value in data.items():
...     if value in noise:
...         del data[key]
...
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
RuntimeError: dictionary changed size during iteration
>>> data
{0: 'Python', 2: 'awesome'}
```

The data was deleted, but the deletion changed the size of data.items, which Python reports as an error. To avoid this problem, you need to produce a separate list rather than iterating over the dict's items (or keys) directly:

```
>>> data = {}
>>> for index, word in enumerate('Python is awesome'.split(' ')):
...    data[index] = word
...
>>> for key, value in list(data.items()):
...    if value in noise:
...      del data[key]
...
>>> data
{0: 'Python', 2: 'awesome'}
```

We used the same techniques here that we used in earlier examples to loop through the key/values of the dictionary. And in this new example, when one key/value matched one of the listed prepositions, it deleted the element of the dict that contained that noise word. Can you think of a data structure that would have been better than a list to hold the noise words?

A More Complex Example

Good programmers build applications out of smaller code pieces. Our final example in this lesson will give you a chance to do just that. You'll combine the pieces of code and programming skills you've learned in this lesson to make a program that prepares a list of invitations. The program will take input as commands from the user. There are five commands: "add" to add a name to the invitation list, "delete" to delete a name, "approve" to approve an invitation that has been added, "list" to list the current invitations, and "quit" to terminate the program's operations.

Create a new file in the editor window as shown:

```
#!/usr/local/bin/python3
invites = {}
options = ['add', 'list', 'approve', 'delete', 'quit']
prompt = 'Pick an option from the list (%s): ' % ', '.join(options)
status_1 = 'unapproved'
status_2 = 'approved'
while True:
    inp = input(prompt)
    if inp not in options:
        print('Please pick a valid option')
        continue
    if inp == 'add':
        name = input('Enter name:')
        if not name:
            continue
        invites[name] = status_1
    elif inp == 'list':
        for name, status in invites.items():
            print('%s (%s)' % (name, status))
    elif inp == 'approve':
        for name in invites:
            if invites[name] == status_1:
                break
        else:
            print('There must be %s status invites. Please pick another option' % statu
s_1)
            continue
        while True:
            print('Please enter a valid name from the list below')
            unapproved = []
            for name in invites:
                if invites[name] == status_1:
                    unapproved.append(name)
            print(", ".join(unapproved))
            name = input('Enter name:')
            if not name:
                break # user changed mind about approving
            if name in unapproved:
                invites[name] = status_2
                print('%s %s' % (name, status_2))
                break
    elif inp == 'delete':
        if not invites:
            print('There must be invites before you delete any of them')
            continue # user changed mind about deleting
        while True:
            print('Please enter a valid name from the list below')
            for name, status in invites.items():
                print('%s (%s)' % (name, status))
            name = input('Enter name:')
            if not name:
                break
            if name in invites:
                del invites[name]
                print('%s deleted' % name)
                break
    elif inp == 'quit':
        print('Quitting invites')
        print('The final invitation list follows')
        for name, status in invites.items():
            print('%s (%s)' % (name, status))
        break
```

☐ Save it in your **/python1** folder as **invite.py** and run it. The program is really just one input validation loop that checks to make sure that the user has entered one of the five available commands. If the user has not done this, the

program repeats the request for input. Most of the commands require further input, and each command allows the user to just press the **Enter** key to ignore the command and request another.

> **Note**   We used the %s method for formatting our prompts here. You should be able to change the program to use the .format() method.

Loop This

We love loops because they let us repeat the same logic again and again as necessary. This means that your program can execute some pretty complex behaviors, particularly when one loop contains others.

In the next lesson, we'll learn how programs can use and store information in files. See you there!

Reading and Writing Files

Lesson Objectives

When you complete this lesson, you will be able to:

- create a file.
- write to a file.
- read files as text.
- append to a file.
- seek to arbitrary positions.
- create a file-based to-do list.
- read binary data.

So far, your programs have used values that were either encoded in the program or provided by the user, and the data disappeared after the session ended. But sometimes you'll need to use data from outside sources, or to permanently save data you've created or modified in a program. Python has built-in functions to handle actions such as creating, writing, and reading files. In this lesson, you'll learn how to use files to create, retrieve, and change data.

Creating a New File

To create a file, you use the **open()** function. This function returns a file object. If you try to open a file that does not exist, the interpreter creates a file with that name for you. To see how the **open()** function works to create a file, start an interactive session and enter these commands:

```
INTERACTIVE SESSION:

cold1:~$ cd python1
cold1:~/python1$  python3
>>> f = open('funnies.txt', 'w')
>>> f
<_io.TextIOWrapper name='funnies.txt' encoding='UTF-8'>
```

This statement assigns the result of the call to the **open()** function to the variable **f**. The **open()** function opens a file to read, write, or append. The first argument gives the name **'funnies.txt'** to the working file (because we changed to the **/python1** folder before we started the python3 interpreter, this file will reside in that folder). The second argument **'w'** tells **open()** that we want to **write** to the file.

When you ask the interpreter to display that variable, it displays the name of the open file associated with a particular object. Right-click the **/python1** folder in your File Browser on the left, and select **Reload**—it now contains an empty **funnies.txt** file. (Don't delete the file or close the interactive console because you'll need access to it in the next section!)

> **Note**   Depending on your environment, you might see an encoding type other than UTF-8.

Writing to a File

Writing to a file is a good way to save information. Python provides two different ways to write to a text file. Let's return to the interactive seesion and take a look:

```
>>> f = open('funnies.txt', 'w')
>>> f.write('Larry\n')
6
>>> f.write('Curly\n')
6
>>> f.write('Moe\n')
4
>>> names = ['Groucho\n', 'Chico\n', 'Harpo\n']
>>> f.writelines(names)
>>> f.close()
```

The **write()** method adds the string (for example, **'Moe\n'**) to the current content of your funnies.txt file, and returns the length of the string added (for example, 4—including the **\n** as one character). The **writelines()** method takes a list of strings and adds each element to the funnies.txt file. Unlike the **print()** function, neither method adds newlines to the content it writes, so we include '\n' with each string so it will be on a separate line in the file. The **close()** method makes sure that all data is written out to the file and that the connection between it and the program is dropped.

> **Note** Look at **funnies.txt** now if you like—you'll see the six names you added. Be sure to close it before you continue with the lesson.

Reading Files as Text

Now that you have some sample data in the **funnies.txt** file, let's see what Python provides to read its contents. Reopen the file in a readable mode. When you open a file for reading, the file object returned by the **open()** function is iterable, which means that if you use it in a **for** loop, each iteration gets the next line from the file until there are no more lines left. Go back to the interactive session and type in the commands as shown:

```
>>> f = open('funnies.txt', 'r')
>>> f.read()
'Larry\nCurly\nMoe\nGroucho\nChico\nHarpo\n'
>>> f = open('funnies.txt', 'r')
>>> f.readline()
'Larry\n'
>>> f.readlines()
['Curly\n', 'Moe\n', 'Groucho\n', 'Chico\n', 'Harpo\n']
>>> f = open('funnies.txt', 'r')
>>> for line in f:
...     print(line)
...
Larry

Curly

Moe

Groucho

Chico

Harpo

>>> f.read()
''
>>> f.close()
```

Notice that we opened the **funnies.txt** file three times. That's because the file content is "used up" by using the **read()**, **readline()**, and **readlines()** methods, and we have to reopen the file to return to the top. **read()** returns all of

the content of the file as a single string. **readlines()** returns the file content as a list of lines. **readline()** returns the next line from the file, so when you called it once and then called **readlines()**, the second call returned a list that didn't include the first line of the file. The last method, **f.read()** returns nothing because the "pointer" is at the end of the file.

When we're done with the file, we close it with the **close()** method. This releases resources that Python was using to look at or write to the file. Some programmers assume files will close automatically at the end of a program, but it's better programming "hygiene" to close files when you finish using them.

Appending to a File

There are six lines of text in your funnies.txt file. Let's add some more. If you open an existing file with the write option **w**, you truncate the file's contents and produce an empty file—any new input will replace the file's original contents. So we *don't* use the write (**w**) option, using the append option (**a**) instead, as the second argument to **open()**. The next example shows the append functionality at work. Enter these commands in your interactive Python console:

```
INTERACTIVE SESSION:

>>> f = open('funnies.txt','a')
>>> f.write('A child of five could understand this. Fetch me a child of five.\n')
65
>>> f.write('Room service? Send up a larger room.\n')
37
>>> f.close()
>>> f = open('funnies.txt', 'r')
>>> for line in f:
...     print(line[:-1])
...
Larry
Curly
Moe
Groucho
Chico
Harpo
A child of five could understand this. Fetch me a child of five.
Room service? Send up a larger room.
>>>
>>> f.close()
```

So, you opened an existing file to append content, and wrote in two new lines before closing it. When you opened it again, all of the old content was followed by the new content you had just written. In this example, you used **[:-1]** to slice each line to exclude the last character (the newline) from the end. This prevents your code from producing the blank lines that were printed out in the previous example.

Seeking to Arbitrary Positions

As we've seen, when a file is being read or written, it has a "current position." You can change this position using the **seek()** method. The first argument should be the position you want to move to within the file (an integer—the beginning of the file is always position 0).

If you give a second integer argument, it must be **0**, **1**, or **2**:

Value	Meaning
0	Position is relative to the start of the file (the first argument cannot be negative). This is the default if you don't supply an argument.
1	Position is relative to the current position (the first argument can be negative to move backward or positive to move forward).
2	Position is relative to the end of the file (the first argument must be negative).

Continue your interactive session:

```
>>> f = open('funnies.txt', 'r')
>>> f.seek(115)
115
>>> f.read()
'Send up a larger room.\n'
>>> f.close()
```

You can't **seek()** past the end of a file. To find the current position in a file, call the **tell()** method.

More File Details

Python file objects give you many handy attributes and methods that you can use to analyze a file, including tools to figure out the name of the file associated with the file object, whether a file is opened or closed, readable, writable, how it handles errors, and if it is seekable. Type the commands as shown below:

```
>>> f = open('funnies.txt','a')
>>> f.name
'funnies.txt'
>>> f.readable()
False
>>> f.writable()
True
>>> f.seekable()
True
>>> f.encoding
'UTF-8'
>>> f.errors
'strict'
>>> f.closed
False
>>> f.close()
>>> f.closed
True
```

Creating a File-Based To-Do List

The built-in **open()** function provides one method to add *persistence* to your applications. In this case, *persistence* means that when you turn off or quit the application, the data remains available. So when you use an application to store information, you can end your Python session, turn off your computer, and still come back and find that data later. Persistence can take many forms, from the types of files we're using in this lesson, to database records, to audio/video files, to various document formats.

Sophisticated persistence engines are called databases. Most of the world's data is stored in databases. These are integrated sets of logically organized files or records. Databases can store text, integers, dates, images, and much more. Usually, databases use the relational model, but there are also hierarchical, object, network, and flat-file models.

To demonstrate the power of persistence, and ways to take advantage of it, we'll create a to-do list application as our next example. Create a new file as shown:

```
#!/usr/local/bin/python3
"""File-based to-do list maintainer."""

otasks = open('open_tasks.txt','a')
otasks.close()
dtasks = open('done_tasks.txt','a')
dtasks.close()
options = ('add','done','quit')
string_input = 'Pick an option from the list (%s): ' % ', '.join(options)
while True:
    open_tasks = open('open_tasks.txt','r').readlines()
    if open_tasks:
        print('-' * 10)
        print('Open Tasks')
        print('-' * 10)
        for i, task in enumerate(open_tasks):
            print(i, task.strip())
    done_tasks = open('done_tasks.txt','r').readlines()
    print('-' * 12)
    print('Done Tasks')
    print('-' * 12)
    for i, task in enumerate(done_tasks):
        print(i, task.strip())

    inp = input(string_input)
    if inp not in options:
        print('Please pick a valid option')
        continue
    if inp == 'add':
        new_task = input('Enter new task: ')
        tasks = open('open_tasks.txt','a')
        tasks.write(new_task + '\n')
        tasks.close()
    if inp == 'done':
        while True:
            done_task = input('Please enter the number of your completed task: ').strip
()
            if done_task.isdigit():
                done_task = int(done_task)
                break
            print('Please enter a task number')
        open_tasks = open('open_tasks.txt','r').readlines()
        for i, task in enumerate(open_tasks):
            if i == done_task:
                print('Task removed: {0}'.format(task))
                open_tasks.remove(task)
                f = open('open_tasks.txt','w')
                f.writelines(open_tasks)
                f.close()
                f = open('done_tasks.txt','a')
                f.write(task)
                f.close()
                break
    if inp == 'quit':
        break
```

☐ Save it in your **/python1** folder as **todo.py**, and run it. Without adding any tasks, type **quit** and press **Enter**. This program starts out by opening each of two data files to append, and then closes them immediately. This forces the computer to create the files, in case this is the very first run. Right-click your **/python1** folder in the File Browser at left and select **Reload**, and you'll see the two new files, **open_tasks.txt** and **done_tasks.txt**.

Now, run the program again and create a few tasks. Quit the program and start it again. The tasks you added are still there! And just like that, you've implemented a persistence engine that uses the flat file model! Open the files and you'll see your tasks have been added to **open_tasks.txt**.

Reading Binary Data

So far you've stored simple text data, which is easy to read with any text editor. However, simple text data is no substitute for the audio files that store our favorite music! But if we open audio files with a text editor, it displays what appears to be a lot of gibberish. Actually, these files are storing the audio data encoded in binary form. This data is impossible for us to read without a special tool called a **hex editor**. Fortunately, your computer doesn't have the same limitations. And because the binary file doesn't have to be comprehensible to humans, in many cases it can represent data more efficiently than a simple text file.

All the audio, image, and video files on your computer are binary files. So are any files compressed into zip or tar format. In fact, the majority of programs on your computer—such as your favorite browser—are comprised of binary files, the notable exception being the Python programs you are writing as part of this course. And even those Python programs are compiled into binary format before the computer actually runs them!

Now that you have a background in binary data and files, let's check out how Python can handle a binary file. We'll use wget to grab the image below:

Type the following code in an interactive console:

```
INTERACTIVE TERMINAL SESSION:

cold1:~$ cd python1
cold1:~/python1$ wget "https://courses.oreillyschool.com/Python1/images/lessons/python-
logo.gif"
--2011-11-14 13:14:26--  https://courses.oreillyschool.com/Python1/images/lessons/pytho
n-logo.gif
Resolving courses.oreillyschool.com... 199.27.144.89
Connecting to courses.oreillyschool.com|199.27.144.89|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 2549 (2.5K) [image/gif]
Saving to: "python-logo.gif"
100%[==============================================================>] 2,549 --.-K/s in
 0s
2011-11-14 13:14:27 (26.8 MB/s) - "python-logo.gif" saved [2549/2549]
cold1:~/python1$ python3
>>> i = open('python-logo.gif', 'rb')
>>> i
<_io.BufferedReader name='python-logo.gif'>
>>> print(i.read(1))
b'G'
>>> print(i.read(1))
b'I'
>>> print(i.read(1))
b'F'
>>> i.read(10)
b'89a\xd3\x00G\x00\xf7\x00\x00'
>>> i.tell()
13
>>> i.seek(0)
0
>>> i.read(3)
b'GIF'
>>> i.close()
```

When you first look at binary data, it can be pretty daunting. But even so, at a glance we can see a useful method and a handy bit of information. The **read()** method fetches the byte(s) you request. Subsequent **read()** requests are fired from your current location in the file. The first three bytes provide the format of the file you are examining (in this case, GIF). This way a program can figure out how to handle a file even if the file extension is missing. In fact, all modern browsers check this information before displaying images for you.

So, what about the **b'89a\xd3\x00G\x00\xf7\x00\x00'**? Well, that's part of the image content used to generate the Python logo. Our example above also shows the **tell()** and **seek()** methods. **seek(0)** "rewinds" the file to the beginning.

Adding an integer argument to the **read()** method instructs your program to read the given number of bytes. If there aren't enough bytes in the file, **read(*n*)** returns as many as there are. This means that if you get an empty sequence of bytes back, you are at the end of the file.

Finally, the "strings" that you get when you read a file in binary mode are what we call *byte strings*—each byte is eight bits, so the *ordinal value* of the characters is in the range 0 to 255. If you aren't familiar with the binary system, don't worry. Just be aware that Python strings in binary differ from regular Python strings in that you can represent pretty much any character (as long as your character set includes it).

Files for Miles

So now you know a little more about files, the basic way to provide persistent storage of information. Files are the basis of most computer-based information storage, so there is a huge amount of literature that covers how to store various types on information in files. For now, the basics will suffice. You can write data out from one program run, and read it back in to make use of it in another program (or a different run of the same program). This is what gives computers the power to run systems with long-term memory.

I'm thinking you're feeling pretty confident about working with files now. But if you have any questions, go ahead and ask your mentor. They're here to help! Good job so far and see you in the next lesson!

Python's Built-In Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- use various built-in functions.

In every previous lesson, you've used some of Python's built-in functions. The first built-in function you used was **print()** back in lesson 1. Since then you've used built-in functions like **range()**, **open()**, and more. Built-in functions are an invaluable part of your Python tool kit. In this lesson, we'll learn even more about them.

First we'll go over some examples that use Python built-in functions, and then explore the functions themselves.

Party Fun with Built-In Functions

Suppose you're throwing a party. Each invitation to your party might have 0, 1, 2, or more people attached to it. You are storing the invitations in a pair of lists. The first list holds the names of the attendees; the corresponding element in the second list is the size of the invited group. You need to know the total number of people attending in order to buy the right amount of food for the party, and for seating purposes, you need to know who has the largest group. Python's built-in functions will help you to execute both of these tasks. Start an interactive session as shown below:

```
INTERACTIVE SESSION:

cold1:~$ python3
>>> invites = ["Jay", "Conan", "Jimmy", "Craig"]
>>> attendees = [3, 2, 0, 5]
>>> sum(attendees)
10
>>> zipped = zip(attendees, invites)
>>> party = tuple(zipped)
>>> party
((3, 'Jay'), (2, 'Conan'), (0, 'Jimmy'), (5, 'Craig'))
>>> max(party)
(5, 'Craig')
>>> for people, name in party:
...     print(people, name)
...
3 Jay
2 Conan
0 Jimmy
5 Craig
>>> for people, name in sorted(party):
...     print(people, name)
...
0 Jimmy
2 Conan
3 Jay
5 Craig
```

Keep the console open. This example helps demonstrate a couple of new functions: **sum()** returns the total of all the elements in its argument; **zip()** interleaves (that is, alternates, like the teeth of a zipper) the elements of any number of sequences. If you call **zip()** with two arguments, when you loop over the result you get a sequence of two-element tuples; call it with three arguments and you get three-element tuples. **zip()** doesn't return a list or a tuple, but something called a generator; we called the **tuple()** function on it so we could see the data without needing to loop over it.

Next, using the same data, let's check to see if any or all invitations have any attendees and the total number of invitations. Type the commands below as shown:

```
>>> any(attendees)
True
>>> all(attendees)
False
>>> len(attendees)
4
```

The **any()** function returns **True** if any element of its argument is true. **all()** returns **True** if all of the elements of its argument are true. **len()** returns the number of elements in the argument.

The rest of this lesson provides an alphabetical reference guide to Python's built-in functions, with brief examples. As you become more familiar with Python, you'll find new and innovative ways to make use of these built-in functions.

abs(x)

The **abs()** function returns the absolute value of an integer, floating point, or complex number. The returned value is always positive. If the input value is a negative integer or floating-point number, then the absolute value is the negated argument. If the argument is complex, a positive result will still be returned, but it's a complicated calculation (the square root of the sum of the squares of the real and imaginary components). Take a look. Type the commands below as shown:

```
>>> abs(3.14)
3.14
>>> abs(-3.14)
3.14
>>> abs(3+4j)
5.0
```

all(iterable)

The **all()** function returns **True** if all elements of the supplied iterable are true (or if there are no elements: technically, you could say it returns **False** if any element evaluates as false). So if all elements in a list, tuple, or set match Python's definition of being true, then **all()** returns **True**. Type the commands below as shown:

```
>>> lst = [1, 2, 3, 4, 5, 6]
>>> all(lst)
True
>>> lst.append('')
>>> all(lst)
False
>>> all([])
True
>>> t1 = ("Tuple")
>>> all(t1)
True
>>> t2 = ("Tuple", "")
>>> all(t2)
False
>>> s = {}
>>> all(s)
True
```

any(iterable)

The **any()** function is the converse of the **all()** function. **any()** returns **True** if any element of the iterable evaluates true. If the iterable is empty, the function returns **False**. type the commands below as shown:

```
>>> lst = ["", 0, False, 0.0, None]
>>> any(lst)
False
>>> lst.append("String")
>>> any(lst)
True
>>> any([])
False
>>> any(("", 0))
False
>>> any(("", 1))
True
>>> any({})
False
>>> any({0: "zero"})
False
>>> any({"zero": 0})
True
```

bool(x)

The **bool** function converts the value to a Boolean, using the standard Python truth testing procedure. If *x* is false or omitted, it returns **False**; otherwise it returns **True**. Type the commands below as shown:

```
>>> bool("Python is fun!")
True
>>> t = []
>>> bool(t)
False
>>> bool(0)
False
>>> bool()
False
>>> bool(1)
True
```

chr(i)

The **chr()** function returns a string of one character, which has the ordinal value equal to the given integer. Type the commands below as shown:

```
>>> chr(90)
'Z'
>>> alphabet = ''
>>> for letter in range(65, 91):
...    alphabet += chr(letter)
...
>>> alphabet
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

dict(arguments)

dict() creates a new data dictionary with items taken from the arguments. If no arguments are passed, an empty dictionary is created. You can call **dict()** with a tuple or list as its argument. In those cases, each of the argument's elements must be a two-element (key, value) list or tuple. You can also use a sequence of *keyword arguments*. We will cover those in the lesson on functions, but in short, a keyword argument is a name followed by an equals sign and a value. Try this example:

```
>>> {'number': 3, 'string': 'abc', 'numbers': [3, 4, 5]}
{'numbers': [3, 4, 5], 'number': 3, 'string': 'abc'}
>>> dict([(1, "one"), [2, "two"], (3, "three")])
{1: 'one', 2: 'two', 3: 'three'}
>>> dict(zip("ABCDEF", range(10, 16)))
{'A': 10, 'C': 12, 'B': 11, 'E': 14, 'D': 13, 'F': 15
>>> dict(
...    number=3,
...    string="abc",
...    numbers=[3, 4, 5]
... )
{'number': 3, 'string': 'abc', 'numbers': [3, 4, 5]}
```

dir(arguments)

The **dir()** function can accept any argument: string, integer, dictionary, function, class, or method. Without arguments, **dir()** returns the list of names in the current local scope. If an argument is given, then the result is a list of the names in the namespace of the given object. The list returned is always sorted in alphabetical order. Type the commands below as shown:

```
>>> p = 'Python'
>>> dir(p)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__', '__format_
_', '__ge__', '_
_getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__', '_
_iter__', '__le__
', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__red
uce_ex__', '__rep
r__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__
', '_formatter_fi
eld_name_split', '_formatter_parser', 'capitalize', 'center', 'count', 'encode', 'endsw
ith', 'expandtabs
', 'find', 'format', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit', 'isidentifi
er', 'islower', '
isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', '
lstrip', 'maketra
ns', 'partition', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstri
p', 'split', 'spl
itlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

globals()

The **globals()** function returns a dictionary representing the current global symbol table. This is always the namespace dictionary of the current module. Type the commands below as shown:

```
>>> list(globals().items())
[('__builtins__', <module 'builtins' (built-in)>), ('__name__', '__main__'), ('__doc__'
, None), ('__package__', None)]
>>> first = "Hello"
>>> second = "Goodbye"
>>> list(globals().items())
[('__builtins__', <module 'builtins' (built-in)>), ('__package__', None), ('second', 'G
oodbye'), ('__name__', '__main__'), ('__doc__', None), ('first', 'Hello')]
```

Note If you see more keys listed than are displayed in this example, it's probably because you've been trying different snippets of code.

help(object)

The **help()** function is your new best friend. Invoke the built-in help system on any object and it will return usage information on the object. For experienced Python programmers, this is the first tool to use when trying to figure out something they don't understand. Once you start writing more advanced Python programs, you'll learn how to write your own help text.

In an interactive Python console, use the **help(*object*)** function on any variable, string, integer, list, tuple, set, or built-in function, including the **help()** function. Some of the text won't make sense to you right now, but you'll still find this function very useful. To scroll through larger help documents, press the space bar. To exit, press **q**. Type the commands as shown:

```
>>> help(globals)
Help on built-in function globals in module builtins:

globals(...)
    globals() -> dictionary

    Return the dictionary containing the current scope's global variables.
>>> help(len)
Help on built-in function len in module builtins:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or mapping.
>>>
```

len(s)

len(s) returns the length of an object. The argument provided may be a sequence (string, tuple, or list) or a mapping (dictionary). Type in these commands:

```
>>> s = "Python"
>>> len(s)
6
>>> lst = [1, 2, 3]
>>> len(lst)
3
>>> d = {"a":"b", "c":"d", "e":"f"}
>>> len(d)
3
```

locals()

The **locals()** function returns a dictionary representing the current local symbol table. Unless it's called inside a function, it will return the same list as **globals()**. Type in this command:

```
>>> locals()
{'__builtins__': <module 'builtins' (built-in)>, '__name__': '__main__', '__doc__': Non
e, '__package__': None}
```

> **Note** Just like the **globals()** function, you will likely see more keys than we show in this example. That's perfectly fine; it means you've probably been testing different snippets of code. Good for you!

max(iterable)

The **max()** function, with a single argument iterable, returns the largest item of a non-empty iterable (such as a string, tuple, or list). With more than one argument, it returns the largest of the arguments. Type these commands:

```
>>> lst1 = [16, 32, 8, 64, 2, 4]
>>> max(lst1)
64
>>> lst2 = ['one', 'two', 'three', 'One', 'Two', 'Three']
>>> max(lst2)
'two'
>>> max(42, 76, -104)
76
>>> max(1, 2, "three")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

Don't close the session—we'll use lst1 and lst2 with the next function! The first result is 64. The second result of 'two' might have surprised you, but Python compares strings "lexicographically" (like they would be sorted for a dictionary, but with all the lower-case letters greater than any upper-case one), not by the meaning of the words. The last expression caused an error, because you can't compare strings and integers: they are fundamentally different types.

min(iterable)

The opposite of the **max()** function, **min(iterable)** returns the smallest item of a non-empty iterable (such as a string, tuple, or list). With more than one argument, it returns the smallest of the arguments. Type these commands:

```
>>> min(lst1)
2
>>> min(lst2)
'One'
>>> min(42, 76, -104)
-104
>>> min(1, 2, 'three')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unorderable types: str() > int()
```

The first result is 2. The second result of 'One' seems to make sense, but be aware that Python returned the lowest value of an alphanumeric sort, "O" being less than "T": Python neither knows nor cares about the meaning of the words. The fourth expression raised an exception here as well, because you can't compare strings and integers.

ord(c)

ord(c) is the inverse of the **chr()** function we discussed earlier. Given a string of length one, it returns an integer representing the ordinal value of the character. For example, ord('A') returns the integer 65. Type these commands:

```
>>> alphabet = 'ABCDEFGH'
>>> for letter in alphabet:
...     print(ord(letter), letter)
...
65 A
66 B
67 C
68 D
69 E
70 F
71 G
72 H
```

pow(x, y[, z])

pow(x, y[, z]) returns *x* to the power *y*. If a third argument *z* is given, then the result is reduced modulo *z* (then you get the remainder after dividing (*x* raised to the power *y*) by *z*).

You might have played with a calculator at one time or another, using repeated multiplication to raise a number to successive powers. In the next example, Python automates the calculations for you. Type these commands:

```
>>> pow(2, 2)
4
>>> pow(2, 3)
8
>>> pow(2, 4)
16
>>> for i in range(5, 12):
...     print(pow(2, i), pow(2, i, 100))
...
32 32
64 64
128 28
256 56
512 12
1024 24
2048 48
```

reversed(seq)

reversed(seq) is a reverse iterator on an object of the type that you can loop through and process. The **list** and **tuple** types are supported with this function, but the **set** type is not (because the elements of a set aren't ordered). Type these commands:

```
>>> lst = [1, 2, 3]
>>> reversed(lst)
<list_reverseiterator object at 0x01E4DC70>
>>> for i in reversed(lst):
>>>     print(i)
...
3
2
1
```

round(x[, n])

The **round(x[, n])** function rounds the decimal value *x* to the nearest integer. If you give a second argument *n*, it rounds to that number of decimal places. Type these commands:

INTERACTIVE SESSION:

```
>>> round(33.5)
34
>>> round(33.3333333333, 2)
33.33
```

sorted(iterable)

sorted(iterable) returns a new sorted list from the items in *iterable*. This arranges your lists, tuples, and sets in a known order. Type these commands:

INTERACTIVE SESSION:

```
>>> numbers = [3, 1, 6, 7, 1100, 10]
>>> sorted(numbers)
[1, 3, 6, 7, 10, 1100]
>>> t = ['Beta','beta','alpha','Alpha']
>>> sorted(t)
['Alpha', 'Beta', 'alpha', 'beta']
>>> lst2 = ['one', 'two', 'three', 'One', 'Two', 'Three']
>>> sorted(lst2)
['One', 'Three', 'Two', 'one', 'three', 'two']
```

The first sorted list provides an expected result. The second list you may not have anticipated. Python sorts in alphanumeric order, but all upper-case letters sort lower than all lower-case letters.

You can also use keyword arguments to specify how the sort keys should be created, and whether to sort in ascending or descending order. Suppose you want to have a case-insensitive search. You can do this by using a function as the **key** argument of the sort. In this case, you use the Python string type's lower-case method. In the second example, you request a descending sort with the **reverse** keyword argument. Type these commands:

INTERACTIVE SESSION:

```
>>> t = ['Beta','beta','alpha','Alpha']
>>> sorted(t, key=str.lower)
['alpha', 'Alpha', 'Beta', 'beta']
```

Note When you use the **lower()** function on otherwise identical strings like 'Beta' and 'beta', Python treats them as identical, keeping them in the same order they were input, so 'beta' might not appear before 'Beta' when you try the above example.

INTERACTIVE SESSION:

```
>>> t = ['Bete','beta','alphie','Alpha']
>>> sorted(t, key=str.lower)
['Alpha', 'alphie', 'beta', 'Bete']
>>> sorted(t, reverse=True)
['beta', 'alphie', 'Bete', 'Alpha']
```

sum(iterable)

sum(iterable) sums the numeric values in an *iterable* such as a list, tuple, or set. **sum(iterable)** does not work with strings because you can't do math on strings (when you add two strings you are really using an operation called *concatenation*). Type these commands:

```
INTERACTIVE SESSION:

>>> s = {1, 2, 3}
>>> sum(s)
6
>>> lst = ['Python','is','fun!']
>>> sum(lst)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

So we are able to add up numbers, but things break down on letters and words. To combine strings in a list, rely on the string **join()** method as shown below:

```
INTERACTIVE SESSION:

>>> lst = ['Python','is','fun!']
>>> ' '.join(lst)
'Python is fun!'
```

zip(*iterables)

The **zip()** function takes iterables and aggregates elements from each of the iterables into a new iterable object. That might sound complicated, but the example below will help illustrate the concept. Type these commands:

```
INTERACTIVE SESSION:

>>> lst_1 = ['Python','is','fun']
>>> lst_2 = [1000, 2000, 3000]
>>> lst_3 = [10, 9, 8, 7, 6, 5]
>>> list(zip(lst_1, lst_2))
[('Python', 1000), ('is', 2000), ('fun', 3000)]
>>> list(zip(lst_1, lst_2, lst_3))
[('Python', 1000, 10), ('is', 2000, 9), ('fun', 3000, 8)]
```

In the first result, we used the **list()** function to create a list of three tuples. The second example is not so clear, as we are missing the last two elements of **lst_3**. That's because the zip function ignored iterations for which it didn't have elements in all of the supplied iterables. This enables us to create dicts using the **zip()** function. Try it out:

```
>>> lst_1 = ['Python','is','fun']
>>> lst_3 = [10, 9, 8, 7, 6, 5]
>>> d = {}
>>> for k, v in zip(lst_1, lst_3):
...    d[k] = v
...
>>> d
{'Python': 10, 'fun': 8, 'is': 9}
>>> zip((1, 2), (3, 4))
<zip object at 0x01AD1E18>
```

zip() returned a generator called a "zip object."

Fun with Built-In Functions

You've worked with lots of different functions in this lesson, and used them to get a real idea of how they work.

Keep your interpreter window open to test your understanding of new functions as you come into contact with them. Experiment and try to find their limits. Use the **help()** function to learn more about the built-in functions too.

You're looking good so far. Keep up the great work!

Defining and Calling Your Own Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- define and call your own functions.
- access the arguments using the names of the parameters.
- return a value that your code displays via the built-in **print()** function.
- access arguments using the names of the parameters.
- use functions to handle namespaces.
- create parameters that receive multiple arguments.

Exploring Functions

Hi and welcome to a new lesson! We're moving right along, huh? Let's keep it going. While working through examples so far in this course, you typed similar pieces of code over and over again. Take a look at this snippet of code from our earlier *complex file handling* example:

OBSERVE:

```
open_tasks = open('open_tasks.txt','r').readlines()
if open_tasks:
    print('-' * 10)
    print('Open Tasks')
    print('-' * 10)
    for i, task in enumerate(open_tasks):
        print(i, task.strip())
done_tasks = open('done_tasks.txt','r').readlines()
print('-' * 10)
print('Done Tasks')
print('-' * 10)
for i, task in enumerate(done_tasks):
    print(i, task.strip())
```

You typed in almost exactly the same code twice. Wouldn't it be nice if you could just write it once and then call it whenever you needed it, like you've done with Python's various built-in functions? Fortunately, you can! Look at the same code, this time rewritten using a function:

OBSERVE:

```
def task_report(task_file):
    tasks = open(task_file,'r').readlines()
    if tasks:
        print('-' * 10)
        print(task_file.replace('_',' ').replace('.txt','').title())
        print('-' * 10)
        for i, task in enumerate(tasks):
            print(i, task.strip())
task_report('open_tasks.txt')
task_report('done_tasks.txt')
```

The second example defines a function **task_report**. It executes the same task as the first example, but operates by calling the function twice. The differences between the two examples are in the file name and the heading that was printed out. The file name is a *formal parameter* (**task_file**) of the function, and the heading is created by using **replace()** to change the underscore (**'_'**) to a space (**' '**) and the extension (**'.txt'**) to nothing (**''**) in the file name, and then applying title case to the remainder with **title()**. (The net result? 'open_tasks.txt' becomes 'Open Tasks' and 'done_tasks.txt' becomes 'Done Tasks.')

By defining and using this function, we saved three whole lines of code. Impressed? No?! You should be. But wait, there's more!

Now, suppose you need to add three more types of task state to your code: "not yet confirmed," "in testing," and "under review." Without our task_report function, we would have had to add 18 lines of code to accomplish this task! But using the function, we can process each file with a single line, and get the job done with just three lines of code! Take a look at the example:

```python
def task_report(task_file):
    tasks = open(task_file,'r').readlines()
    if tasks:
        print('-' * 10)
        print(task_file.replace('_',' ').replace('.txt','').title())
        print('-' * 10)
        for i, task in enumerate(tasks):
            print(i, task.strip())
task_report('open_tasks.txt')
task_report('done_tasks.txt')
task_report('not_yet_confirmed.txt')
task_report('in_testing.txt')
task_report('under_review.txt')
```

Good Python developers never repeat a stanza of code twice. Instead, we put it into a function, and call that function as often as we need it. If you see lots of repetitive code, it generates that yucky *code smell* I mentioned earlier. The code might work, but changing it, maintaining it, and using it in other places will be harder than it needs to be and the code will be more prone to errors.

Write Your First Function

Let's take a shot at writing a function. We'll write some code that averages a list of values. In the editor window, type the code as shown below:

```python
#!/usr/local/bin/python3
def average(lst):
    """ Averages a list, tuple, or set of numeric values"""
    return sum(lst) / len(lst)

tst_lst = [1, 2, 3, 4]
print('Average this list: {0}'.format(tst_lst))
print(average(tst_lst))
t = (243, 132, 987, 342, 13)
print('Average this tuple: ',t)
print(average(t))
s = {1, 2, 3, 4, 25}
print('Average this set: {0}'.format(s))
print(average(s))
```

☐ Save it in your **/python1** folder as **average.py**, and run it:

```
cold1:~$ cd python1
cold1:~/python1$ ./average.py
Average this list: [1, 2, 3, 4]
2.5
Average this tuple: (243, 132, 987, 342, 13)
343.4
Average this set: {25, 2, 3, 4, 1}
7.0
```

How does it work?

```
#!/usr/local/bin/python3
def average(lst):
    """ Averages a list, tuple, or set of numeric values"""
    return sum(lst) / len(lst)

tst_lst = [1, 2, 3, 4]
print('Average this list: {0}'.format(tst_lst))
print(average(tst_lst))
t = (243, 132, 987, 342, 13)
print('Average this tuple: ',t)
print(average(t))
s = {1, 2, 3, 4, 25}
print('Average this set: {0}'.format(s))
print(average(s))
```

The function occupies only the first three lines of code in this example. The Python keyword **def** introduces a function definition. It must be followed by the **function name** and the **list of formal parameters** in parentheses. The code that makes up the function is called the *function body*. The function body must be indented. The string **""" Averages a list, tuple, or set of numeric values"""** is the function's documentation string, often abbreviated as *docstring*. The interpreter uses docstrings to give programmers information about how the function works and how it should be called. Finally, the last line tells the function to **return** the **sum()** of the values entered, divided by the number of values as determined by the **len()** function. This returned value becomes the value of the function call during evaluation of expressions.

Our function is followed by test code that lets us verify that the function works correctly. Each time the **average()** function is written with a list of numbers in it, such as **average(tst_lst)** or even **average([10,20,30,40,50])**, our function code is run with those numbers as formal parameters.

In the average.py example, we used the name **lst** for our parameter. It could have any name, but for the sake of clarity, use a name that makes the purpose of the variable clear. Also, be careful not to use names of existing Python functions or other objects. You don't want to use **list** or **tuple** as variable or parameter names, for example, because they are the names of Python built-in functions. If you do, your program may behave in completely incomprehensible ways.

Parameters and Arguments

Parameters are the names you give to the inputs to the function when the function is defined. *Arguments* are the values you provide when you call the function. Inside the function body, your code can access the arguments using the names of the parameters.

Suppose you want to write a function that prints out the elements in a list, and you want to provide an option to have the function print the list in reverse order. To do this, you'll use *positional* and *keyword* parameters. Type the code as shown:

```
#!/usr/local/bin/python3
def print_list(lst, rev=False):
    """ prints the contents of a list. """
    if rev:
        lst = reversed(lst)
    for i in lst:
        print(i)

print_list(['Printing', 'a', 'list'])
print()
print_list(['Printing', 'a', 'reversed', 'list'], True)
print()
print_list(lst=['A', 'list', 'with', 'specified', 'arguments'],rev=False)
```

☐ Save it in your **/python1** folder as **print_list.py**, and run it. This function takes two parameters. You're familiar with the first parameter, **lst**; the second parameter, **rev=False**, introduces a new feature—a keyword parameter, which has a default value (the value following the equals sign, which in this case is **False**). If you call the function without passing an argument corresponding to the **rev** parameter, it uses that default value.

The function's code looks at the value of **rev**, and if it is true, it re-binds the parameter to a reversed copy of the list. It does this rather than reversing the list in place, because such a reversal would affect code outside of the function (though there's nothing illegal about changing a mutable object inside of a function, you want to make sure that the users of the function know they should expect such changes. We'll go over parameters and arguments in greater detail in future lessons).

Returning Values

The first function you wrote in this lesson, **average()**, returned a value that your code then displayed via the built-in **print()** function. When a function call is written in an expression (for example, in **print(average(tst_lst))**), the value of the function call in that expression is actually the value that the function returned in its **return** statement (2.5). But the second function you created, **print_list()**, did not include a **return** statement. This is equivalent to the function ending with **return None**. So all functions will return some value, but by convention, functions that don't need to return anything can implicitly return **None**. If the function isn't intended to return a value, it's confusing to add an explicit **return** statement.

You can either use the function calls in control flow code (that is, code that controls the order in which tasks are executed, such as **if** or **while** statements) or save the values returned by functions, binding them to a variable in an assignment statement and using that value again and again without needing to rerun the function. To see these principles in action, create a new file as shown:

```
CODE TO TYPE:
#!/usr/local/bin/python3
def structure_list(text):
    """Returns a list of punctuation in a text"""
    punctuation_marks = "!?.,:;"
    punctuation = []
    for mark in punctuation_marks:
        if mark in text:
            punctuation.append(mark)
    return punctuation


text_block = """\
Python is used everywhere nowadays.
Major users include Google, Yahoo!, CERN and NASA (a team of 40 scientists and engineer
s
is using Python to test the systems supporting the Mars Space Lander project).
ITA, the company that produces the route search engine used by Orbitz, CheapTickets,
travel agents and many international and national airlines, uses Python extensively.
The YouTube video presentation system uses Python almost exclusively, despite their
application requiring high network bandwidth and responsiveness.
This snippet of text taken from chapter 1"""

for line in text_block.splitlines():
    print(line)
    p = structure_list(line)
    if p:
        print("Contains:", p)
    else:
        print("No punctuation in this line of text")
    if ',' in p:
        print("This line contains a comma")
    print('-'*80)
```

☐ Save it in your **/python1** folder as **return_value.py**, and run it. The **structure_list()** function accepts a single parameter called *text*. This value is checked to find common punctuation marks. These results are placed into a list and that list is returned.

The tricky part is the loop itself and what it does with the returned value of **structure_list()**. Instead of immediately printing the value, we save it to the variable **p**. This variable is subsequently used in two different **if** statements. The first checks to see if the list **p** is empty, then prints an appropriate result. Then the variable is used again to determine whether or not a comma is present.

Multiple Return Values

So, what if you need to return two values? Suppose that, in addition to the punctuation in our last example, you also want to return the location of the word "Python." You could write a second function, but it's often more efficient when the two results require related logic in order to have your function return another value. Try out the example below and get a better look at this concept:

```
#!/usr/local/bin/python3
def structure_list(text):
    """Returns a list of punctuation and the location of the word 'Python' in a text"""
    punctuation_marks = "!?.,:;"
    punctuation = []
    for mark in punctuation_marks:
        if mark in text:
            punctuation.append(mark)
    return punctuation, text.find('Python')

text_block = """\
Python is used everywhere nowadays.
Major users include Google, Yahoo!, CERN and NASA (a team of 40 scientists and engineer
s
is using Python to test the systems supporting the Mars Space Lander project).
ITA, the company that produces the route search engine used by Orbitz, CheapTickets,
travel agents and many international and national airlines, uses Python extensively.
The YouTube video presentation system uses Python almost exclusively, despite their
application requiring high network bandwidth and responsiveness.
This snippet of text taken from chapter 1"""

for line in text_block.splitlines():
    print(line)
    p, l = structure_list(line)
    if p:
        print("Contains:", p)
    else:
        print("No punctuation in this line of text")
    if ',' in p:
        print("This line contains a comma")
    if l >= 0:
        print("Python is first used at {0}".format(l))
    print('-'*80)
```

☐ Save and run it. We modified the function to return a two-element tuple. The first element is the punctuation as computed in the previous version. The second element is the location of the word "Python." If the word doesn't exist in the text, -1 is returned, as determined by the **find()** method's specification.

The function result is assigned to two separate variables using an unpacking assignment, and an additional test is made on the returned index value to determine whether to report the presence of the word "Python."

Functions and Namespaces

Using functions in Python has the added benefit of helping us begin to understand namespaces.

When you call a function, Python dynamically creates a new namespace and binds the argument values to the appropriate parameter names. Assignments made during execution of the function call result in bindings in the function call namespace. When the function returns, the namespace is automatically destroyed, and any bindings inside the namespace are lost.

You can sum up how functions handle namespaces in Python by understanding these two rules:

1. Variables bound within a Python function body only exist in namespaces created by calls of that function.

2. Variables bound in the global namespace can be accessed by functions, but may not be bound unless specifically declared to be global.

Let's test out the first rule. As you will see, the variable **c** defined below is assigned inside of the **test()** function. Start an interactive session and enter the commands shown:

```
>>> def test(a, b):
...     c = a + b
...     return c
...
>>> test(1, 2)
3
>>> c
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'c' is not defined
```

And now let's test the second rule. Type the commands below as shown:

```
>>> def test_a():
...     print(a)
...
>>> test_a()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in test_a
NameError: global name 'a' is not defined
>>> a = "Python"
>>> test_a()
Python
```

You can see that when the function attempts to access a global variable **a**, the function fails in its first call, because **a** has not yet been created in the global environment. The interpreter knows that **a** is not local to the function because the function body contains no assignment to it. Once the variable is created by an assignment in the module namespace, a call to the function succeeds without raising an exception.

So, if any assignment is made to a variable inside a function body, the variable is local to the function. Changing a global variable inside a function body isn't a best practice, but sometimes it's a necessary evil. To achieve that end, you use a **global** statement to declare that the variable, although assigned inside of the function body, is in the module (global) scope. To demonstrate this, type the commands below as shown:

```
>>> def test_a():
...     global a
...     a = "XML"
...     print(a)
...
>>> a = "Python"
>>> test_a()
XML
>>> print(a)
XML
```

Here the value **"Python"** is bound to **a** in module scope. After the function is called, you can see that **a** has been re-bound by the assignment inside of the function.

Parameters That Receive Multiple Arguments

Sometimes when you create a Python function, you don't know how many arguments you are going to get and you want the caller to be able to provide any number of arguments. For example, you may want to create a function that takes all the numbers given as parameters and multiply them together. To do this, we use a special parameter

specification, ***name**. There can be only one such parameter, and it *must* follow any standard positional and/or keyword parameters.

When you prefix the parameter with the asterisk (*) character in the function definition, this tells the interpreter to collect any unmatched positional arguments into a tuple and then bind the tuple to the name following the asterisk in the called function's namespace. Inside of your function, this tuple can be used like any other Python iterable. Let's check it out. Create a new file in the editor window as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
def multiplier(*args):
    """ Multiply the arguments together and return the result.
        Return 0 if nothing is provided.
    """
    if not args:
        return 0
    product = args[0]
    for a in args[1:]:
        product *= a
    return product

print(multiplier())
print(multiplier(1,2,3,4))
print(multiplier(6,7,8,9,10,11,12,13))
print(multiplier(10,20,100))
```

☐ Save it in your **/python1** folder as **argument_list.py**, and run it. The **multiplier()** function, our single parameter **args** (which you can think of as the "sequence parameter") is prefixed with *, so all positional arguments to a call will appear inside of this tuple. The rest of the function is made up of familiar code (if anything is unfamiliar, ask your instructor for a little help). We can call the function with any number of arguments.

The * sequence parameter must follow any standard positional or keyword parameters. This can be useful when regular arguments are also required. For instance, you may want to provide an optional amount to be added to the product. You'd accomplish that by using a keyword argument with a default value of zero. Let's see how this is done. Modify the program as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
def multiplier(total=0.0, *args):
    """ Multiply the arguments together, add a prior total, and return the result.
        Return 0 if nothing is provided.
    """
    if not args:
        return total
    product = args[0]
    for a in args[1:]:
        product *= a
    print("product:", product)
    return product + total

print(multiplier())
print(multiplier(1,2,3,4))
print(multiplier(6,7,8,9,10,11,12,13))
print(multiplier(10,20,100))
```

☐ Save and run it. The first parameter of each set is now passed as the **total**, and the rest as **args**.

Putting It All Together

When you were in grade school, you learned that six times seven (6 x 7) was equivalent to adding six to itself seven times (6 + 6 + 6 + 6 + 6 + 6 + 6). Calculating this the long way took time, so you memorized the end result. If you learned your "times tables" at school, you can probably still respond immediately, even now, when asked "what is six times seven?" Storing something in memory to save the trouble of working it out each time you need the answer is called *caching*.

Caching is taking calculated values from arguments and storing them so that you can return the values if asked to compute a result from the same arguments again later. This way, instead of calculating the same thing a hundred times, you save each calculation the first time you make it, and recall it when needed. When applied to a function, this caching technique is often referred to as *memoization*.

To illustrate, we will use the built-in **input()** method to prompt for two numeric values. The code does multiplication the old way (6 + 6 + 6 + 6 + 6 + 6 + 6). Finally, we'll use the ability of functions to use the global namespace to *cache* the results, so when you try it with big numbers (10 million * 10 million), you don't need to repeat lengthy calculations.

In the code example below, we create a **kid()** function to do the math. Rather than introduce some horrendously complicated function that would be difficult to understand, we'll use a more manageable function. **kid()** does multiplication the hard way! Create a new file in the editor window and type the code shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
""" Demonstrates the need for caching """

def kid(a, b):
    """ Multiplication the hard way """
    c = 0
    for i in range(b):
        c += a
    return c

while True:
    a = input('enter a number: ')
    b = input('enter another number: ')
    a = int(a)
    b = int(b)
    print(kid(a,b))
```

☐ Save it in your **/python1** folder as **caching.py**, and run it. Try it with small numbers first, perhaps 4 and 5. Then try something large like 5 and 10000000 (one and seven zeros). You'll have to wait awhile as the computer adds 5 ten million times. That crazy kid takes forever to do basic math!

Now, modify your **kid()** function so that it maintains a record of the arguments it has been called with, and saves previously-computed results in a global dict so that before it even starts to perform a calculation, it can provide a previously-computed result, thereby saving time. Edit the code below as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
""" Demonstrates the need for caching """

global_cache = {}

def kid(a, b):
    """ Multiplication the hard way """
    if (a, b) in global_cache:
        return global_cache[(a, b)]

    c = 0
    for i in range(b):
        c += a
    global_cache[(a, b)] = c
    return c

while True:
    a = input('enter a number: ')
    b = input('enter another number: ')
    a = int(a)
    b = int(b)
    print(kid(a,b))
    print(global_cache)
    print('-'*40)
```

Now try the program again. Enter 5 * 10000000. Wait a few seconds for the response and try it again. You'll notice the second time it returns almost instantly.

Here, when the function is called, it immediately checks the global **global_cache** dict to see whether this particular set of arguments has been used before. If it has, the *cached* result is immediately returned, bypassing the lengthy computation. If the argument set isn't found in **global_cache**, then it is computed in the usual way, but before the result is returned, it is added to the **global_cache** so this new result can be produced immediately if we ever need it again.

A Solid Foundation

In this lesson, you started to learn how to write functions, understand the difference between parameters and arguments, how return values work, and a little more about namespaces. I'm really impressed with your progress so far! Now that you have a pretty good grip on Python basics, let's move on and learn about modules and imports, and even more about namespacing.

See you in the next lesson!

The Python Standard Library

Lesson Objectives

When you complete this lesson, you will be able to:

- employ the *principle of modularity*.
- construct many independent namespaces and handle them separately.
- import Python modules.
- locate modules.

Increased Versatility

Let's get to work and discuss a key concept in programming, the *principle of modularity*. The idea behind it is that unrelated parts of a system should be kept separate from each other, and related parts should be grouped together.

Python comes with a large set of library modules and packages (we'll talk more about packages in a later course—they're like modules, but with a bit more structure). It's well worth learning about the standard library because it contains modules that can save you time and effort, and also allow you to do some really cool stuff with Python.

Let's try a few experiments using the standard library. First we'll figure out how to import a Python library module. Type the commands shown below in an interactive session:

```
INTERACTIVE SESSION:

cold1:~$ python3
>>> import textwrap
>>> textwrap.wrap("This is a very long piece of text. This should appear as shorter lin
es.", 12)
['This is a', 'very long', 'piece of', 'text. This', 'should', 'appear as', 'shorter',
'lines']
>>> import time
>>> time.time()
1327529513.914024
>>> time.gmtime()
time.struct_time(tm_year=2012, tm_mon=1, tm_mday=25, tm_hour=22, tm_min=12, tm_sec=7, t
m_wday=2, tm_yday=25, tm_isdst=0)
>>> time.asctime(time.gmtime())
'Wed Jan 25 22:13:04 2012'
>>> import base64
>>> base64.encodestring(b"This is a byte string")
__main__:1: DeprecationWarning: encodestring() is a deprecated alias, use encodebytes()
b'VGhpcyBpcyBhIGJ5dGUgc3RyaW5n\n'
>>> s = base64.encodebytes(b"This is a byte string")
>>> base64.decodestring(s)
__main__:1: DeprecationWarning: decodestring() is a deprecated alias, use decodebytes()
b'This is a byte string'
>>> base64.decodebytes(s)
b'This is a byte string'
```

Here you made use of functionality from three standard library modules—textwrap, time, and base64. We have linked the name of each module to the appropriate section of Python's standard library documentation. You get access to the resources of a module by qualifying the module's name with the name of the appropriate resource. So "a.b" means "look in a's namespace and return what is bound to the name b there."

The DeprecationWarning message is in our code to remind those programmers using earlier versions of Python that our strings are now Unicode. In older versions, strings were by default made up of ASCII (8-bit) characters. In Python 3, the **base64.encodestring()** function has been renamed **base64.encodebytes()**. The old name is still available, but not for long, so a message is printed to alert programmers to use the newer name.

Namespaces

Earlier, we discussed Python's *object space*, the location where data objects like integers and strings are stored. We also learned that when you run a program, the interpreter creates a *namespace*. Within *namespace*, values in *object space* are bound to names by assignment statements, function definitions, and such.

A Python program has a "global" namespace, where names are bound by assignments and function definitions within the main body of the program. When you call a function, Python dynamically creates a new namespace and binds the argument values to the parameter names. Assignments made during execution of the function call (normally) result in bindings in the function call ("local") namespace. When the function returns, the namespace is automatically destroyed, and any bindings inside the namespace are lost. On occasion, this means that some of the values will no longer have references. When that happens, the memory used to store those values becomes reclaimable as garbage. (Don't worry if you don't have a grip on all of this stuff just yet. It'll make more sense when we get to the experimentation!)

When we write large programs "monolithically" (as whole chunks), we may inadvertently use the same name for two different purposes at different places in the program. We can avoid that problem by incorporating the principle of modularity into our work; we'll write programs as collections of small chunks that are relatively independent of one another. This will also make our programs easier to read and understand.

With Python, we are able to construct many independent namespaces and handle them separately. The same name can be defined in two different namespaces, because the uses don't collide. When the interpreter looks for the value bound to a particular name, it looks in three specific namespaces. First, it looks in the local namespace (assuming a function call is active). Next, it looks in the global namespace. Finally, it looks in the "built-in" namespace, which holds the names of objects that are hard-wired into the Python interpreter, like exceptions and built-in functions.

Python Modules

A module is a collection of statements that are executed. Every program you have written so far in this course is a Python module. You wrote them as stand-alone programs. When you run a module as a program, the interpreter terminates after all of the code has been executed. Running the program is one way to cause its code to be executed. Another way is to *import* it. When you write **import modx** in your program, the interpreter looks for the **modx.py** file. It also looks for its compiled version: **modx.pyc**. If **modx.pyc** is up to date, it will save the interpreter the work of compiling it.

If the file is not found, an *ImportError* exception is raised. Otherwise, the interpreter executes the code in the module, and binds the module's namespace to the name of the module in the current namespace. So, if **modx** defines a function **f()**, after you have imported the module, you can call that function with **modx.f()**—the dot operator tells the interpreter to look up the name **f** in the namespace bound to the name **modx**.

Suppose module **z** defines function **g()**, module **y** imports module **z**, and your program imports module **y**. You could call the function as **y.z.g()**. The interpreter would look up **y** in the local namespace, retrieving the namespace of module **y**. Then it would look up **z** in that namespace, retrieve the namespace of module **z**, and in that namespace look up the name **g** and retrieve the function.

Okay, I think we've got enough to think about. Let's get busy with some practical application! We'll create a program called **importer.py** that imports a module called **moda**, that in turn imports a module called **modb**. The program is going to call a function defined in **modb**. Create the **moda.py**, **modb.py**, and **importer.py** programs, respectively, as shown, in your **/python1** folder:

CODE TO TYPE:
```
"""moda.py: Imports modb to gain indirect access to the triple function."""

import modb
```

CODE TO TYPE:
```
"""modb.py: Defines a function that can be used by importing the module."""

def triple(x):
    """Simple function returns its argument times 3."""
    return x*3
```

```
#!/usr/local/bin/python3
"""importer.py: imports moda and calls a function from a module moda imports."""

import moda
print(moda.modb.triple("Yippee! "))
```

□ Save them all, and run the **importer.py** program. When it runs, it imports module moda. This binds the moda module's namespace to the name **moda** in the program's (global) namespace. When module moda is imported, its code is executed. This causes module modb to be imported, binding it to the name **modb** on module moda's namespace. When modb is imported by moda, *its* code is executed, and the **def** statement binds the name **triple** to the function definition in modb's namespace.

Now when the interpreter sees the statement **print(moda.modb.triple("Yippee! "))**, it looks up the name **moda** in the global namespace, then looks up the name **modb** in *that* namespace, and finally looks up the name **triple** in *that* namespace. This final lookup returns a reference to the triple function, which is then called with the argument "Yippee! " and your program will print "Yippee! Yippee! Yippee! ".

The namespace labeled "GLOBAL NAMESPACE" is actually the global namespace of the **importer** module run as the main program. This diagram shows the relationship between the namespaces of the various modules:
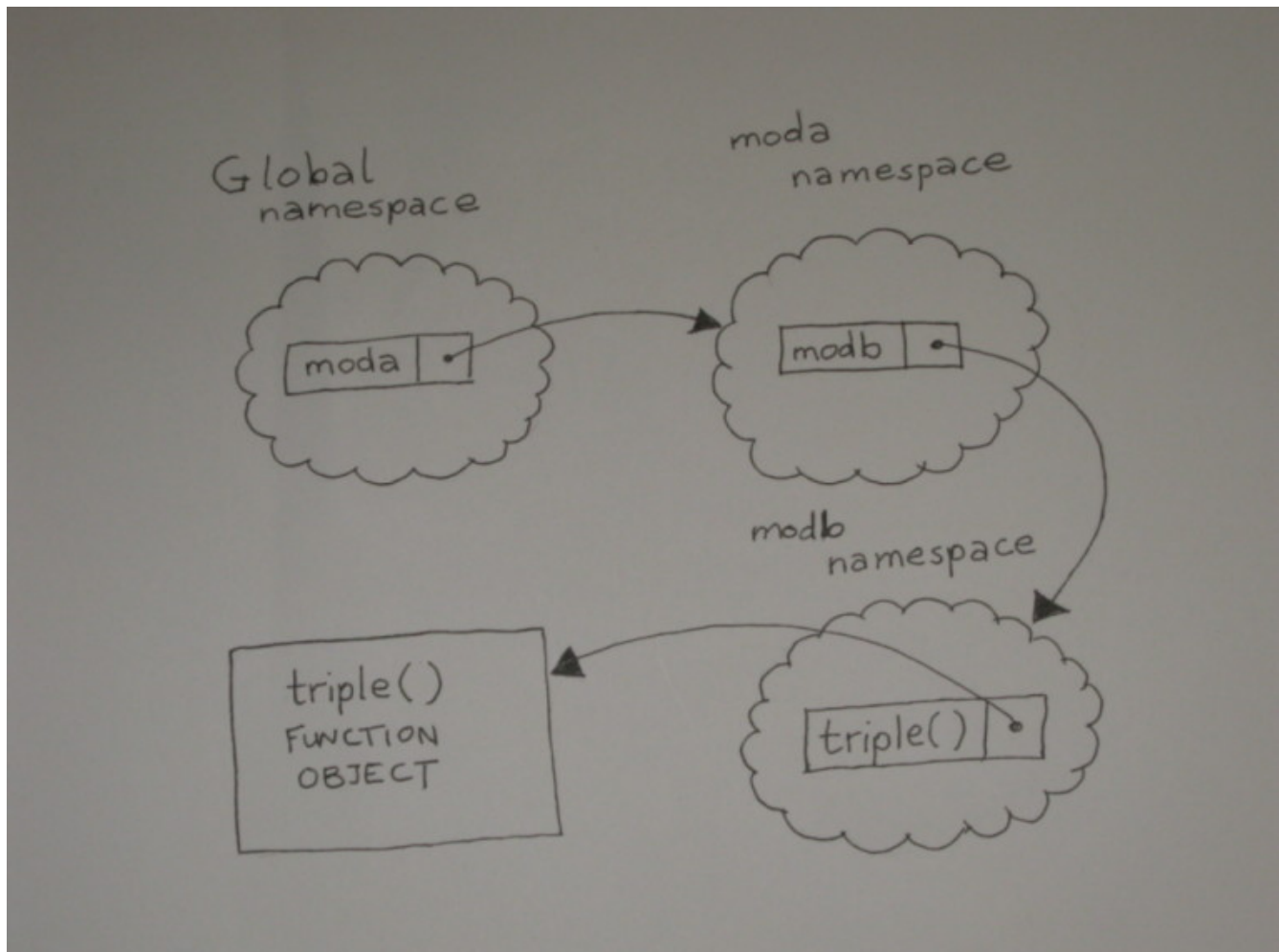
Writing Modules to be Testable

In later courses we will talk about testing code. But even before we start using the *unittest* module, we can start writing importable modules to do some basic testing.

When a module is imported by a program, the interpreter binds the special name **__name__** in the module's namespace to its name. When a module is run as a program, **__name__** receives a special value "__main__". You can assume that your module will be imported, but if it gets run as a program (*that is*, if __name__ == "__main__"), then the user isn't trying to use it, but instead wants to test it.

Some standard library modules have a section at the end that contains the statement:

if __name__ == "__main__"

The code that follows that statement is there to test the module's functionality.

We'll write code like that to test our functions as well, and make it easier to verify that they work as intended. The more you do to make your modules self-testing, the easier it is to detect when a small change has broken the code.

Splitting Up Your Programs

So far almost all of our programs have been made up of single program files. As the programs get more complex, we'll build them as collections of components. A component you build for one program might be useful in another. You could just *copy* the component's code, but then if you needed to modify it, you'd have to modify each copy separately. This makes extra work for you and increases the chance of errors.

Fortunately, Python lets you write your code as a collection of *modules*, each of which is a separate text file. This makes it easier to use your code in various contexts.

Let's take a program that uses functions and split it into two pieces. Create this program in the editor window:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Contains functions to manipulate number representations."""
def commafy(val):
    if len(val) < 4:
        return val
    out = []
    while val:
        out.append(val[-3:])
        val = val[:-3]
    return ",".join(reversed(out))

def commareal(val):
    if "." in val:
        before, after = val.split(".", 1)
    else:
        before, after = val, "0"
    return "{0}.{1}".format(commafy(before), after)

# Testing code only ...
if __name__ == "__main__":
    for i in [0, 1, 12, 123, 1234, 12345, 123456,
            1234567, 12345678, 123456789, 1234567890]:
        print(i, ":", commafy(str(i)), ":", commareal("{0:.2f}".format(i/1000)))
```

☐ Save it in your **/python1** folder as **funcs.py** and run it.

The first module defines the required functions. The second produces results by calling one of the functions. It gains access to the function it needs by importing the module that defines it.

The **commafy** function takes a whole number (which is assumed to be a string comprising all digits) and, beginning from the right, splits it into chunks of three digits. The value string is shortened to remove each chunk after it is added to the **out** list. Any chunk of less than three digits that remains at the end will be captured automatically by slicing. When no digits remain, the **out** list is reversed to put the chunks in the correct order, and the chunks are joined together with commas to provide the function's return value.

The **commareal()** function takes a string representation of a real number or integer. If the string contains a decimal point, it is split around that. If there is no decimal point, a single "0" is used. The **commafy()** function is used to insert commas into the portion before the decimal point, and the output string is constructed from the "commafied" portion before the decimal point and the unchanged portion after the decimal point.

Although this module is designed to be imported by other programs, it will test itself if it's run as a main program. It iterates over a set of integers, printing out the number, its "commafied" version, and the **commareal()** value of the number divided by 1,000 and represented to two decimal places. When the module is imported, the condition **if __name__ == "__main__"** is false, so the testing code does not execute.

Now, create this program:

```
#!/usr/local/bin/python3
"""Take user input, convert to float, and print
out the number to two decimal places, with commas."""
import funcs

while True:
    inval = input("Enter a number: ")
    if not inval:
        break
    number = float(inval)
    print(funcs.commareal("{0:.2f}".format(number)))
```

☐ Save it in your **/python1** folder as **funcalls.py** and run it. This program performs an infinite loop, terminated from within when the user presses **Enter** without typing a number in response to the "Enter a number" prompt. Otherwise, the user's input is converted to a floating-point number, and is formatted back into a string representation with two decimal places. The result of the **commareal()** function is printed back to the user (via **funcs.py**) before the loop repeats.

Other Ways to Import a Module

The **import** statement has some useful variations that can alter the way imported items are made available in the importing namespace.

import ... as

What if you need to import a module, but you've already used its name in your code? You can avoid rewriting your code using the **import ... as** syntax, which allows you to import a module using a name of your choice rather than its natural name. So, if you write **import time as t**, the module is imported in the standard way, *but* rather than being bound to its standard name in the importing namespace, the module namespace is bound to the name **t**. Now you can write a call on the **asctime()** function in the module as **t.asctime()**, and continue to use the name "time" for other purposes.

The **time** namespace is now called **t** in the **__main__** namespace:
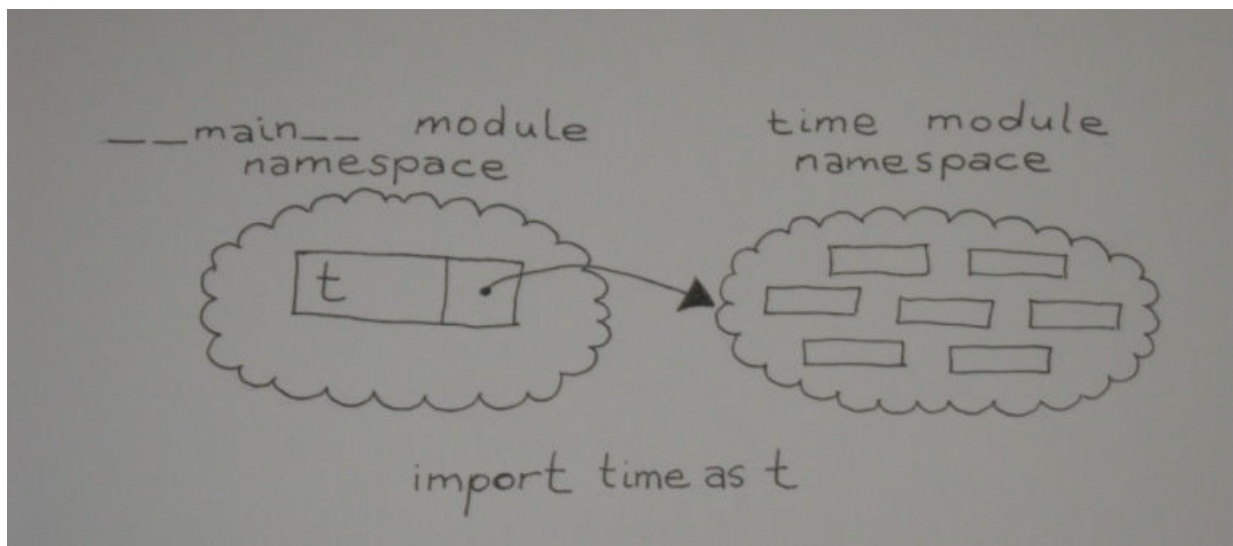
from ... import ...

Sometimes you'll just want to bring the *names* from a module into the importing namespace so they can be used directly rather than qualifying the module name. Do this sparingly though, because once you've done this, it becomes more difficult to locate various resources in the program.

An alternative way to handle situations where the name "time" is already in use, is to import the "asctime" name into the current namespace directly with **from time import asctime**, and write the calls on the function as **asctime()**. Because the **__main__** namespace contains no direct reference to the **time** module,

other names in **time**'s namespace are not available to the **__main__** module. The name **asctime** is copied from the **time** module's namespace to the **__main__** namespace:
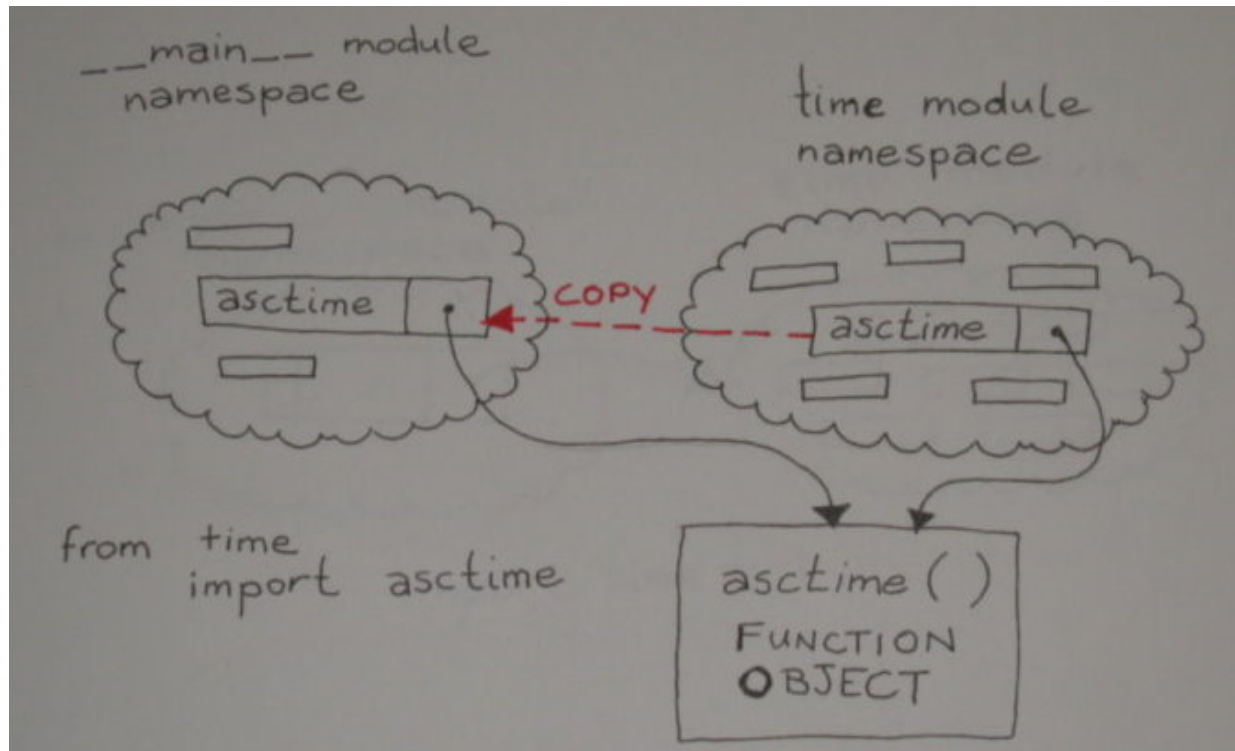
Under most circumstances, you do not want to use **from … import …** to import all names defined in a module using the statement **from module import ***. While this may seem like a great way to define the necessary symbols, it puts the imported module in charge of what gets loaded into your namespace. Unless you are really familiar with the imported module's code, you'll have no way of knowing whether it defined symbols that you're already using. If it did define them, they will overwrite your definitions or your symbols will overwrite the definition from the modules. Either way, you'll receive no notification that this has happened, and you will be left with a tricky debugging exercise.

Certain well-written and sophisticated library modules (such as the Tkinter graphical user interface library) recommend this form of import. Do not try to emulate this in your own designs—it is an invitation to disaster!

The System Path

How does the interpreter know where to find modules? It looks for module **modname** by searching in a specific list of directories for a file called **modname.py**.

Let's look at the system path. It is defined, appropriately enough, in a module called **sys**. You have to import it before you can examine it. To see what's on the path, type the following commands in an interactive session:

```
INTERACTIVE SESSION:

>>> import sys
>>> for p in sys.path:
...     print(p)
...

/usr/local/python34/lib/python34.zip
/usr/local/python34/lib/python3.4
/usr/local/python34/lib/python3.4/plat-linux
/usr/local/python34/lib/python3.4/lib-dynload
/usr/local/python34/lib/python3.4/site-packages
>>>
```

When the interpreter looks for a module, it searches these paths, starting at the top of the list, and stopping when it finds the module. This path can be useful to know if you have a program that doesn't seem to be finding the module

you wanted it to find.

Reduce, Reuse, Recycle!

You're picking this stuff up like a pro! You've learned how your programs can make use of external functionality, and how you can split your own programs up to make them more modular. This will make them easier to manage, help you to write code that can be used in lots of different programs, and make you an efficient programmer! You'll *reduce* your work by *reusing* and *recycling* your code. In the next lesson, we'll revisit functions and learn about even more features. See you there!

More About Functions

Lesson Objectives

When you complete this lesson, you will be able to:

- capture *keyword arguments* whose names do not correspond to the name of any parameter.
- specify keyword parameters in a function that has a dict-parameter.
- import functions and help().
- execute functions by dispatch.

Now that you've got the basics of functions down, we'll build on that knowledge with keyword parameters, switches, importing functions, and more!

Arbitrary Keyword Parameters

We learned earlier that when an unknown number of positional arguments will be provided, you can capture the extra ones (that don't correspond to any of the formal parameters) by specifying a parameter name that is preceded by a single asterisk (*). In much the same way, you can capture *keyword arguments* whose names do not correspond to the name of any parameter. To do that, we'll prefix the last defined parameter with two asterisks, and call a *dict-parameter*. Create a new program in the editor window as shown:

```
CODE TO TYPE:

#!/usr/local/bin/python3
""" Demonstrates capture of keyword arguments"""

def keywords(**kwargs):
    "Prints the keys and arguments passed through"
    for key in kwargs:
        print("{0}: {1} ".format(key, kwargs[key]))

def keywords_as_dict(**kwargs):
    "Returns the keyword arguments as a dict"
    return kwargs

if __name__ == "__main__":
    keywords(guido="Founder of Python", python="Used by NASA and Google")
    print(keywords_as_dict(guido="Founder of Python", python="Used by NASA and Google")
)
```

☐ Save it in your **/python1** folder as **keyword_args.py**, and run it. You output looks like this:

```
OBSERVE:

python: Used by NASA and Google
guido: Founder of Python
{'python': 'Used by NASA and Google', 'guido': 'Founder of Python'}
```

The program has two functions that capture general keyword arguments. When you call such a function, the interpreter matches up the positional and keyword arguments with their corresponding parameters, then takes any unmatched keyword arguments and puts them into a dict, which it binds to the dict-parameter. The first function, **keywords()**, iterates over the keys of the dict, printing the keys (which are the names of the unmatched keyword arguments) and the associated values (which are the values following the equals signs). The second function, **keywords_as_dict()**, just returns the keyword arguments, demonstrating that the dict-parameter is in fact a dict.

Parameters, Sequence-Parameters, and Dict-Parameters

Sometimes you need to mix different argument-passing methods. In an earlier lesson, you learned how to include specific positional parameters in a function that also uses a sequence-parameter. You can also specify keyword parameters in a function that has a dict-parameter.

Suppose you want a function that prints the description of a college course including a name that is a standard positional parameter, an instructor, any number of students, and possibly other staff with assigned roles. To do this in Python, you combine multiple types of parameters. You'll use positional parameters, as well as a sequence-parameter and a dict-parameter. Let's give it a try in a program. Type the code below as shown:

```
#!/usr/local/bin/python3
def description(name, instructor, *students, **staff):
    """Print out a course description.
    name:          Name of the course
    instructor:    Name of the instructor
    *students, ...: List of student names (positional arguments)
    **staff, ...:   List of additional staff (keyword arguments)
    """
    print("=" * 40)
    print("Course Name:", name)
    print("Instructor:", instructor)
    print("-" * 40)
    for title, name in staff.items():
        print(title.capitalize(), ": ", name)
    print("{0:-^40}".format(" registered students "))
    for student in students:
        print(student)

if __name__ == "__main__":
    description("Python 101",
            "Steve Holden",
            "Georgie Peorgie",
            "Mary Lamb",
            "Penny Rice",
            publisher="O'Reilly School of Technology",
            author="Python Software Foundation"
            )
    description("Django 101",
            "Jacob Kaplan-Moss",
            "Baa-Baa Blacksheep",
            "Mary Contrary",
            "Missy Muffet",
            "Peter Piper",
            publisher="O'Reilly School of Technology",
            author="Django Software Foundation",
            editor="Daniel Greenfeld"
            )
```

☐ Save it in your **/python1** folder as **courses.py**, and run it:

```
cold1:~$ cd python1
cold1:~/python1$ ./courses.py
=====================================
Course Name: Python 101
Instructor: Steve Holden
-------------------------------------
Publisher : O'Reilly School of Technology
Author : Python Software Foundation
--------- registered students ----------
Georgie Peorgie
Mary Lamb
Penny Rice
=====================================
Course Name: Django 101
Instructor: Jacob Kaplan-Moss
-------------------------------------
Publisher : O'Reilly School of Technology
Editor : Daniel Greenfeld
Author : Django Software Foundation
--------- registered students ----------
Baa-Baa Blacksheep
Mary Contrary
Missy Muffet
Peter Piper
```

Let's take a closer look:

```python
#!/usr/local/bin/python3
def description(name, instructor, *students, **staff):
    """Print out a course description.
    name:          Name of the course
    instructor:    Name of the instructor
    *students, ...: List of student names (positional arguments)
    **staff, ...:   List of additional staff (keyword arguments)
    """
    print("=" * 40)
    print("Course Name:", name)
    print("Instructor:", instructor)
    print("-" * 40)
    for title, name in staff.items():
        print(title.capitalize(), ": ", name)
    print("{0:-^40}".format(" registered students "))
    for student in students:
        print(student)

if __name__ == "__main__":
    description("Python 101",
            "Steve Holden",
            "Georgie Peorgie",
            "Mary Lamb",
            "Penny Rice",
            publisher="O'Reilly School of Technology",
            author="Python Software Foundation"
            )
    description("Django 101",
            "Jacob Kaplan-Moss",
            "Baa-Baa Blacksheep",
            "Mary Contrary",
            "Missy Muffet",
            "Peter Piper",
            publisher="O'Reilly School of Technology",
            author="Django Software Foundation",
            editor="Daniel Greenfeld"
            )
```

The first and second parameters (**name** and **instructor**) are positional, and so are bound to the first and second arguments of any call. Any additional positional arguments are placed into the **students** tuple. Finally, any keyword arguments are placed into the **staff** dict.

The **name** and **instructor** parameters are printed out. The function then iterates over the items (each item is a (key, value) pair of the **staff** dict-parameter) to print details about any additional staff. Finally, the function loops through the **students** to list the individuals taking the class.

WARNING

Take care when using sequence- and dict-parameters. With regular (positional and keyword) parameters, you can usually determine the interface of the function (that is, how it should be called) from the function and parameter names. When sequence- and dict-parameters are used, this is more difficult to determine.

If you *do* use sequence- and dict-parameters, make sure you document the purpose of each parameter in the function's docstring. This is good practice in any case, but especially so when the interface is more complex.

Let's take a closer look at what our docstrings give us. Try these commands in an interactive session:

```
>>> import courses
>>> help(courses.description)
Help on function description in module courses:

description(name, instructor, *students, **staff)
    Print out a course description.
    name:              Name of the course
    instructor:        Name of the instructor
    *students:         List of student names (positional arguments)
    **staff:           List of additional staff (keyword arguments)

(END)
```

By documenting your function correctly, you've provided useful information to anyone who imports your module. (Your fellow programmers thank you!) Of course, the module itself can also have useful documentation, though in this case, there just wasn't much to provide. Continue your previous interactive session to verify that your documentation appears as expected:

```
>>> help(courses)
Help on module courses:

NAME
    courses

FUNCTIONS
    description(name, instructor, *students, **staff)
        Print out a course description.
        name:              Name of the course
        instructor:        Name of the instructor
        *students, ...:    List of student names (positional arguments)
        **staff, ...:      List of additional staff (keyword arguments)

FILE
    /users/smiller/python1/courses.py

(END)
```

Nice! The interpreter created a manual page for your module, just from the documentation strings that you entered. Now anyone who wants to use your module can import it into an interactive session and learn all about it using Python's standard **help()** function. I like it!

Importing Functions and help()

In the previous lesson, you learned about imports, including how to bring functions you've written into other programs. Now let's go over a handy trick that all Python developers love. First, we'll import the keyword_args.py module you wrote earlier in this lesson and run the built-in **help()** function over it. To get out of the help interface, just press **q**. Then type the commands below as shown:

```
>>> import keyword_args
>>> help(keyword_args)
Help on module keyword_args:

NAME
    keyword_args - Demonstrates capture of keyword arguments

FUNCTIONS
    keywords(**kwargs)
        Prints the keys and arguments passed through

    keywords_as_dict(**kwargs)
        Returns the keyword arguments as a dict

FILE
    /users/smiller/python1/keyword_args.py

(END)
```

So, thanks to the **help()** method, we can use the interactive interpreter to find important information about the functions we've written. These code statements are really driven by the docstrings you write into your Python code. All of the functions of a module are part of its documentation.

Sweet. If all of this isn't enough to make you start sprinkling doc strings around your code, then nothing will persuade you! You can document modules, functions, and classes just by making their first executable statement a documentation string. That's the kind of simple power that makes Python famous!

Function Execution by Dispatch

So far, when you've needed to control the flow of a program in Python, you've used the **if** statement to choose between two alternatives. But what if you need to select from multiple options? One way is to use **if**, **elif**, and **else**, but that can become unwieldy—especially when large numbers of alternatives are involved. If you had a hundred or a thousand lines of code between the **if** statements, the resulting program could likely be difficult to read, and even more difficult to maintain.

Thankfully, Python gives you a good way to work around this using tools you've already learned. You can write each alternative set of actions as a function, and then use a dictionary to define logic flow. The keys represent possible actions, and the functions are the actions themselves. This sounds a lot more complex than it actually is; let's use an example to clarify things:

```
>>> def add(a, b):
...     return a + b
...
>>> def sub(a, b):
...     return a - b
...
>>> sw = {'adder':add, 'subber':sub}
>>> sw['adder'](3,2)
5
>>> sw['subber'](3,2)
1
>>> sw
{'adder': <function add at 0x397588>, 'subber': <function sub at 0x397618>}
```

First we created the two simple functions, **add()** and **sub()**, then we placed them inside the **sw** dict. Then we called them (like any other Python dict) by referencing their keys, and passed in arguments. This provides a nice, clean way of organizing and calling functions. In the last two lines of the example, we printed out the logic flow. When a dict of functions is used this way, it is called a *dispatch table*.

Ready for a more complex example? Good! We'll put five functions into a dict, then use a **while** loop and an **input** statement to act as our user interface. We'll dispatch the appropriate function according to the user's input. A lot of this will look familiar to you. Let's go ahead and get it working. Create a new program as shown below:

CODE TO TYPE:

```python
#!/usr/local/bin/python3
""" A program designed to display switching in Python """

import sys

def print_text(text, *args, **kwargs):
    """Print just the text value"""
    print('text: ' + text)

def print_args(text, *args, **kwargs):
    """Print just the argument list"""
    print('args:')
    for i, arg in enumerate(args):
        print('{0}: {1}'.format(i, arg))

def print_kwargs(text, *args, **kwargs):
    """Print just the keyword arguments"""
    print('keyword args:')
    for k, v in kwargs.items():
        print('{0}: {1}'.format(k, v))

def print_all(text, *args, **kwargs):
    """Prints everything"""
    print_text(text, *args, **kwargs)
    print_args(text, *args, **kwargs)
    print_kwargs(text, *args, **kwargs)

def quit(text, *args, **kwargs):
    """Terminates the program."""
    print("Quitting the program")
    sys.exit()

if __name__ == "__main__":
    switch = {
        'text': print_text,
        'args': print_args,
        'kwargs': print_kwargs,
        'all': print_all,
        'quit': quit
    }

    options = switch.keys()
    prompt = 'Pick an option from the list ({0}): '.format(', '.join(options))
    while True:
        inp = input(prompt)
        option = switch.get(inp, None)
        if option:
            option('Python','is','fun',course="Python 101",publisher="O'Reilly")
            print('-' * 40)
        else:
            print('Please select a valid option!')
```

☐ Save it in your **/python1** folder as **switch.py**, and run it. Try all the different options. Also, try typing something that isn't one of the options. Before we start reviewing this program, take a minute and check out the difference between this program and earlier ones in the course. Doesn't this one just *look* cleaner?

Now, let's look at the functions:

```python
""" A program designed to display switching in Python """

import sys

def print_text(text, *args, **kwargs):
    """Print just the text value"""
    print('text: ' + text)

def print_args(text, *args, **kwargs):
    """Print just the argument list"""
    print('args:')
    for i, arg in enumerate(args):
        print('{0}: {1}'.format(i, arg))

def print_kwargs(text, *args, **kwargs):
    """Print just the keyword arguments"""
    print('keyword args:')
    for k, v in kwargs.items():
        print('{0}: {1}'.format(k, v))

def print_all(text, *args, **kwargs):
    """Prints everything"""
    print_text(text, *args, **kwargs)
    print_args(text, *args, **kwargs)
    print_kwargs(text, *args, **kwargs)

def quit(text, *args, **kwargs):
    """Terminates the program."""
    print("Quitting the program")
    sys.exit()

if __name__ == "__main__":
    switch = {
        'text': print_text,
        'args': print_args,
        'kwargs': print_kwargs,
        'all': print_all,
        'quit': quit
    }

    options = switch.keys()
    prompt = 'Pick an option from the list ({0}): '.format(', '.join(options))
    while True:
        inp = input(prompt)
        option = switch.get(inp, None)
        if option:
            option('Python','is','fun',course="Python 101",publisher="O'Reilly")
            print('-' * 40)
        else:
            print('Please select a valid option!')
```

All of the functions insist on the same arguments, even if most of them only use a portion of those arguments. The **first three functions** are clear enough; the **fourth function** just calls all three of them, and the **last function** uses the Python standard library **sys** module to quit the program.

Now, let's move on to everything that follows **if __name__ == "__main__":**. First, we create the **switch dict**, which has five elements—the values are each of the previously defined functions. Then, we construct an **options** list from the **switch.keys()**—keys of the switch dict. Then, we prompt the user for an option and start the input loop.

In the input loop, **option = switch.get(inp, None)** takes the user's option and either finds the function in question or returns a None object. If an option is found (**if option**), then the parameters are passed to the user-selected function. If no option is found, the user is prompted to **'Please select a valid option!'**.

The result is a cleaner application where reuse or integration of new functions is much easier. For example, let's add in the **description()** function from the **courses.py** module you wrote earlier in this lesson. Modify the code and the switch dict as shown:

```python
#!/usr/local/bin/python3
""" A program designed to display switching in Python """

import sys
import courses

def print_text(text, *args, **kwargs):
    """Print just the text value"""
    print('text: ' + text)

def print_args(text, *args, **kwargs):
    """Print just the argument list"""
    print('args:')
    for i, arg in enumerate(args):
        print('{0}: {1}'.format(i, arg))

def print_kwargs(text, *args, **kwargs):
    """Print just the keyword arguments"""
    print('keyword args:')
    for k, v in kwargs.items():
        print('{0}: {1}'.format(k, v))

def print_all(text, *args, **kwargs):
    """Prints everything"""
    print_text(text, *args, **kwargs)
    print_args(text, *args, **kwargs)
    print_kwargs(text, *args, **kwargs)

def quit(text, *args, **kwargs):
    """Terminates the program."""
    print("Quiting the program")
    sys.exit()

if __name__ == "__main__":
    switch = {
        'text': print_text,
        'args': print_args,
        'kwargs': print_kwargs,
        'all': print_all,
        'course': courses.description,
        'quit': quit
    }

    options = switch.keys()
    prompt = 'Pick an option from the list ({0}): '.format(', '.join(options))
    while True:
        inp = input(prompt)
        option = switch.get(inp, None)
        if option:
            option('Python','is','fun',course="Python 101",publisher="O'Reilly")
            print('-' * 40)
        else:
            print('Please select a valid option!')
```

☐ Save and run it. Choose the **course** option; your results may seem a little silly, but they are correct based on the argument being passed to the function—and the instructor *does* exist, and we think students *are* fun! See how easily we can integrate new functionality into our program? The logic doesn't change at all, only the data that drives it.

What's Your Function?

In this lesson, we reinforced what you already knew about functions and imports. You learned how to take code written inside of functions and use it in other places, and that documentation in docstrings can be really useful. You've reaped the benefits of splitting your own programs to make them more modular. And finally, you've seen how your earlier work could have been written more efficiently to benefit from this modular approach. In the next lesson, you'll learn

about Python's classes and object-oriented programming.

Keep in mind, as we push on, that good practice for Python developers means never repeating any stanza of code twice. Instead, put it into a function, and call the function twice!

Alright then, let's keep this train rolling!

Classes and Object-Oriented Programming

Lesson Objectives

When you complete this lesson, you will be able to:

- create the individual instances that behave according to their (built-in) type definition.
- define your own object classes.
- define object behavior.
- bind a name during the execution of the class body.
- define your own data types called *classes*.
- use __init__().

The Nature of Objects

In Python, they say "everything is an object." Let's explore this idea using lists as an example. Every list contains specific and likely different elements. But all lists have certain capabilities in common as well. You can append items to a list, retrieve individual elements by indexing, and so on. So objects have two distinct features. First, each object has its own unique data, private to that object and distinct from the other objects in its class. Second, each object is an instance of some *class* or *type*, which specifies how it can behave, or, in other words, which methods and operations can be used with that object.

The Python language contains some built-in data types. The interpreter knows how objects of a given type should behave, but of course, it has no idea which instances of which types your programs will create. So, the interpreter contains the *definitions* of the data types, but your program creates the individual instances, each of which behaves according to its (built-in) type definition.

In our first example here, we'll explore the nature of one of Python's objects: the complex number. Type the commands as shown:

```
>>>  c = 3+4j
>>> type(c)
<class 'complex'>
>>> dir(c)
['__abs__', '__add__', '__bool__', '__class__', '__delattr__', '__divmod__', '__doc__',
 '__eq__',
'__float__', '__floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs_
_', '__gt__',
'__hash__', '__init__', '__int__', '__le__', '__lt__', '__mod__', '__mul__', '__ne__',
'__neg__',
'__new__', '__pos__', '__pow__', '__radd__', '__rdivmod__', '__reduce__', '__reduce_ex_
_',
'__repr__', '__rfloordiv__', '__rmod__', '__rmul__', '__rpow__', '__rsub__', '__rtruedi
v__',
'__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '__truediv__', '
conjugate',
'imag', 'real']
>>> c
(3+4j)
>>> c.__add__
<method-wrapper '__add__' of complex object at 0x01A84110>
>>> c.real
3.0
>>> c.imag
4.0
>>> type(c.imag)
<class 'float'>
>>> c.imag = 2.5
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: readonly attribute
```

The interpreter reports that the *type* of **c** is **<class 'complex'>**. The call on **dir(c)** shows us that many methods have names that begin with double underscores— a lot of the names represent operators that you may want to use on a complex number (add, subtract, divide, and so on). Two of the names do not begin with double underscores: "real" and "imag." In mathematics, complex numbers have a real and an imaginary part. Each complex number in Python has two *attributes* called **real** and **imag**; those names are bound to floating-point numbers. You can access the value of each attribute separately, but because all numbers in Python are immutable, the interpreter won't allow you to change them.

Defining Your Own Object Classes

In keeping with a long-standing tradition in the object-oriented programming world, Python lets you define your own data types called *classes*. In Python we tend to reserve the word "type" to mean a class that is built into the interpreter, and "class" to mean those defined by the programmer. Python uses the compound **class** statement to introduce a class definition. The indented suite that follows the **class** statement contains descriptions of the various methods that should be available. The simplest suite is a single **pass** statement. Let's take a look. Type the commands below as shown:

```
>>> class First:
...     pass
...
>>> First
<class '__main__.First'>
>>> first = First()
>>> first
<__main__.First object at 0x02699A90>
>>> dir(first)
['__class__', '__delattr__', '__dict__', '__doc__', '__eq__', '__format__', '__ge__', '
__getattribute__',
'__gt__', '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__', '__new__'
, '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__'
, '__weakref__']
>>> first.name = "My first object"
>>> first.location = "Here"
>>> first.__dict__
{'name': 'My first object', 'location': 'Here'}
>>> first.name
'My first object'
>>> type(first.__dict__)
<class 'dict'>
```

We establish the type with a **class** statement. Then we create an *instance* of our new class named **first**, by calling **First()** as though it were a function. The interpreter identifies the instance by its name and address in hexadecimal (base 16): "<__main__.First object at 0x02699A90>". This instance of your class is equipped to behave in certain ways. The behaviors shown in the result of the **dir(first)** call are common to all Python objects.

Unlike built-in instances of classes, when using classes you create yourself, you can bind values to named attributes. These bindings work just like the binding of values to keys in a dict, because, in fact, they are dicts. Assignment to a dotted name results in that name being added as a key to a dict named **__dict__**, with the associated value becoming the dict value. For most names, **inst.name** is equivalent to **inst.__dict__["name"]**.

Class and Instance Namespaces

The **class** statement takes an indented suite as its body. When you bind a name during the execution of the class body, that class, like an instance, will have a namespace. Bindings in that class body are created within the class namespace. To fully understand that, run the following commands in an interactive session:

```
>>> class Second:
...     one = "Depp"
...     two = "Pitt"
...
>>> dir(Second)
['__class__', '__delattr__', '__dict__', '__doc__', ... ,
'__weakref__', 'one', 'two']
>>> Second.__dict__
dict_proxy({'__module__': '__main__', 'two': 'Pitt', 'one': 'Depp', '__dict__': <attrib
ute '__dict__' of '
Second' objects>, '__weakref__': <attribute '__weakref__' of 'Second' objects>, '__doc_
_': None})

>>> Second.__dict__.keys()
['__module__', 'two', 'one', '__dict__', '__weakref__', '__doc__']
>>> Second.__dict__["one"]
'Depp'
>>> Second.one
'Depp'
>>> second = Second()
>>> dir(second)
['__class__', '__delattr__', '__dict__', '__doc__', ...,
'__weakref__', 'one', 'two']
>>> second.__dict__
{}
>>> second.two
'Pitt'
>>> Second.two = "Clooney"
>>> second.two
'Clooney'
```

Don't close that interactive session; we'll be using it again shortly. Our example shows some subtle differences between classes and instances. Although each contains a **__dict__**, the instance was a dict already, and bindings to the instance are seen only in that dict. In the class, however, the bound names also appear as part of the class's namespace, and **__dict__** is no longer a dict, but something called a dict_proxy. A dict_proxy provides a selective view of the class's namespace. These differences are significant to a Python implementer, but for now you can file this information away.

More importantly, notice that the names **one** and **two** have been bound in the class namespace, and now also appear in the instance namespace (though not in its **__dict__**). In addition, they have the same value in the instance namespace as they do in the class namespace. If you rebind the name in the class namespace, it also changes in the instance. Our example demonstrates that names that *appear* to be in the instance namespace are *actually* defined in the class.

We'll take a closer look at the relationship between a class and its instances later. For the moment, just be aware that you can access attributes of the class in any of its instances. If you bind the same attribute to the instance, it does not change the class at all—the binding remains local to the instance. Continuing the interactive session, type the commands below as shown:

```
>>> second2 = Second()
>>> second2.one
'Depp'
>>> second2.one = "Bloom"
>>> Second.one = "DiCaprio"
>>> second.one
'DiCaprio'
>>> second2.one
'Bloom'
>>> dir(second2)
['__class__', '__delattr__', '__dict__', ..., 'two', 'one']
>>> second2.__dict__
{'one': 'Bloom'}
```

Here we created a second **Second** instance, named **second2**, which initially showed the same value as the class for its **one** attribute. When we assigned "Bloom" to the **second2** instance's **one** attribute, it overrides the class attribute, but only for that one instance. The **second** instance's **one** attribute still reflects the class's value for that attribute. When the **Second** class's **one** attribute is rebound, the **second** instance's **one** attribute also changes, but not that of the **second2** instance. (Phew! Did you catch all that?)

The attributes of a class can be accessed by all instances of that class but, as we've just seen, an assignment to an instance attribute of the same name will override the class attribute. Check out the last two expressions in the last session; not only does the **second2** instance have a **one** attribute (the one inherited from the **Second** class) in its namespace, it also has a **one** attribute in its **__dict__** as a result of being bound to **second2.one**. The interpreter is looking in an instance's **__dict__** first, and only looks in the namespace if it fails to find the attribute in the dict.

Defining Object Behavior

Hopefully it's starting to feel natural to you to write functions that operate on instances of classes. Now let's suppose Python didn't have complex numbers, and you had to implement them yourself. How would you create a new complex number, and how would you add two complex numbers together? You could define a class called **Cplx** (Python already uses **complex** for the existing complex data type), then write a **cplx()** function to create a complex number from the values of its real and imaginary parts. Then you could implement a **cadd()** function that takes two complex numbers and returns the sum of the two as its result. If you were feeling really ambitious, you could also write a **cstr** function to call from inside **print()** to output complex values.

The resulting code, with a couple of calls on the functions to test the code, might look like the program we'll create now. Create a new program as shown:

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:
    pass

def cplx(real, imag):
    c = Cplx()
    c.real = real
    c.imag = imag
    return c

def cadd(c1, c2):
    c = Cplx()
    c.real = c1.real+c2.real
    c.imag = c1.imag+c2.imag
    return c

def cstr(c):
    return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = cplx(0.0, 0.0)
    one = cplx(1.0, 0.0)
    i = cplx(0.0, 1.0)
    result = cadd(zero, cadd(one, i))
    print(cstr(result))
```

□ Save it in your **/python1** folder as **cplx.py**, and run it. The result **1.0+1.0j** prints. You aren't using very much of Python's class mechanism though. To do that, you need to separate the creation of the instances from their initialization. Then you'll rename the **cplx()** function to **cinit()**, and change its code so that it operates on an existing rather than a new instance, initialize it and return the instance. This initially complicates your calling code, because you now have to create the instances before initializing them, but don't worry about that now. Let's play with some code! Modify your program as shown:

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:
    pass

def cplx(real, imag):
    c = Cplx()
    c.real = real
    c.imag = imag
    return c

def cinit(c, real, imag):
    c.real = real
    c.imag = imag

def cadd(c1, c2):
    c = Cplx()
    c.real = c1.real+c2.real
    c.imag = c1.imag+c2.imag
    return c

def cstr(c):
    return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = Cplx()
    cinit(zero, 0.0, 0.0)
    one = Cplx()
    cinit(one, 1.0, 0.0)
    i = Cplx()
    cinit(i, 0.0, 1.0)
    result = cadd(zero, cadd(one, i))
    print(cstr(result))
```

□ Save and run it. It prints the same result as before—after all, it's really the same code.

Defining Behavior as Methods

So far, we've focused on the data attributes of classes and their instances. We know that when a class and one of its instances both have the same name, the instance attribute "wins." We can access a class's attributes via the instance, as long as the instance doesn't have its own attribute with the same name. But assignment is only one way to bind values to a class. Another way is through the **def** statement used to define functions.

Go ahead and edit **cplx.py** so that the functions become methods of the class:

> **Note**   To indent the function declarations, just select the block of code you want to indent and press **Tab**.

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:
    pass

    def cinit(c, real, imag):
        c.real = real
        c.imag = imag

    def cadd(c1, c2):
        c = Cplx()
        c.real = c1.real+c2.real
        c.imag = c1.imag+c2.imag
        return c

    def cstr(c):
        return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = Cplx()
    Cplx.cinit(zero, 0.0, 0.0)
    one = Cplx()
    Cplx.cinit(one, 1.0, 0.0)
    i = Cplx()
    Cplx.cinit(i, 0.0, 1.0)
    result = Cplx.cadd(zero, Cplx.cadd(one, i))
    print(Cplx.cstr(result))
```

□ Save and run it. You might see warnings on the **def** lines stating that the methods should have self as the first parameter, but you can ignore them for now. You'll still get this result: **1.0+1.0j**.

By declaring a function as part of the class body, we bind the function name within the *class* namespace rather than the module namespace. This means that, to call the function, it must be preceded by the class name and a dot. Because the class body is no longer empty, you don't need the **pass** statement any more.

Now let's break your code! Don't worry; we'll fix it right up once you understand the details of the breakage. The **Cplx** class has three new attributes—**cinit**, **cadd**, and **cstr**. You can access *class attributes* (attributes bound in the class namespace) through an instance of the class. So you'd think that you could access those methods through the instance, rather than the class. But when you change the code to do that, a strange error occurs. Modify **cplx.py** to call the methods on the instances as shown:

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:

    def cinit(c, real, imag):
        c.real = real
        c.imag = imag

    def cadd(c1, c2):
        c = Cplx()
        c.real = c1.real+c2.real
        c.imag = c1.imag+c2.imag
        return c

    def cstr(c):
        return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = Cplx()
    zero.cinit(zero, 0.0, 0.0)
    one = Cplx()
    one.cinit(one, 1.0, 0.0)
    i = Cplx()
    i.cinit(i, 0.0, 1.0)
    result = zero.cadd(zero, one.cadd(one, i))
    print(result.cstr(result))
```

Save and run it. You might be surprised to see a traceback and error message:

```
Traceback (most recent call last):
File "./cplx.py", line 21, in <module>
zero.cinit(zero, 0.0, 0.0)
TypeError: cinit() takes exactly 3 positional arguments (4 given)
```

This message may be a bit difficult to understand. It says that **zero.cinit(zero, 0.0, 0.0)** has four arguments, but it's clear that it provides only three. Where is the source of the fourth argument?

When the interpreter sees a reference to a class's method relative to an *instance*, it assumes that the method will need to know which instance it was being called upon. Consequently, it inserts the instance as the first argument automatically. Methods are being called with too many arguments because the interpreter *assumes* you will want a reference to the instance, and inserts it automatically. The fix for your code is to remove the explicit instance arguments. Fix cplx.py as shown:

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:

    def cinit(c, real, imag):
        c.real = real
        c.imag = imag

    def cadd(c1, c2):
        c = Cplx()
        c.real = c1.real+c2.real
        c.imag = c1.imag+c2.imag
        return c

    def cstr(c):
        return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = Cplx()
    zero.cinit(zero, 0.0, 0.0)
    one = Cplx()
    one.cinit(one, 1.0, 0.0)
    i = Cplx()
    i.cinit(i, 0.0, 1.0)
    result = zero.cadd(zero, one.cadd(one.cadd(i))
    print(result.cstr(result))
```

□ Save and run it. You should get **1.0+1.0j** as your result again.

Python Deep Magic: Hooking into Python's Class Mechanism

The code you have developed so far works, but it's a little on the ugly side. Separating the creation of objects from their initialization means that two lines of code are required to create a complex number. To remedy the ugly, we'll unleash some of Python's "deep magic!" (How cool is that?) Deep magic refers to programming techniques that are not widely known, and that may be kept secret deliberately. But you, my friend, are about to learn some Python deep magic to cast about yourself! Let's start with the special method Python provides to beautify your code: **__init__()**.

Using __init__()

When you create an instance of a class by calling it, the interpreter looks to see whether the class has an **__init__()** method. If it finds **__init__()**, it calls that method on the newly-created instance. Because it's an instance method call, the new instance is inserted as the first argument to the call. Further, if the call to the class has any arguments, they are passed to **__init__()** as additional arguments.

> **Note** The **__init__()** method **must not** return a value. If **__init__()** returns something, it affects the instance creation process. This causes the interpreter to raise an exception, and your program to fail. You'll learn about instance creation in more detail later.

By renaming the **Cplx** class's **cinit()** method to **__init__()**, you can shorten the code that creates and initializes the new instance to a single line. Very nice. Python users appreciate elegance and simplicity. Ugly Python code may be a sign that the language isn't being used to its full advantage. Let's try a bit more experimentation. Edit **cplx.py** below as shown:

```
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:

    def __init__(c, real, imag):
        c.real = real
        c.imag = imag

    def cadd(c1, c2):
        c = Cplx(c1.real+c2.real, c1.imag+c2.imag)
        return c

    def cstr(c):
        return "%s+%sj" % (c.real, c.imag)

if __name__ == "__main__":
    zero = Cplx(0.0, 0.0)
    one = Cplx(1.0, 0.0)
    i = Cplx(0.0, 1.0)
    result = zero.cadd(one.cadd(i))
    print(result.cstr())
```

☐ Save and run it. You'll get **1.0+1.0j** for a result yet again. Python objects tend to have a lot of those special methods with names that begin and end with double underscores. To make discussing them easier, "__init__()" is often pronounced "dunder-init"; "dunder" being an abbreviation for "double under." We'll convert the other methods of your complex class to "dunder" methods in a bit.

More on Python's Dunder Methods

When printing in Python, you get lots of help from the **print()** function. Without going into too much detail, the function converts each argument into a string by calling the object's **__str__()** method. So each class in Python can determine exactly how its instances get printed by defining a **__str__()** method. You can rename your **cstr()** method to **__str__()** and print **Cplx** instances directly.

Similarly, when you write **a + b** in Python, the interpreter tries to execute the task in a number of ways: first it tries to compute **a.__add__(b)** (which requires that **a** has a **__add__** method). If that doesn't work, Python tries to compute **b.__radd__(a)**. So, to enable your program to add **Cplx** objects, rename the **cadd** method to **__add__**.

Being Selfish

Let's take another quick peek at the first argument of your class's methods—the one that the interpreter puts in automatically when you call a method on an instance. Experienced Python programmers would be able to interpret the code in the last listing, but they would want to know why the argument was called **c** or **c1**.

There is an almost universal convention that the first argument of a method should be called **self**. Reading other people's programs is difficult enough, so it's important to stick to convention—not only will it make your code easier for other programmers to read, it will make it easier for *you* to read as well, and that's an important time saver.

So how should the code look when you make all the changes discussed in the last two sections? Edit **cplx.py** below as shown:

```python
#!/usr/local/bin/python3
"""Initial implementation of complex numbers."""

class Cplx:

    def __init__(self, real, imag):
        self.real = real
        self.imag = imag

    def __add__(self, c2):
        c = Cplx(self.real+c2.real, self.imag+c2.imag)
        return c

    def __str__(self):
        return "%s+%sj" % (self.real, self.imag)

if __name__ == "__main__":
    zero = Cplx(0.0, 0.0)
    one = Cplx(1.0, 0.0)
    i = Cplx(0.0, 1.0)
    result = zero + one + i
    print(result)
```

☐ Save and run it. You'll still get a result of **1.0+1.0j**.

A Solid Foundation

How does it feel to be an up-and-coming Python programmer? You've really come a long way! You've learned the basics of object-oriented programming in Python. The Python interpreter offers a lot of hooks in the form of **__xxx__()** methods that you can use to make your own classes as convenient and natural to work with as the built-in Python types.

In future lessons, you'll do lots more object-oriented programming; I'm confident you can handle it!

Exception Handling

Lesson Objectives

When you complete this lesson, you will be able to:

- handle exceptions.
- identify and locate exceptions.
- handle multiple exception types.
- handle multiple exceptions with one handler.
- flag error conditions from your own code.

Working through Exceptions

As you've worked through the lessons and tried out the code, you've seen plenty of Python error messages. Most of them have been *syntax errors*, which are caused by mistakes made while entering code examples. The rest, caused by all other kinds of mistakes, are called *exceptions*.

Syntax errors can and will crash your program, but soon you'll know exactly how to diagnose and fix them! Let's go right to work. Type the commands below in an interactive session as shown:

```
INTERACTIVE SESSION:

>>> print('Hello, world)
  File "<stdin>", line 1
    print('Hello, world)
                       ^
SyntaxError: EOL while scanning string literal
>>> print('Hello, world')
Hello, world
```

To handle this sort of error message, correct the syntax of your code so it makes sense to the interpreter. This is the most common kind of bug, and now you know how to squash it!

But even if your code is syntactically correct, it can still throw exceptions when you run it. The most common exceptions are TypeError, KeyError, and NameError. The odds are pretty good that you've encountered them already during this course, for instance if you mistyped a variable name or entered a non-numeric value that your program tried to convert into a number. Let's take a look at something like that. Type the commands below as shown:

```
INTERACTIVE SESSION:

>>> 'chapter ' + 15
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>> snakes = {'python':'fun','mamba':'dance'}
>>> snakes['cobra']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cobra'
>>> print(my_var)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'my_var' is not defined
```

Keep this interactive session open because we'll be doing another code example with the **snakes** dict.

The first lesson of exception handling is learning to catch exceptions, and then handle them so that they don't bring

your program crashing down. The next level of exception handling teaches you how to handle different types of exceptions at the same time.

How to Catch an Exception

As you saw in the interactive session above, the interpreter raises a KeyError exception when a dict does not contain a key specified as an index for retrieval. You catch errors using **try/except** statements like the one in our next example. Type the commands below as shown:

```
INTERACTIVE SESSION:

>>> try:
...     snakes['cobra']
... except KeyError:
...     print('Exception detected')
...
Exception detected
```

The **try** statement attempts to execute the code contained in its indented suite. That suite may be made up of several lines of code, but this example attempts to evaluate only the expression **snakes['cobra']**. (This key was chosen intentionally because it will raise an exception. We know you can handle it!). This causes the interpreter to trigger the exception handler for the KeyError exception, the **except** statement. The except suite contains the expression **print('Exception detected')**.

Congratulations—you caught an exception! Of course, the exception handler does nothing for you if you don't handle the correct exception. The next example illustrates this point. Type the code below as shown:

```
INTERACTIVE SESSION:

>>> try:
...     3/0
... except KeyError:
...     print("Exception detected")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
```

Although the **try** statement has an exception handler, it doesn't handle the actual exception (ZeroDivisionError) that is raised. In this case, the interpreter behaves as if there is no handler. In the interactive interpreter, this means you see a "stack traceback." If an unhandled exception happens when you are running a program, you still get the stack traceback, and then the program terminates.

Verifying Numeric Input

In earlier lessons, you used mathematical algorithms to learn about integers, loops, and functions. In some cases though, you ran into problems verifying numeric input. A good example is the sort of **input()** problem shown here. Type the commands below as shown:

```
INTERACTIVE SESSION:

>>> inp = input('Integer: ')
Integer: four
>>> 10 + int(inp)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'four'
```

Exception handling combined with a loop is really handy. You write an infinite loop, and break out of it when the user's input does not raise an exception. Let's try that problem again. Type the commands as shown:

```
>>> while True:
...     inp = input("Integer: ")
...     try:
...         print(int(inp)+10)
...         break
...     except ValueError:
...         print("Please enter an integer")
...
Integer: thing
Please enter an integer
Integer: python
Please enter an integer
Integer: 12.3
Please enter an integer
Integer: 12
22
>>>
```

This loop won't blow up if a user enters non-numeric data for a numeric field, which is a reasonably common pattern in Python coding.

Handling Multiple Exception Types

If you don't catch an exception, it will ultimately be raised to the interpreter. But if a **try** statement is contained inside another one, the outer **try**'s exception handler gets the chance to handle the exception. Create a new program in the editor window as shown:

```
#!/usr/local/bin/python3
""" Nested exception handling"""

def divide(a, b):
    """ Return result of dividing a by b """
    print("=" * 20)
    print("a: ", a, "/ b: ", b)
    try:
        try:
            return a/b
        except TypeError:
            print("Invalid types for division")
    except ZeroDivisionError:
        print("Divide by zero")

if __name__ == "__main__":
    print(divide(1, "string"))
    print(divide(2, 0))
    print(divide(123, 4))
```

☐ Save it in your **/python1** folder as **nested.py**, and run it:

```
cold1:~$ cd python1
cold1:~/python1$ ./nested.py
====================
a:  1 / b:  string
Invalid types for division
None
====================
a:  2 / b:  0
Divide by zero
None
====================
a:  123 / b:  4
30.75
```

The statement **print(divide(1, "string"))** raises a TypeError exception because it isn't possible to divide a number by a string. This exception is caught by the inner handler and handled. The function then ends without returning a value, so its result is **None**. The statement **print(divide(2, 0))** also raises an exception, but in this case it isn't caught by the **except** of the inner **try** because it isn't a TypeError. Consequently, the exception "bubbles up" to the next level, where there *is* a handler for the ZeroDivisionError that occurs. Finally, the statement **print(divide(123, 4))** represents a legal arithmetic operation and gets past both error handlers and returns the appropriate result.

By nesting exception handlers, you can catch errors that are thrown at different levels and handle them appropriately. Every additional level of nesting removes some readability from your program, though, so avoid doing it when you can. Fortunately, you can avoid some of that because Python allows you to attach several **except** clauses to a single **try** statement. Edit **nested.py** below as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
""" Nested exception handling"""

def divide(a, b):
    """ Return result of dividing a by b """
    print("=" * 20)
    print("a: ", a, "/ b: ", b)
    try:
        result = a/b
        print("Sometimes executed")
        return result
    except TypeError:
        print("Invalid types for division")
    except ZeroDivisionError:
        print("Divide by zero")

if __name__ == "__main__":
    print(divide(1, "string"))
    print(divide(2, 0))
    print(divide(123, 4))
```

☐ Save and run it. When the exception is raised inside of the **try** suite, the interpreter tries to match it against each of the **except** clauses, in turn. If it finds a matching clause, it executes the associated handler suite. If none of the **except** clauses match the exception, then none of the handlers are run, and the interpreter starts to examine the handlers of any outer **try** statements. The output from running this program should look like this:

```
cold1:~/python1$ ./nested.py
====================
a:  1 / b:  string
Invalid types for division
None
====================
a:  2 / b:  0
Divide by zero
None
====================
a:  123 / b:  4
Sometimes executed
30.75
```

The **print("Sometimes executed")** statement and the following **return** aren't executed when an exception is raised. One particularly useful feature of exceptions is that you can use them to change the flow of your program's logic when conditions are, well, exceptional.

Handling Multiple Exceptions with One Handler

Sometimes you want to take the same action for several different exceptions. You can do this by specifying the exceptions as a tuple after the **except** keyword. Then the handler will be executed if any of the exceptions in the tuple occur during execution of the **try** clause.

Raising Exceptions

You may want to be able to flag error conditions from your own code. This is especially useful when you are writing code to be used by other people. You flag error conditions with the **raise** statement; this is useful in two contexts:

- If you want to handle some of the consequences of an exception, but then re-raise it to be handled by some outer handler, you can do so by executing a statement consisting of only the keyword **raise**. This will cause the same exception to be presented to the outer handlers.

- If you detect some condition in your code that requires you to stop running the program, you can raise a specific exception of your choice by following the **raise** keyword with an exception. You can create that exception by calling any of the system exceptions with a string argument. Let's try out some of these features in **nested.py**. Type the code below as shown:

```python
#!/usr/local/bin/python3
""" Nested exception handling"""

def divide(a, b):
    """ Return result of dividing a by b """
    print("=" * 20)
    print("a: ", a, "/ b: ", b)
    try:
        return a/b
    except (ZeroDivisionError, TypeError):
        print("Something went wrong!")
        raise

if __name__ == "__main__":
    for arg1, arg2 in ((1, "string"), (2, 0), (123, 4)):
        try:
            print(divide(arg1, arg2))
        except Exception as msg:
            print("Problem: {0}".format(msg))
```

□ Save and run it. You should see this:

```
====================
a:  1 / b:  string
Something went wrong!
Problem: unsupported operand type(s) for /: 'int' and 'str'
====================
a:  2 / b:  0
Something went wrong!
Problem: int division or modulo by zero
====================
a:  123 / b:  4
30.75
```

The **except** statement in the divide() function now specifies the same handler for both ZeroDivisionError and TypeError exceptions. The handler prints an informative message ("Something went wrong!") and then re-raises the same exception. Since there are no further handlers in the function, the re-raised exception is now caught by the **except** statement in the main program.

In this case, the **except** statement catches pretty much any exception, because all exceptions are direct or indirect subclasses of **Exception**. Also, the exception specification can be followed by an **as** clause, which specifies a name to bind to the exception that is being handled. You can see from the **print()** function call that when an exception is converted to a string, you get the message associated with the exception.

Specific and Generic Exceptions

Using specific exceptions is handy because doing that allows you to hone in on the exact exception you want to handle. But what happens when you have code where exceptions might be raised in places you can't anticipate? Python will allow you to omit the exception specification altogether. This clause must follow all **except** clauses with exception specifications, and will catch any exception whatsoever. The next example uses both specific and generic specifications to catch exceptions from a Test class that possesses an **add()** method specifically included to produce an AttributeError or TypeError. Create a new program as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
""" Named and generic exception handling"""

def add(a, b):
    """ Print the results of adding a set and a value"""
    try:
        a.add(b)
        print(a)
    except AttributeError:
        print("({0}) is not a set object".format(a))
    except TypeError:
        print("({0}) is not a hashable object".format(b))
    except:
        print("This is a generic exception")

class Test(object):
    """ Just a simple test class """

    def add(self, a):
        """ Demonstrates how you need to be able to handle unpredictable results
. """
        d = {'python':'fun'}
        return d[a]

if __name__ == "__main__":
    s = set((1,2,3))
    add(s, 4)
    add(1, 4)
    add(s, [4, 5, 6])
    t = Test()
    add(t, 1)
```

□ Save it in your **/python1** folder as **exceptions.py**, and run it. In our **add()** function, we plan for 'a' to throw either an AttributeError, TypeError, or something we can't predict. Remember, the plain **except** clause must follow the named exceptions. In the last two lines, we attempt to use the **add()** method of the **Test** instance to use the supplied parameter as an index to a dict with only one key. Consequently, the final call to **add()** raises a KeyError exception, which in turn causes the final **except** clause to be activated, because the exception raised is neither an AttributeError nor a TypeError.

After you run the program, follow the logic through to make sure you understand exactly why it behaves the way it does. If there's anything you don't understand about all of this, talk it over with your instructor.

When to Use Exceptions

Some Python objects are equipped with methods that remove the need for exceptions. The dict is a good example of this, as you'll see in the next code sample—the **dict.get()** method tries to use the first argument as a key into the dict, but if no such key exists, it returns the second argument. Type the commands below as shown:

```
INTERACTIVE SESSION:

>>> d = {1:'python'}
>>> d[1]
'python'
>>> try:
...     d[10]
... except KeyError:
...     print('no snake here')
...
no snake here
>>> d.get(10,'no snake here')
'no snake here'
```

Of course, **dict.get()** only works if you know that **d** is of type dict. If you don't know that, you might want to handle specific exceptions raised under those circumstances. For example, you might try this:

```
INTERACTIVE SESSION:

>>> d = [1,2,3,4]
>>> try:
...     d[10]
... except KeyError:
...     print('no snake here')
... except IndexError:
...     print('no snake here either')
...
no snake here either
```

Exceptional Work So Far!

You need to be able to anticipate when things might go wrong with your programs, so you can catch and handle the exceptions that are raised. This will make your programs more robust, and capable of handling anything that users can throw at them.

Ideally, you build programs that never terminate with an uncaught exception. With your new knowledge of exception handling, you are much closer to reaching that goal.

You're almost there, just one lesson to go before your final project! Great work so far, keep it up!

Building and Debugging Whole Programs

Lesson Objectives

When you complete this lesson, you will be able to:

- build and debug an entire program.
- document and test Python code.
- code in Python.

Putting it All Together

Now that you've explored the core elements of Python, you know enough to write some pretty complex programs. But there's still lots left to learn. In this lesson, we'll go over a few more concepts that you can use to complete your final project. We'll decipher the more advanced mysteries of Python in future courses.

For our last lesson together in this course, we'll gather a bit more information about how to put programs together.

You'll want to know about testing and debugging (locating and fixing problems in your code). These topics are kind of like icebergs: much of their substance lies under the surface and will not reveal itself immediately; you'll continue learning for as long as you work with Python. In fact, I have been a practicing programmer myself for over forty years, and I continue to learn more and more about testing and debugging.

The Art of Computer Programming

The title of this subsection is also the title of a book series being written by Donald Knuth, a computer science professor at Stanford University. Though not yet finished, it already fills four massive volumes. Don't be discouraged if you feel like you don't have a handle on all there is to know about Python yet! If you're going to become really good at this, your programming education will be ongoing.

As your grasp of the language increases, you'll inevitably find that when you review code you wrote some time ago, you'll have found better ways to express the same algorithms. It's good practice to reevaluate your code occasionally—even if it works, it can probably be improved. On the flipside, as they say, "if it ain't broke, don't fix it." Unless there's a benefit to changing the code (like increased speed or reduced complexity), then leave it alone.

It's all too easy to introduce subtle errors into your programs by changing code to other code you think is equivalent. Until you're a bit more experienced and confident in your decisions, just be satisfied with programs that work.

Design Techniques

Two common terms flung about by programmers are *top-down* and *bottom-up* design. In top-down design, you defer thinking about the detail of a problem until you have mapped out the overall structure it will have. Working bottom-up, you begin by building a set of primitive operations that you can then fold together with *glue logic* to solve your problem.

The top-down approach lets you avoid having too much confusing detail to deal with early in the design cycle. Good top-down design focuses first on the program's large-scale *architectural* features.

The bottom-up approach is useful when you already understand your data and the ways you need to manipulate it. Using a **test-driven development** approach to programming, you write tests first, and then write your program to pass the tests. Each function and method is written to pass its tests, so you know that your lower-level components do indeed behave as expected.

The top-down and the bottom-up approaches can also be used together on the same project. It's a little like two teams boring a tunnel from opposite sides of a mountain: if the two do not meet, they have not been working harmoniously together.

By taking a top-down approach initially, you can operate a divide-and-conquer scheme, and avoid being overwhelmed by detail early in the design. If your coding problem isn't too complex, you might find that you have already solved it before you ever start working bottom-up.

Agile Programming

Agile programming techniques focus on delivering the simplest code that meets the requirements, or as agile practitioners often say, "the simplest thing that could possibly work." Agile methods place great emphasis on *refactoring* your code when it becomes too complex. Refactoring means changing the way your program is organized without changing its behavior. Refactoring is generally used when handling large programs, but it can be helpful whenever complexity starts to overwhelm you. Refactoring can help you to:

- **Remove duplicate code:** When two different functions provide the same result, or one function is a special case of another, we refactor the two functions into one, and we'll have less code to maintain.
- **Isolate existing logic from a needed change:** If you have to change certain cases currently handled by a single class, you might find it advantageous to refactor the class by turning it into two subclasses of a common base class. The changed behavior can then be implemented in just one of the subclasses.
- **Make the program run faster:** When performance becomes sluggish, it may be that your original choice of algorithm or data structure was inappropriate, so you refactor to streamline your process.

Some aspects of agile development are meant to be used by teams of software developers rather than individuals. Let's go over a few key principles that apply to most agile technologies:

- **Design and code are test-driven:** Whenever you add functionality to your program, you first write a test, for automatic execution, that checks to make sure that the functionality is present and performs properly. Your work should proceed in small increments—never add two features at the same time.
- **Integrate continuously:** Each time you change or fix a module, after running its tests, integrate the module back into the system and run the system tests to make sure that your change has not had any unintended consequences.
- **Refactor mercilessly:** To **refactor mercilessly** means that if tasks are performed similarly in two places, move them around so they're done in one place instead, and then called or inherited by the two original places. If you have coding standards and they are violated, fix them. If you notice structural defects, fix them. After each change, rerun all of your tests to verify that your code has not been broken during the refactoring process.
- **Release early and often:** Release your program to the users before adding too many features. You can use their feedback to guide further development, and deliver the most important functions of your program faster.
- **Keep it simple:** Don't include complexity that you think might be handy later. Simplicity has many benefits, and often "later" never arrives.
- **Code is not *owned*:** Agile programming is a team effort, so it is never "Joe's code" or "Jim's code;" it's "our code." Never fear changing code created by someone else—it's yours to use and testing will help you make sure you don't break it.

Documenting and Testing Python Code

Python comes with two testing frameworks built-in. If you have been using the **JUnit** testing framework, consider using the **unittest** module, which is based on JUnit. You'll probably find the **doctest** module easier to use, because it works by embedding executable Python statements and their expected outcomes into the docstrings that are embedded into all Python code.

Because the docstrings are available to the program, testing framework can use information embedded in them to verify that code is functioning correctly.

To see how **doctest** works, create a new program as shown:

```
#!/usr/local/bin/python3
"""Demonstrates the doctest module in action."""

def square(x):
    '''Returns the square of a numeric argument.

    >>> square(3)
    9
    >>> square(1000)
    1000000
    >>> square("x")
    Traceback (most recent call last):
    ...
    TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
    '''
    return x*2

def _test():
    import doctest, testable
    return doctest.testmod(testable)

if __name__ == "__main__":
    _test()
```

Save it in your **/python1** folder as **testable.py**, and run it. This program contains a bug: instead of returning its argument raised to the second power (squared), the **squared()** function returns its argument multiplied by two. This is an easy mistake to make—we only left out a single asterisk—but it renders the function incorrect. Our output looks like this:

```
**********************************************************************
File "/users/smiller/python1/testable.py", line 7, in testable.square
Failed example:
square(3)
Expected:
9
Got:
6
**********************************************************************
File "/users/smiller/python1/testable.py", line 9, in testable.square
Failed example:
square(1000)
Expected:
1000000
Got:
2000
**********************************************************************
File "/users/smiller/python1/testable.py", line 11, in testable.square
Failed example:
square("x")
Expected:
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
Got:
'xx'
**********************************************************************
1 items had failures:
3 of 3 in testable.square
***Test Failed*** 3 failures.
```

When you run the program, it calls the **_test()** function, which in turn imports the **doctest** module. It also imports the program itself, and then finally calls the **doctest.testmod()** function with the module as an argument. This causes the examples in the **square()** function's docstring to be run, and compared with the

output listed under each expression.

Because the results don't agree with the predictions in the docstring, the differences are reported as errors, and the output makes it clear that something is wrong with the program.

Let's fix the error by changing the operation in the **square()** function to an *exponentiation* (feel free to toss the word *exponentiation* into conversation as well, to impress your friends), as shown:

CODE TO TYPE:

```
#!/usr/local/bin/python3
"""Demonstrates the doctest module in action."""

def square(x):
    '''Returns the effective length of a string
    allowing for tabs of a given length tlen.

    >>> square(3)
    9
    >>> square(1000)
    1000000
    >>> square("x")
    Traceback (most recent call last):
    ...
    TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
    '''
    return x**2

def _test():
    import doctest, testable
    return doctest.testmod(testable)

if __name__ == "__main__":
    _test()
```

☐ Save and run it. You get no output. That's good. The doctest system is designed to help you to detect when your code is working incorrectly and to hone in on the tests that are failing. You'll learn more about testing in other courses, but for now, doctest is a great place to start. Your doctests can be integrated into other schemes as you move forward.

'Keep It Simple, Stupid' (KISS)

The **KISS** principle (albeit a tad harsh) is one that programmers find helpful. Of course, when you're first learning a language, sometimes nothing seems simple. Breaking up our operations into smaller pieces helps us understand the big picture. We can see, then, that every program is made up of a sequence of operations. Each operation is either a basic statement, or a choice between several alternatives, or a loop. When the user makes a choice, the action to be taken is a sequence of operations—and each operation can be a basic statement, or a choice between several alternatives, or a loop.

Refactoring

The concept of refactoring code can be compared to the editions of a textbook over time. The first edition provides the main body of text, while in following editions, editors clean up mistakes, make style changes, or add more information. The core content of the book doesn't really change, but the details get better.

When you refactor, you aren't adding new functionality, you are making the code better. You exchange duplicate code for calls and inheritance where possible, fix structural defects, change code to match coding standards (if you have them), and most importantly, *make sure that it passes all of your tests*.

If you are going to refactor your code mercilessly, *you must have tests*. Without sufficient testing, you cannot be certain that your changes have not broken your program.

Let's take some code and refactor it mercilessly. It isn't often we strive to be merciless, so let's enjoy this rare opportunity! In our sample program, we have some code that is truly miserable to look at, but it works. Create the file shown below:

```python
#!/usr/local/bin/python3
"""Demonstrates an opportunity for refactoring."""

def list_multiply(LIST_A, LIST_B):
    """ Sums two lists of integers and multiplies them together

    >>> list_multiply([3,4],[3,4])
    49
    >>> list_multiply([1,2,3,4],[10,20])
    300
    """

    TOTAL_A = 0
    for i in LIST_A:
        TOTAL_A += i
    TOTAL_B = 0
    counter = 0
    while True:
        if counter > len(LIST_B) - 1:
            break
        TOTAL_B = TOTAL_B + LIST_B[counter]
        counter += 1
    return TOTAL_A * TOTAL_B

def _test():
    import doctest, refactor
    return doctest.testmod(refactor)

if __name__ == "__main__":
    _test()
```

□ Save it in your **/python1** folder as **refactor.py** and run it. You should get no errors, but if this code makes you wince, then you are on track to become a good Python programmer! While the code is *technically* correct, it just plain smells. Some variables are upper-case and some are lower-case. Two different loops are used to do the same action of summing up the integers in two lists, when a simple built-in **sum()** function would suffice. Can you imagine making the necessary alterations if you had to add the capability to handle a third or fourth list to your code? Ouch.

Fortunately the code comes with doctests, so you can do some merciless refactoring. Edit the program as shown:

```python
#!/usr/local/bin/python3
"""Refactored version of previous example."""

def list_multiply(a, b):
    """ Sums two lists of integers and multiplies them together

    >>> list_multiply([3,4],[3,4])
    49
    >>> list_multiply([1,2,3,4],[10,20])
    300
    """

    return sum(a) * sum(b)

def _test():
    import doctest, refactor
    return doctest.testmod(refactor)

if __name__ == "__main__":
    _test()
```

Huge difference! □ Save and run it again; the doctest should still work. Refactoring like this allows you to make

changes to improve your code without the fear of breaking it.

And if you need to add functionality, refactored code makes it that much easier. Because the code is generally simpler (always remember KISS), it will be less difficult to extend it to work with any number of lists of integers. Try this version of the code, and make sure it passes the tests:

CODE TO TYPE:

```python
#!/usr/local/bin/python3
"""Adding functionality, much easier with refactored code!"""

def list_multiply(*lists):
    """ Sums any number of lists of integers and multiplies them together

    >>> list_multiply([3,4],[3,4])
    49
    >>> list_multiply([1,2,3,4],[10,20])
    300
    >>> list_multiply([4,3,2,1],[50,50],[5,5,5])
    15000
    """

    total = 1
    for l in lists:
        total *= sum(l)

    return total

def _test():
    import doctest, refactor
    return doctest.testmod(refactor)

if __name__ == "__main__":
    _test()
```

Go Forth and Code in Python!

☐ Save and run it again; the doctest should still work. Wow. Remember when you were a total Python newbie? You've come a long way since Lesson 1! Now you know almost all of Python's syntax, and you're familiar with the statements that make up the language. You know how to structure programs as sets of functions, and how to deliver functions in modules that can be re-used by several different programs.

You still don't know all there is to know about Python (who does?), but now you're in position to understand much of the Python code you encounter. Read lots of Python code; it's a great way to learn more about the language and to increase your understanding of the library and third-party modules it uses. You can practice doing just that and applying the Python tools you've acquired here, in your final project. Thanks so much for taking this Python journey with us, it's been a real pleasure working on it with you. Good luck!