

# Chapter 3, An Introduction to Python

John M. Morrison

March 11, 2016

## Contents

<b>0</b>	<b>Introduction</b>	<b>2</b>
<b>1</b>	<b>What is coming up?</b>	<b>3</b>
1.1	The Nitty-Gritty: Launching Python in Linux or MacOSX . . . .	3
1.2	Using Python in Windoze . . . . .	4
1.3	All Users . . . . .	4
<b>2</b>	<b>Types, Objects and Numbers</b>	<b>5</b>
2.1	Python's Number Types . . . . .	6
2.2	Python's String Type . . . . .	7
2.3	Getting More Information about Strings and Built-in Types . . .	8
2.4	Python's Boolean Type . . . . .	9
2.5	Casting . . . . .	10
2.6	Relational Operators and the Boolean Type . . . . .	11
<b>3</b>	<b>Sequence Types</b>	<b>12</b>
3.1	On the Importance of Type . . . . .	15
<b>4</b>	<b>Making your first Python Program hello.py</b>	<b>17</b>
4.1	Creating a Program in IDLE . . . . .	18
4.2	A Comparison with Some Other Languages . . . . .	18
4.3	Running Your Program . . . . .	19

<b>5</b>	<b>Comments in Python and on Python</b>	<b>20</b>
<b>6</b>	<b>Taking Full Advantage of print</b>	<b>22</b>
<b>7</b>	<b>Variables, Assignment, Operators and Type</b>	<b>25</b>
7.1	Rules for Variable Names . . . . .	28
7.2	Language Keywords . . . . .	29
<b>8</b>	<b>Expressions and the Symbol Table</b>	<b>29</b>
8.1	The Inside Dope on Assignment . . . . .	32
8.2	A Shorthand Convenience: Compound Assignment Operators . .	33
8.3	Python is a strongly, dynamically typed language. . . . .	33
8.4	Python is a Garbage-Collected Language, or Charles Dickens Wrote of Orphan Heaven . . . . .	34
<b>9</b>	<b>Sequence Operations</b>	<b>36</b>
9.1	Indexing . . . . .	36
9.2	Slicing . . . . .	37
9.3	The in Keyword . . . . .	38
<b>10</b>	<b>Advanced Topic: Understanding the Pointing Relationship</b>	<b>40</b>
<b>11</b>	<b>Mutability and its Dangers</b>	<b>42</b>
<b>12</b>	<b>Advanced Topic: Pooling</b>	<b>46</b>
<b>13</b>	<b>Useful Learning Resources</b>	<b>48</b>

## 0 Introduction

Now we will begin to learn about a *programming language* called Python. Python allows us to teach the computer how to do chores we want it to do. We must learn about the grammar and structure of the language to use it correctly. Happily, you can use Python in an interactive mode (or shell) and “talk” to it directly.

The Python site <http://www.python.org> has an abundance of useful infor-

mation. Python is available for Mac, all flavors of UNIX, and 'Doze on this site. You can program locally on your own box or use a UNIX server. There are complete instructions on the site for installing and using Python on any platform. We will emphasize using Python in a UNIX environment in this book.

Python is available in two versions, currently 2.7 and 3.2. There are important differences between the two versions. We will primarily use Python 3 this book, but will point out features of Python 2 as we go along, so you may use either version. However, it is best to write new code in Python 3

## 1 What is coming up?

The purpose of this chapter is to introduce you to *objects*, which are regions of memory representing data. Objects come in various flavors called *data types*, or just types for short.. You will become familiar with several of the basic types and how to work with them. You will learn how to use Python interactively as well as how to write simple programs in Python.

Happily Python can be run on just about any computer. We will show you how to run it on all systems. The next section discusses Python in a UNIX envitnment; tis covers both Linux and MacOSX. The next section discusses installing and running Python in Windoze. If you are running Windoze, go back to the section on running it in UNIX and try the example shown.

### 1.1 The Nitty-Gritty: Launching Python in Linux or MacOSX

To begin an interactive Python session, type

```
$ python3
```

at the UNIX prompt and hit enter. You will see something like this

```
$ python3
Python 3.2 (r32:88452, Feb 20 2011, 11:12:31)
[GCC 4.2.1 (Apple Inc. build 5664)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> is the Python prompt. It indicates that Python is ready to do work. Python can work as a calculator. Try typing in some expressions and having them evaluated. Here is a sample session. Replicate it then do some experiments on your own.

```

>>> 2 + 3          #addition
5
>>> 4*5            #multiplication
20
>>> 2**4           #exponentiation
16
>>> 33/6           #division
5.5
>>> 33//6          #integer division
5
>>> 33 % 6         #mod, or remainder, operator
3
>>>

```

To quit, type control-d or `quit()`. The character control-d is the end-of-file character in UNIX. If Python gets “stuck”, you can type control-C (hang up) to terminate its present task. This should bring you a fresh Python prompt, although Python may grumble. If the prompt is showing, Python has no present task and is ready to receive commands..

## 1.2 Using Python in Windoze

To install Python, obtain the `.msi` file from the Python website. Download it onto your computer and then double-click on it to launch its install shield. Click through the boxes to install. Once you are done, click on the Start menu and you should see Python in your menus. You should install Python 3, since this book is Python 3-centric.

Select Start → Python → IDLE. IDLE is a graphical editor for Python. It will show you an interactive prompt, and you can use that for the examples here, just as a UNIX user does. You can obtain useful information about IDLE at this link, <http://docs.python.org/library/idle.html>. This freely-available environment runs on all three major platforms. In Linux, you will have to download it from your distribution’s repositories using your package manager.

You can also get Python working in PowerShell (You use this, don’t you?) by consulting the instructions posted at <http://learnpythonthehardway.org/book/>.

## 1.3 All Users

You can find some very useful information at <http://learnpythonthehardway.org/book>. Learning Python the Hard Way is an excellent accompaniment to this book. It is loaded with great exercises.

## 2 Types, Objects and Numbers

Computing is about the manipulation of data; all data in Python are represented by *objects*. Objects are regions of storage in memory. This most basic information about a Python object is its *data type* or *type*. Every Python object knows its type. All objects are stored in an area of memory called *the heap*. You can think of the heap as a contiguous chunk of RAM that serves as a warehouse for the data you are working with..

Three very basic types in Python are `int`, which represents an integer (whole number), `bool`, which holds a value of `True` or `False` and `str`, which represents a *character string*, which is simply a glob of text. Hence, Python is able to store integers, Booleans and text in memory.

**Python 2 Notes** The type `int` only represents 32 bit integers in two's complement notation. Python 2 prevents type overflows by automatically promoting calculations involving these into the type `long`, which works just like Python 3's `int`.

The division operator `/` in Python 2 by default performs integer division. Take note of the following

```
>>> 33/6           integer division is the default
5
>>> 33.0/6         decimal point triggers floating point arithmetic
5.5
>>> 33//6          the // operator works in Python 2
5
>>>
```

The moral of the story: Use `//` to make your intent to do integer division explicit. If you adhere to this convention, you will have no integer division problems when using Python 2 or 3.

We now return to our main thread. A computational object has three important properties: state, identity and behavior.

- The identity of an object is its most basic property: It is what an object *is*. Identity refers to an objects physical presence in memory.
- The *state* of an object refers to the values the object is holding. This is what an object *knows*. For example, the state of an integer is simply the integer the integer object is storing. The state of a string is the character sequence in its glob of text.
- Objects have *behavior* this is what an object can *do*. For instance, the number types we will meet very soon exhibit the expected useful behavior

in the presence of arithmetic operators; you saw this happening in the first sample Python session we created. Strings have the ability to do such things as creating a copy of themselves with all alpha characters in caps or all alpha characters in lower-case. We will demonstrate this after exploring Python's number types.

## 2.1 Python's Number Types

To get started, we begin with Python's number types. We will see that objects of number type are aware of arithmetic operations, and that, like all other Python objects, they know their type.

- **int** This is the integer type; its state is simply the integer being stored. The integer type deals with whole numbers. In Python 2, division of integers is integer division; in contrast Python 3 automatically interprets division as floating point by default; use `//` to trigger integer division in any version of Python. We encourage you to use `//` in any Python program; then your Python 2 programs will not break when you bring them into Python 3. On most modern machines, integers are 32 bit integers stored in two's complement notation.
- **float** This is the *floating point* type; these are decimal numbers. They may be output with scientific notation. The expression `3.55e5` means  $3.55 * 10^5$  or 355000. Floating point numbers in Python are IEEE 754 double-precision floating point numbers. This is the standard used for floating point numbers in almost all modern programming languages.
- **long** This is an extended-precision integer. Normally integers are restricted to a range from  $-2^{32}$  to  $2^{32} - 1$ . The **long** type is only limited by memory. Try it out by computing something like `2**10000`. If an integer calculation becomes too large, it is seamlessly promoted to a calculation of **long** type. Python 3 merges the **int** and **long** types.

Python's number objects share several features. In each case, the state of the number object is just the number it is storing. Numbers have arithmetic operators as behaviors, and they know their types.

To learn the type of a Python object, just use the **type** function as shown here. We do this on the three number types here.

```
>>> type(5)
<class 'int'>
>>> type(141213221414122342141)
<class 'int'>          #you will see <type 'long'> in Python 2
>>> type(1.414)
<class 'float'>
```

You should create an interactive session and use the `type` function on a string.

The number types are equipped with a collection of operators. We shall establish a little terminology here. A *binary operator* is an operator that takes two *operands*. For example `+` is a binary operator for any number type. A binary operator takes two objects, and produces a third object. For example, the result of `2 + 2` is 4. The standard operators `+`, `-`, `*`, `/`, `//` and `%` are all binary operators.

We will say these arithmetic operators are *infix* operators because they occur between their operands. There are **prefix** operators that occur before their operands. The operator `-` that changes the sign of a number is an example of a prefix unary (one operand) operator. Finally there are postfix operators, which occur after their operand(s); we will meet some of these later.

**Programming Exercises** In this set of exercises, you will use Python to do some scientific unit conversions. This will get you used to using the interactive prompt and number calculations. If you are using Python 2, Be careful of any integer divisions that could occur. Be reassured: You may use parentheses to override the default order of operations. Also, the order of operations you know and love from Algebra I works just fine.

1. Determine the number of cubic feet of water in a cubic mile of water.
2. If a cubic foot of water weighs 62.4 lbs, figure out the weight of a cubic mile of water in tons.
3. The earth weighs approximately  $6.58 \times 10^{21}$  tons. Assuming the earth is spherical and it has a radius of 3960 mi, determine the average density of the planet in pounds per cubic foot. You will need to look up the formula for the volume of a ball; you may approximate  $\pi$  with 3.14.
4. Find the surface area of the earth in square miles. Determine the equivalent in acres.

## 2.2 Python's String Type

Python string objects hold globs of text. A glob of text can be enclosed in single or double quotes. You must use the same type of quote on both sides. We demonstrate this here.

```
>>> "hello"
'hello'
>>> 'hello'
'hello'
>>> "hello"
File "<stdin>", line 1
```

```

    "hello"
    ^
SyntaxError: EOL while scanning string literal
>>>

```

Note the punishment dished out by Python when you place a single quote on one side and a double-quote on the other.

You can *concatenate*, or glue together, strings using the `+` operator. You can obtain the length of a string using the built-in `len` function. Examples are shown here.

```

>>> "hello" + " there"
'hello there'
>>> len("hello")
5

```

**Exercises** In this next set of exercises, you will get a preview of string behaviors. Making a string in Python is simple; you just enclose text in double-quotes or single-quotes. Enter each item at the command line. What happens in each case? Create your own strings and experiment.

1. `"abcABC123".upper()`
2. `"abcABC123".lower()`
3. `"abcABC123".capitalize()`
4. `len("abcABC123")`

## 2.3 Getting More Information about Strings and Built-in Types

Let us make a first visit to the Python documentation. The Python site contains a wealth of information that you can begin to explore and use. It also offers lots of nice examples for you to try in interactive mode. We will visit it for the purpose of learning about strings.

Begin by going to <http://www.python.org>. Click on the documentation link and select the Python 3.x documentation.

You will want to explore this site for tutorials and other information. Now visit <https://docs.python.org/3/library/stdtypes.html>; this page gives a lot of detail on the built-in types. Now go down to Section 4.7.1, String Methods.

It's time to open an interactive session and for you to experiment. Go through the exercises shown here and get a guided tour of some very useful goodies. Experiment with all of these and a few more. Python strings are smart and they can do a whole lot of work for you.



## Programming Exercises

1. Make a string named `s` and initialize it with mixed cases. Now use `s.capitalize()`. What happens? What if the string's first character is a number? a space?
2. Now let us try the `endswith` method. You will notice that the documentation presents it in this form.

```
endswith(suffix[, start[, end]])
```

A suffix such as `".html"` is required. You can see if a string ends with a given suffix. Can you figure out what is happening here?

```
>>> x.endswith("bun", 3, len(x)-2)
True
>>> x.endswith("bun", 0, len(x)-2)
True
```

What role do the last two (optional) arguments play?

3. If there is an `endswith`, there is a `startswith`. Experiment with it.
4. What is happening here?

```
>>> food = "pizza"
>>> food.find("z")
2
>>> food.rfind("z")
3
>>>
```

5. Make the string

```
>>> x = "          I am very spacy....          "
```

6. What do `rstrip()`, `lstrip` and `strip` do to it? Why might this be useful for getting stuff from a user who types in response to an `input` statement?
7. Try some of these out.

```
isspace()
isdecimal()
islower()
isalpha()
```

What do all of the methods of form `isSomething` have in common?

## 2.4 Python's Boolean Type

A *Boolean* value is a truth-value with the possible values of `True` or `False`. The tokens `True` and `False` are valid Python constants. The exercises shown here will take you on a guided tour of this type.

## Exercises

1. At the Python prompt enter `type(True)` and see the Python name for the Boolean type.
2. At the Python prompt enter `not True` and `not False`. What happens?
3. If `b` represents a boolean value, what is the relationship between `b` and `not not b`?
4. Since `not` is an operator, would you describe it as prefix, postfix or infix? Would you describe it as binary or unary?
5. There is a binary infix operator `and` for Booleans. Enter all four possible combinations of `True` and `False` with the operator `and` in between them. If `a` and `b` represent Boolean values, when is `a and b` true? When is it false?
6. There is a binary infix operator `or` for Booleans. Repeat the previous exercise for `or`.

## 2.5 Casting

Casting allows you to ask Python to view the object you are looking at as having a different type, provided the change makes sense. Here we show how to cast an integer to a floating-point number.

```
>>> float(5)
5.0
```

A common use of casting is to convert a numerical string into a string or a number into a string. This sample session demonstrates a few simple casts. Try violating the rule and see how Python reacts.

```
>>> int("123")
123
>>> float("1.414")
1.4139999999999999
>>> str(2.7818)
'2.7818'
```

Casting is simple; the syntax is `newType(object)`, where `newType` is the new type you want the object to have. Be warned: Python will hiss at casts that make no sense. Be reassured: Anything can be cast to a string, but the result might not make much sense or be very useful. Also note that the original objects type does not change; the cast is a *temporary* request for a change of context.

## Programming Exercises

1. Perform the cast `int("211", 3)`. What happened? What is the significance of the second integer used? Experiment with other values and unravel the puzzle. When does this sort of cast throw a surly error message?
2. Enter the value `0b11100011` at the Python prompt. What happens? When does this procedure go sour? Tell exactly what is happening here.
3. What happens when you cast an integer to a `bool`? What about the other way around?
4. What happens when you attempt the cast `int("cows")`?

## 2.6 Relational Operators and the Boolean Type

Let's show an example of the Boolean type at work. It is very useful for looking at comparisons between objects which are done with *relational operators*. These are binary infix operators.

The operator `>` is the greater than operator and `==` is the isequalto operator. The operator `<=` is less than or equal to. Notice that `=` is not a relational operator. The operator `=` is the operator  $\leftarrow$  described back in Chapter 0. It performs assignment, which we shall deal with when we discuss variables.

The relational operators are context-sensitive; their behavior depends upon the types of their operands. You will find no surprises with number operands. Experiment with these at the Python command line. Here we a list of common relational operators. All of these are infix binary operators.

- `>` This compares two numbers, replying with `True` if the left number is larger than the right, and `False` otherwise.
- `<` This compares two numbers, replying with `True` if the left number is smaller than the right, and `False` otherwise.
- `<=` This compares two numbers, replying with `True` if the left number is smaller than or equal to the right, and `False` otherwise.
- `>=` This compares two numbers, replying with `True` if the left number is larger than or equal to the right, and `False` otherwise.
- `!=` This is the not-equals operator.
- `==` This checks for equality.

Let us show these operators at work.

```
>>> 5*5 > 6*4
True
```

```
>>> 5*5 <= 6*4
False
>>> 2 + 2 == 4
True
>>> 2 + 2 = 4
File <stdin>, line 1
SyntaxError: can't assign to operator
>>>
```

Notice the nastygram issued in response to the last command. It is a common error to use the assignment operator `=` instead of `==` check for equality. Take note of this error message; it is not the last time you will see it.

### Programming Exercises

1. What happens when you cast a Boolean to a string?
2. Enter `True + True` in an interactive session? What happens? What about `True * 5`? Can you state a general principal here?
3. Try casting various strings to a Boolean. Do you ever get `False`? (You can if you choose the right string)
4. Try casting various strings to a Boolean. Do you ever get `False`? (You can if you choose the right string)
5. Try casting various numbers (floats or ints) to a Boolean. Do you ever get `False`? (You can if you choose the right number)

## 3 Sequence Types

Have you ever had a nocturnal itch to store a bunch of related items together? Frinstance, if you have a sock drawer in your dresser, you can pull out a (hopefully clean) sock out of the drawer without undue rooting around in and under your dresser amongst the growling dust kittens? At another level of organization, you might even pair matching socks together when are finished laundering them so you can find pairs easily!

It is often a useful idea to store a group of related things in one place. Your dresser has drawers; hopefully you actually use them. If you do, you likely keep socks in one drawer, underwear in another (or in another part of the sock drawer), shirts in another, etc. We can do the same sort of organizing on Python objects: this will be accomplished with two new types, tuples and lists.

We have seen how to store a glob of text in a single place; to do this we use a Python string. As you saw in the exercises, a string is a smart character sequence the knows its characters and which can perform tasks based on the

characters it contains Strings and these two new types are called sequence types; these store sequences of objects under a single name. You will see that Python has a simple and elegant interface common to all sequence types.

Sequences are examples of *data structures*; these are containers for objects that are organized in various ways. As we progress we will learn about several types of data structures; for now we will look at lists, tuples and strings. This little table summarizes the basic properties of these three types.

- **str** This is the string type, which stores any sequence of characters (a string). The state of a string is completely specified by this character sequence. Strings constitute the chief means of storing text in Python. Once a string object is created in memory, its state cannot be changed; strings are *immutable*.
- **list** This is the list type which stores a sequence of Python objects. This sequence of objects completely specifies the state of a list. Lists provide a means of storing a collection of related items under a single name. A list is mutable; you can change the state of a list object. We will discuss mutation of lists at the end of the chapter. The objects present in a list or tuple are called its *entries* or *items*. The state of a list includes the objects present on the list, as well as the order in which they are stored.
- **tuple** This behaves much like a list, but tuples are immutable. A tuple is a “frozen list;” you will see that you cannot add elements to it or delete them from it. Its state, as the state of a list is embodied in the collection of object it contains and the order in which they are stored.

You might ask, “Why have an immutable type; it seems to be a disadvantage?” You will see later that immutability can have many advantages, and that, on the flip side, mutability can be very dangerous. We will shortly discuss this topic in its own section.

To get back to our main thread, we begin with a very simple example with strings. We saw before that we can concatenate strings and find the lengths of strings as follows.

```
>>> "hello" + " there"
'hello there'
>>> len("hello")
5
```

Let’s make some grammatical observations about strings. When you enter a string such as “hello”, you must enclose it in single quotes or double quotes; Python allows both, but be sure to delimit your string with the same kind of quote on both sides, or, as you have seen, you will be greeted with a surly error message.

The function `len()` tells you the length of any object of sequence type; in particular it tells you the number of characters in a string or the number of items present in a tuple or list.

You can concatenate (glue together) sequences of the same type using a `+` sign, as we just saw with strings and shall see in the next two sample sessions. Notice how a list is enclosed in square brackets. Each item inside of this list is a string, so each item must be in quotes. A list can contain Python object of any types; we say that Python lists are heterogeneous sequences. Notice how `len` works here, and how `+` concatenates sequences.

```
>>> jayWard = ["moose", "squirrel", "Wattasmatta U"]
>>> chuckJones = ["Bugs Bunny", "Daffy Duck", "Yosemite Sam"]
>>> cartoons = jayWard + chuckJones
>>> cartoons
['moose', 'squirrel', 'Wattasmatta U',
'Bugs Bunny', 'Daffy Duck', 'Yosemite Sam']
>>> len(chuckJones)
3
>>>
```

Notice how we are able to label the lists we created with the names `jayWard`, `chuckJones`, and `cartoons`. These names are called *variables*. We will discuss variables in greater depth shortly. Variables are Python's means of creating the symbols we discussed in Chapter 0.

We can also do all of this with tuples; notice that tuples are enclosed in parentheses.

```
>>> jayWard = ("moose", "squirrel", "Wattasmatta U")
>>> chuckJones = ("Bugs Bunny", "Daffy Duck", "Yosemite Sam")
>>> cartoons = jayWard + chuckJones
>>> cartoons
('moose', 'squirrel', 'Wattasmatta U',
'Bugs Bunny', 'Daffy Duck', 'Yosemite Sam')
>>> len(chuckJones)
3
>>>
```

Both tuples and lists are sequences. The difference is that we can add items to lists and modify them; these operations are not possible for tuples. We shall do a collection of examples later, showing how all of this works. But first, let us look at the common features of sequences.

**Sequence Operations** Here are a few useful features for handling sequences.

- `len()` This tells you the length of a sequence. Proper usage: If `x` is a string, `len(x)` is the length, or number of characters in, `x`. If `x` is a list or tuple, `len(x)` is the number of elements in `x`. The `len` construct is actually an example of a Python *function*; we will discuss Python functions in the next chapter.
- `in` If `x` and `y` are strings, the expression `y in x` evaluates to `True` when `y` is a contiguous substring of `x`. Otherwise it evaluates to `False`. The `in` keyword checks for membership of an object in a tuple or a list.
- `+` This concatenates sequences. Beware that, unlike the addition of numbers, this operation is not commutative. The sequence being concatenated must be of the same type, or Python will hiss.

## Programming Exercises

1. Make the string `x = "foo"` and cast it to a list and a tuple. What happens?
2. Make a list of strings. What happens when you cast it to a string? What about any list of Python objects?
3. Make a numerical list `numbers` and then evaluate `sum(numbers)`, `max(number)` and `min(numbers)`. What do you see? Do these work for tuples too?
4. What happens if you type `y = "cow's"`? What about `"cows"`? What general principle can you infer here?
5. Create a string and multiply it by a positive integer. What happens? What about a tuple or a list?
6. How can you use casts to take a non-negative integer and obtain a list containing all of its digits in order? *Hint*. Lists can sort themselves!

## 3.1 On the Importance of Type

We've now seen the action of a variety of operators on numbers. When you create an expression such as `5 + 3`, `+` is called the **operator** and `5` and `3` are the operands.

In general, the behavior of these operators is entirely dependent on the types of the operands. The principle at work here is that *type establishes context*.

Consider the binary operator `+`; this operator will take two numbers and return their sum or take two sequences and concatenate them (glue them together). If you are adding two numbers and either is a float, everything automatically becomes a float. This is true for `-`, `*` and `/` as well. If you try to add a number and a string, Python will rebel. Here is an example.

```
>>> "foo" + 5
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
>>>
```

You have been unceremoniously informed that Python has encountered a `TypeError` and it will have no further congress with your folly. Python will not concatenate a number and a string. Python will concatenate two strings, two tuples or two lists. Try mixing types and see Python hiss; by so doing it is defending the integrity of its type system and protecting you from errors that could pass silently and return to wreak havoc at some maximally unfortunate time.

There is an excellent semantic reason for this phenomenon. Remember, the binary operators are a behavior of numbers. If you use `+` on number and a string, you introduce confusion to Python; Python asks, “Shall I use string behavior or number behavior?” It then realizes it is confronted with a dangerous question. An ambiguity which should not pass silently is introduced: Python reacts by ending its activity in an error state. This informs you, the programmer, that there is a problem and Python will force you to make your intent explicit. You can achieve this in the example we just saw by casting the number 5 to a string by using `str(5)`.

The `*` operator has a useful behavior when it operates on an integer and a sequence. Look at this session; here we will use Python’s `print` function. It evaluates the expression inside of the parentheses and puts the result to `stdout`, which as you know from Chapter 1, is the screen.

```
>>> print(""*5)
*****
>>> print(""*5 + "&"*3)
*****&&&
>>> 3*"a"
'aaa'
>>>
>>> "a"*(-5)
''
>>> "a"*0
''
>>> [1,2,3]*3
9
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> (1,2,3)*3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>>
```



Here is what we surmise: If you multiply a sequence by a positive integer, that string will be repeated that integer number of times. If you multiply a sequence by a negative integer or zero, the result is an empty sequence.

**Exercises** These exercises are important; you will learn about the relational operators and how they act on strings. Do not skip them!

1. Create several strings with lower-case letters. Compare them with the relational operators. Repeat this for strings with upper-case letters. What do you see?
2. You can see the numerical (ASCII) value for any character by placing it in a string and using the `ord` function. Here is an example.

```
>>> ord("a")
97
>>> ord("A")
65
>>> ord("b")
98
>>>
```

You can do a reverse-lookup with the `chr` function as follows.

```
>>> chr(97)
'a'
>>> chr(98)
'b'
>>> chr(65)
'A'
>>>
```

See what happens when you type in various letters, numerals and symbols.

3. Create several single-character strings with lower and upper case letters. Compare these with the relational operators. What conclusion can you draw? Explore the byte-values of various characters and see in particular how the upper and lower case letters work. See how the digits 0-9 work.
4. Describe the behavior of the comparison operators `<=`, etc on strings consisting of only letters.
5. How do the order operators (`<`, `<=`, `>`, and `>=` behave on numerical lists? Perform an exploration and see if you can write down a simple rule.

## 4 Making your first Python Program `hello.py`

If you are using IDLE, look in the next subsection to get started, then come back here and follow the instructions. Be assured: things work pretty similarly on all platforms.

So far, we have used interactive mode in Python. When our session ends our stuff evanesces, unless we copy all of the commands and save them somewhere. Now we will create Python *program*; this is just a sequence of Python statements. Here is our first program. Enter it using vi or into an IDLE window; name the file `hello.py`

```
#!/usr/bin/env python3
print("Hello, World!")
```

In Python 3, we use `print("Hello, World!")`; specifically, surround anything you wish to print with parentheses. This is because in Python 3, `print` is a function, much like `len`. In Python 2, `print` it is just a simple command. You can use parentheses in Python 2 for `print`; this works just fine in both Python 2 and Python 3. This is another way to make your code work in the future if you are a python 2 user.

The first line of our program looks like gibberish, but shortly you will see it is useful. The program will run without it, but as we shall see in a few moments, it does something very cool on UNIX computers.

Whoa! It is crystal clear what the second line is doing! It's printing out the phrase "Hello, World". Before we run it we know what our program is going to do. Now take a little break, look at the next comparison section just ahead, feel lucky, and we will then run the program.

## 4.1 Creating a Program in IDLE

From the File menu select File → New. You will get a new editor window. Select Save As and save the (empty) contents as `hello.py`. Then enter your program, as instructed in the previous paragraphs. Make sure you pick a reasonable directory to save into; otherwise your stuff will get saved in places where you do not want it or you cannot later find it.

## 4.2 A Comparison with Some Other Languages

Here is the Hello World program in Java. You must be sure that it is in a file named `Hello.java`.

```
public class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, World!");
    }
}
```

Huh, `public?? static??? void?.....` And what is this `String[]` thing? Here it is in C++:

```
#include<iostream>
using namespace std;
int main(void)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

We see another traffic jam of arcane language keywords and mysterious notations. Happily, for us using Python, there is some serious plumbing here we don't have to plumb.

### 4.3 Running Your Program

Now, thank your lucky stars, save the program and exit vi. To run the program, do this at the UNIX command line

```
$ python3 hello.py
```

When you are done you will see this.

```
$ python3 hello.py
Hello, World!
$
```

**If you are using IDLE** Hit F5 to run your program. Its output will appear in the first window with the Python prompt.

The action of our program is to take the string "Hello World!" and to put it to the screen. Recall from Chapter 1 that the screen by default is actually `stdout`, or standard output.

We do this here. You may, if you wish, redirect standard output to a file as follows. We show the contents of the file using `cat`.

```
$ python hello.py > hugeTextFile.txt
$ cat hugeTextFile
Hello, World!
$
```

The command `python` behaves as a UNIX command. It takes as an argument the name of the file containing the program you wish to run. The use of the

.py extension is purely optional, but it does confer an important benefit. Your editor recognizes this extension and it automatically colors text in ways that will help you work faster and smarter. It is also configured to automatically format programs nicely.

**If you are using Python 3 (likely)** Go into your home directory and open your `.bashrc` file and enter this line.

```
alias python="python3"
```

Do not put spaces around the `=` or this will cause problems. When you are done, type this command at the UNIX prompt.

```
$ source .bashrc
```

Now, when you use the `python` command, you will use Python 3. If you want to run `python2`, just use a backslash like so.

```
$ \python
```

This will use the unaliased version of the `python` command.

Now we shall get to the mysterious first line. At the UNIX command line enter the following while in the directory with your program.

```
$ chmod u+x hello.py
$ ./hello.py
Hello, World!
$
```

The first line of the program tells UNIX how to find the Python interpreter, so your program automatically runs Python by itself. You can still run your program by typing `python hello.py` with the same result. This first magic line is often called the “shebang line”. The shebang line, if present, *must* be the first line of a Python program. Python programs can be executed repeatedly as needed and can be shared with others. Since they are text files, they occupy little space in your hard drive.

## 5 Comments in Python and on Python

Anything after a pound sign (`#`) on any line of a Python program is ignored by Python. You can use this feature to *document* your program. Documenting programs makes them understandable to you later. You can also use this feature

to include any instructions on how to properly run and use your program. In the professional world, others will often have to read and understand your code; in this arena good documentation is especially important.

All of the programs you write should have a *comment box* at the top. Here is `hello.py` with a comment box at the top.

```
#!/usr/bin/env python3
#####
#
#   Author:  Morrison
#   Program Name:  hello.py
#   Date:  15 June 2007
#   Description:  This program puts "Hello, World!" to stdout.
#
#####
print("Hello, World!")
```

Notice the color comments turn in the editor window. Also you should notice that the shebang line is a comment. It is seen by UNIX but ignored by Python. You will see comments appear in a special color if you use IDLE. The shebang line in IDLE will simply be a comment.

**A Scolding on Style** As you develop your skills bear in mind that programs should read like well-written technical paper. While it is important that it be correct for the computer to execute it as specified, it is important for it to be easy to understand. If you are a programmer, it is vital for your teammates to easily be able to understand your code so they don't waste valuable time attempting to decipher your obfuscatory coding arabesques. Since it costs a company over \$100 an hour in wages, bennies and overhead to keep a programmer at his desk, you can see that clear coding style and good documentation are essential to a company's bottom line. Since, presumably your time is valuable too, you will want to make best use of it by making your programs clear. You may want to go back and use them later.

**A Further Homily** There are three central values in programming: simplicity, clarity, generality. Bear these cardinal virtues in mind as you code. For a most excellent disquisition on this point of view, type `>>> import this` at the Python prompt. Be guided by this wisdom. Do this periodically as your knowledge advances and more will reveal itself to you.

## 6 Taking Full Advantage of print

Python's print function in Python 3 has some useful features that will give you greater control over output. We will mention two basic ones here, **end** and **sep**.

Let us begin with **end**. By default, Python places a newline after each use of the **print** function.

```
>>> print("Example")
Example
>>>
```

Now let us use **end = ""** and see the result.

```
>>> print("Example", end = "")
Example>>>
```

Here we see the line is ended with an empty string, so no newline is placed after **print**. You can tell this because the prompt did not move down to the next line. To confirm this create the program **trafficJam.py**

```
print("a", end = "")
print("b", end = "")
print("c", end = "")
print("d", end = "")
print("e", end = "")
print("f", end = "")
```

```
$ python trafficJam.py
abcdef$
```

All of the print statements are output on a single line. Since the last print did not create a newline, the prompt did not move down a line. Use **end** to end a print statement with anything you like. You can insert a newline by using **\n**.

Now let us look at **sep**. You can print several items at once by giving Python a comma-separated list.

```
>>> print(2*3, "cows", False)
6 cows False
>>>
```

By default, items printed out are separated by a space. You can use any separator you want by specifying it with **sep**

```
>>> print(2*3, "cows", False, sep = "|")
6|cows|False
>>>
```

Use `sep = ""` if you want no space between the items.

**Programming Exercises** Now you will have a chance to write some small programs and try out what you have learned.

1. Write a program that displays the following on the screen.

```
*
**
***
****
*****
```

2. Write a program that puts this "Christmas tree" on the screen.

```

      *
    ***
  *****
*****
*****
*****
*****
*****
*****
*****
*****
*****

```

3. Google "ASCII art"; you will find some interesting sites that create art from keystrokes in a terminal window. You can print out a string containing many lines using triple quotes like so

```
#!/usr/bin/env python3
print("""Here is a multiline
string
that goes on forever.
""")
```

Write a program that prints some ASCII art to the screen. See if you can make your own creation.

4. **Learn about Magic Characters** Python has some magic characters, or metacharacters, that are quite standard amongst modern languages. The sequence `\n` of two keystrokes actually represents a single character. So does `\t`. Here are two other metacharacters, `"` and `\"`. Figure out what these do.

5. Now use metacharacters to create a single string that prints this to the screen.

```
*****
*****
*****
*****
*****
```

and this

```
*   *   *   *   *
*   *   *   *   *
*   *   *   *   *
*   *   *   *   *
```

Refine this and make the strings you use as short as possible. How low can you go?

6. Make a this list of strings

```
>>> x = ["abcd", "efgh", "ijkl"]
```

Now enter `x[0][0]` at the Python prompt. What happens? Explore this business of double-subscripts and learn how it works. Does this work for tuples as well?

7. Python will print out a comma-separated list of items of any types. Try this.

```
>>> example = "Mr. Yoda Ears weighs", 11, "pounds. This is", True
>>> print (example)
```

8. Enter this at the Python prompt

```
>>> thing = [[1,2], [3, [4, 5], 6, [7, 8, 9]]]
```

Fiddle around and see if you can get Python to print this out.

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

What kind of object is this `thing`?

9. Use `sep` to print out this line from an HTML table. Can you use `end`, too?

```
<tr><td>3.4</td><td>4.5</td><td>5.6</td><td>6.7</td><td>7.8</td></tr>
```

10. Use `print` to format this. See if you can make it work at least 3 different ways. There's a panoply of 'em. Be as clever as possible.

```
-----
|0|1|2|3|4|5|
```



```

-----
|1|2|3|4|5|6|
-----
|2|3|4|5|6|7|
-----
|3|4|5|6|7|8|
-----

```

## 7 Variables, Assignment, Operators and Type

Now, we will see how to create the symbols we described in Chapter 0 using Python. There are two parts to this process. A *variable* in Python is a name that points to an object stored in memory; to wit, Python variables know how to find their objects. Here is a simple way to think of it. Your telephone number is a separate entity from your telephone. Your telephone number is like a variable: it is a means by which you can refer to, or send messages to, your telephone. Your telephone is the object and its number is its variable name. A phone needs to have a number or it is “orphaned;” it cannot be contacted via the phone system. A phone can be reprogrammed to a new number.

Variables point to objects; objects are what actually harbor type. We shall show a simple sample session here, and supply blow-by-blow commentary. As you read this, open your Python shell, and experiment as you follow along. Do not be afraid to “break things” and experiment. This is how we learn.

Lets begin by creating a variable named `x` and printing the value it points at.

```

>>> x = 5
>>> print (x)
5

```

When you see `x = 5`, do not read, “`x` equals 5;” read instead, “`x` gets 5.” The assignment operator `=` sets up a pointing relationship. The name on the left, `x`, points at the object on the right, 5. Assignment is not a symmetric operation, as we see in this little Python session.

```

>>> 5 = x
File "<stdin>", line 1
SyntaxError: can't assign to literal

```

Python is rebelling, informing us that the constant 5 cannot point at an object.

**Useful Terminology** Suppose you perform an assignment in this way.

```
leftHandSide = rightHandSide
```

What can appear on the right-hand side is an *expression*, which is just a combination of variables, literals and operators such as `+`, `-`, `*` and `/`. When an assignment occurs, the right-hand side is evaluated first. The result of this is called an *rvalue*. The item on the left must be able to point at an object. So far, the only things that can point at an object are variables. Things that can point at objects are called *lvalues*. Variables are lvalues. List items, since they can accept assignment, are also lvalues.

Next observe how we do some basic arithmetic. There is one surprise here if you are a Python 2 user. The division operator does integer division in Python 2 by default.

```
>>> y = 4
>>> print(x*y)
20
>>> print(x + y)
9
>>> print(x - y)
1
>>> print(x**y)
625
>>> print(x/y)
1.25
>>> print(x//y)
1
```

The operations `+`, `-`, `*`, and `/` behave exactly as we expect them to. However, since `x` and `y` point at integers, `/` is integer division. Python has a native exponentiation operator `**`.

In Python 2, If you want decimal numbers, you must cast to a the `float` type, which handles decimal numbers. Here is how to do it. Of course, this will work just fine in Python 3 as well.

```
>>> print float(x)/y
1.25
```

Using a decimal point will cause Python to view a number as a `float`. We could have written

```
>>> x = 5.0
```

and Python would view `x` as pointing at a floating point number. This would cause division to be floating point division.

Here is one nice little feature of `**` for floating point numbers. It provides a cheap way to compute a square root.

```
>>> w = z**(.5)
2.2360679774997898
>>> w*w
5.0000000000000009
>>>
```

Notice that floating point numbers do not store exactly. Do not be disturbed by the presence of a wacky digit or two out in insignificant digitville. This phenomenon is not particular to Python. Rather it is an artifact of the way in which floating point numbers are stored in computers.

Now let's point a variable at a string and demonstrate the action of `*`.

```
>>> name = "Ada"
>>> name * 5
'AdaAdaAdaAdaAda'
```

You can check the type of an object attached to a variable by using the `type()` function.

```
>>> type(x)
<class 'int'>
>>> type(y)
<class 'int'>
>>> type(name)
<class 'str'>
>>> z = 5.0
>>> type(z)
<class 'float'>
```

We should point out here that when we are entering `type(x)`, we are *not* asking the variable `x` its type. What we are asking is, “`x`, what is the type of object you are pointing at?” Remember, objects have type, variables are merely names. It is the object itself that actually tells you its type.

Lists and tuples are also types. We shall show that right here.

```
>>> moose = [1,2,3]
>>> regalis = (1,2,3)
>>> type(moose)
<class 'list'>
>>> type(regalis)
```

```

<class 'tuple'>
>>> moose == regalis
False
>>>
<class 'list'>

```

Observe that a list can never equal a tuple, and vice versa. Objects of different types are never equal; this is the so-called *species test*. Two objects of different species cannot be seen as equal. You can cast a list to a tuple, and vice versa, as we show here.

```

>>> moose == regalis
False
>>> goose = tuple(moose)
>>> goose
(1, 2, 3)
>>> hoose = tuple(moose)
>>> hoose = tuple(regalis)
>>> hoose = list(regalis)

```

## 7.1 Rules for Variable Names

The first character of a variable name can be a letter or the underscore (`_`) character. Subsequent characters may consist of numbers, letters or `_`. In principle, there is no length limit on a variable name, but you should try to be reasonable. The name of a variable must not start with a number. It cannot have a space or punctuation symbols in it. Avoid using underscores at the beginning and end of variable names; these are often used for special names with special interpretations which can cause surprises.

We recommend the *camel notation* for variable names requiring more than one word for a good description. You can also separate words with underscores if you wish like so: `is_even`. In this notation, words are separated by underscores instead of capitalization. Pick one scheme and stick with it; consistency is helpful to programmers.

Here are examples of legal variable names.

```

numberOfTrials
firstName
lastName
socialSecurityNumber
isRejected
number_of_cackles

```

Here is rogue's gallery of illegal variable names, and the reason why they are

taboo.

2BorNotToB	starts with a number
period.piece	presence of a period
semi;colon	illegal punctuation mark
space cadet	spaces are NOT allowed

## 7.2 Language Keywords

Certain words are reserved by Python for critical functions; never use these for variable names or you will get mysterious error messages. An example of such a keyword is `print`. If you type `print` in a `.py` file in `vi`, it turns a special color (depending on your system). Note that color; any word turning that color when typed is a Python keyword. Keywords also come out in a special color in `IDLE`. You will be alerted to keywords as we proceed.

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

## 8 Expressions and the Symbol Table

Python keeps track of variables and objects via a mechanism called the *symbol table*. You should think of the symbol table as a dictionary containing all variables as their “words” and the objects they point at as “definitions.”

Recall that an expression is any combination of variables and operators. For example, if `x` and `y` are variables, `x/y` is an expression. A variable by itself is an lvalue i.e. , it is capable of pointing at an object, because it can have a value assigned to it. Most expressions are not lvalues; for example, it makes no sense to write `x/y = 5`. Expression that are lvalues include such things as list items.

When an expression is encountered in Python, it is *evaluated*. In this process, the values pointed at by each operator are looked up in the symbol table, and they are combined as the expression dictates. For example, suppose that we have a variable `x` pointing at 5 and a variable `y` pointing at 2. In this case, the symbol table includes the following entries.

<code>x</code>	$\rightarrow$	5
<code>y</code>	$\rightarrow$	2

When we evaluate  $x + y$ , Python looks up, or *fetches* the value 5 is fetched from the symbol table for  $x$  and the value 2 from the symbol table for  $y$ . Then, 5 is substituted in for  $x$  and 2 for  $y$ . The result of evaluating  $x + y$  is  $5 + 2 = 7$ .

For objects of numerical type, the standard *order of operations* you learned in Algebra applies: first come parentheses, then exponents, then multiplication and division occur from left to right, and finally, addition and subtraction occur from left to right.

The assignment operator `=` has *lower* precedence than any of these. Let's see some examples of this at work. Notice that when an expression entailing variables is evaluated, the variables do not change. We merely fetch their values from the symbol table.

The assignment operator works in the reverse order from other operators. Things in an assignment statement are processed from right to left (Arabic style reading). So, in an assignment such as this one

```
x = x + y
```

the evaluation process works as follows. The value for  $x + y$  is found. Then the result is assigned to  $x$ .

Let's begin another Python session and illustrate this

```
>>> costello = 6
>>> abbott = 45
>>> moe = "chucklehead"
>>> joe = "nitwit"
>>> schempp = "dim bulb"
```

Making these assignments results in the following symbol table.

costello	6
abbott	45
moe	"chucklehead"
joe	"nitwit"
schempp	"dim bulb"

Now watch this code. A complex sequence of events occurs.

```
>>> abbott = abbott * costello
>>> print abbott
270
```

Python always begins by looking at the right-hand side of the assignment and it works to the left.

```
abbott = abbott * costello
```

It fetches the values for `abbott` and `costello` from the symbol table and evaluates `abbott * costello`. The result of this evaluation, 270, overwrites `abbott`'s entry on the symbol table. Now the symbol table looks like this. The old value, 45, for `abbott` is *orphaned*; it is still in memory for a while, but it has no variable referring to it. We show the updated symbol table.

costello	6
abbott	270
moe	"chucklehead"
joe	"nitwit"
schempp	"dim bulb"

Let's watch the evolution of the symbol table as we move along here.

```
>>> costello = (abbott - costello)*3
>>> costello
792
```

You might wonder what happens to orphaned values in Python. Do they must pile up, cluttering memory? The answer to this is no. Python has a facility called a *garbage collector*. The garbage collector lurks in the background, patrolling memory and freeing up the space occupied by orphans so it can be used for other purposes.

Coming back to our main thread, we see that `abbott` is pointing at 270 and that `costello` is pointing at 6. We evaluate the expression

```
(abbott - costello)*3
```

and the result is 792. This value overwrites `costello`'s old value and the symbol table looks like this.

costello	6
abbott	792
moe	"chucklehead"
joe	"nitwit"
schempp	"dim bulb"

Next, we will alter `moe`'s entry.

```
>>> moe = joe + schempp
>>> print moe
nitwitdim bulb
```

During this process, the values of `joe` and `schempp` are fetched from the last symbol table. They are concatenated and `moe` is redirected to point at by `"nitwitdim bulb"`. Here is the new symbol table.

costello	6
abbott	792
moe	"chucklehead"
joe	"nitwit"
schempp	"nitwitdim bulb"

Finally we see that `joe` and `schempp` are unaltered.

```
>>> print joe
nitwit
>>> print schempp
dim bulb
>>>
```

## 8.1 The Inside Dope on Assignment

You would likely do this without thinking.

```
>>> a = b = 5
>>> a
5
>>> b
5
>>>
```

Let's take a look inside and see what happens. First of all, when you evaluate something such as `5 + 2` and get `7` you are completely unsurprised. What is actually happening here is that `+` is actually a mathematical function. It takes its two operands add them, and returns the result (evaluation).

But, this is deception. Try this

```
>>> 5 == (x = 5)
File "<stdin>", line 1
    5 == (x = 5)
        ^
SyntaxError: invalid syntax
>>>
```

So, the value `5` is not, in fact, returned by the assignment. However, Python does support chained assignments such as



```
>>> b = c = d = 5
```

Python also supports *simultaneous assignment*. Take note of this little session.

```
>>> x = 1
>>> y = 2
>>> x,y = y, x
>>> print(x)
2
>>> print(y)
1
>>>
```

## 8.2 A Shorthand Convenience: Compound Assignment Operators

Python offers a shorthand that makes expressions cleaner and more succinct. If you have a binary infix operator `op`, which can be `+`, `-`, `*`, `/`, `%`, or `**`, you can write `x op= y` for `x = x op y`. These work for numbers and `+=` works for sequences. This little session show compound assignment at work.

```
>>> x = 5
>>> y = 2
>>> z = "foo"
>>> x += y
>>> x
7
>>> z += "goo"
'foogoo'
>>> z *= 3
z
'foogoofoogoofoogoo'
```

You should experiment with these operators and deliberately do illegal stuff. See and learn how to recognize the surly error messages that will result. Do not put a space in a compound assignment operator, or you break it and you will get an error message. Note that the item appearing on the left side of a compound assignment operator must be an lvalue.

## 8.3 Python is a strongly, dynamically typed language.

Variables are typeless: the objects they point at actually have type. The type of a variables object is determined when the program runs, hence the term

“dynamically typed.” By using the assignment operator `=`, you can make any variable point at a Python object of any type.

Python is strongly typed because it enforces rules about object type when expressions are evaluated. Python objects themselves have a keen awareness of their identity. The object 5 knows, “I am an integer.” This is important because, when you use operators such as `+`, the types of the operands determine the action of the operators.

Python’s typing system is sometimes referred to as “duck typing.” Suppose you have a variable `x` pointing at an object. If `x` has a behavior `foo()`, we would trigger that behavior by typing `x.foo()`. If you trigger the behavior `x.foo()`, Python checks at run time to see if `x`’s object has a `foo()` behavior available to it. If it does, the `foo()` action is triggered. If it does not, an error message is generated and your program dies an ignominious death in an error state. In other words, Python reasons that if “it quacks like a duck”, “it’s a duck”.

In contrast, other languages such as C, C++, and Java are statically typed. This means that a variable knows its type before the program is run and that it can only point at objects of its type. In such languages, type is determined at compile time, i.e. at the time the executable is built. This little example is quite informative

```
>>> x = 4
>>> x.upper()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: 'int' object has no attribute 'upper'
>>> x = "flimflam"
>>> x.upper()
'FLIMFLAM'
```

At first, we make `x` point at the integer 4. When we try to obtain `upper()` behavior from `x`, we get a nastygram from Python, saying, `upper()`? No such anniemule! Next we point `x` at a string. Since a string has behavior `upper()`, Python happily complies with our wish.

## 8.4 Python is a Garbage-Collected Language, or Charles Dickens Wrote of Orphan Heaven

Consider the fate of the string object "George" in this little session.

```
>>> name = "George"
>>> print name
George
```

```
>>> name = "John"
>>> print name
John
```

In the beginning, Python creates the string "George" and makes the variable name point at it. On the second line, we use print to see this; Python cheerfully echoes back "George". In the next line, name now points at "John". If you have been paying close attention, you would now ask indignantly, "Wait a minute, string objects are immutable? You cannot change **name**!"

What happened? Behind the scenes, you will see that "George" has been orphaned; "George" is an object with no variable pointing at it. There is no way for us to retrieve "George", once we orphan him. Objects are orphaned when they no longer have any variable pointing at them. Hence, we did not change the state of "George"; we just abandoned him to electronic oblivion.

Do orphaned objects just pile up in RAM, occupying space? Happily, the answer to this is no. We previously pointed out before that Python has the garbage collector; this lurks in the background and it reclaims the memory that orphaned objects occupy. This memory is then freed for your program to use. You just worry about creating objects; Python's garbage collector destroys them for you after you are finished with them.

Objects of types `int`, `float`, `long` and `bool` all work this way. All are immutable and when you do an assignment, the old value is orphaned, unless another variable is pointing at it. For example, this code

```
x = 5
x = x + 1
```

causes the following to happen. The integer object 5 is created in memory, and the variable `x` points at it. On the next line an assignment occurs. The value 5 is fetched from the symbol table under the name `x`. The righthand side is evaluated to 6. Then `x` now points at the value 6, and the object 5 is orphaned.

Java is also garbage-collected; C and C++ are not. In C and C++, you must manually destroy many of the objects you create. Failure to do this is often a source of *memory leaks* which can plague C/C++ programs. You can see this if you run a web browser for a long time. As a web browser runs a long time, it often gets slow and crotchety. This is because of memory leaks. Restarting the browser wipes the old browser process and all of its unused objects that piled up like cyberplaque, slowing your session to an infuriating crawl. Restart gives you a clean process with no leftover objects.

## 9 Sequence Operations

There are a variety of operators for sequences that are extremely helpful. When you write programs that process data, sequences play a prominent role, so methods that handle sequences help us to keep from reinventing the wheel. Let's first show a sample session, then explain their action in detail.

### 9.1 Indexing

You are given access to the entries a sequence with the `[]` operator. In the example below, a string is treated as a sequence of characters. The integer inside the `[]` is called an index. Python uses 0 for the first index of a sequence. We show this for a list and a string; it works the same for a tuple. Try it!

All of these indexing operations give you a copy of a part of a sequence. Since you get a copy, slicing and item access do not change the object they are applied to. Notice that Python begins counting at 0.

```
>>> x = "abcdefghijklmnopqrstuvwxy"
>>> y = ["aardvark", "bat", "cerval"]
>>> x[0]
'a'
>>> y[0]
'aardvark'
>>> x[1]
'b'
>>> y[1]
'bat'
>>> x[25]
'z'
```

Python will hiss if you attempt to use an index that is out-of-bounds. Here is a very common `IndexError` mistake.

```
>>> x[26]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

You will get the same message if you try to gain access to `y[3]`.

Python has a clever feature for counting from the end of a sequence.

```
>>> x[-1]
'z'
```

```
>>> y[-1]
'cerval'
>>> x[-2]
'y'
>>> y[-2]
'bat'
>>> x[-26]
'a'
>>> y[-3]
'aardvark'
```

Don't go too far!

```
>>> x[-27]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Python's indexing system makes a lot more sense if you think of indices as living between the elements of a sequence. For example in the string "hello" you should have this mental picture

```
*-- -5-- -4-- -3-- -2-- -1
| h | e | l | l | o |
0----1----2----3----4----5
```

The indices lurk between the items in the sequence. The positive indices are “normal;” they point to the character immediately to their right. The negative indices are “sinister;” they point to the character to their left. This is extremely handy when we talk about taking slices from a sequence.

## 9.2 Slicing

You can get pieces of sequence using a feature called *slicing*. Here we can get the all of the string entries *before* index 5 or staring at index 5.

```
>>> x[:5]
'abcde'
>>> x[5:]
'fghijklmnopqrstuvwxyz'
```

We can obtain the length of a sequence using `len()`.

```
>>> len(x)
26
>>> len(y)
3
```

You can also specify where to start and where to end before in a string slice. Here we get the slice of `x` starting at index 5 and ending before index 7.

```
>>> x[5:7]
'fg'
```

Notice how the design of indices make things spiffy.

```
>>> x[:5] + x[5:]
'abcdefghijklmnopqrstuvwxyz'
```

### 9.3 The `in` Keyword

The keyword `in` is actually a binary infix operator. Here we show it where both operands are strings.

```
>>> "abc" in x
True
```

Yeah, `"abc"` is in `"abcdefghijklmnopqrstuvwxyz"`; the `in` feature checks and sees if the its left operand is a contiguous substring of the operand on the right.

```
>>> "abc" in x[5:]
False
```

We see that `"abc"` is not in `"fghijklmnopqrstuvwxyz"`.

```
>>> "abe" in x
False
```

The characters `"abe"` appear in order in `x`. They, however, are not contiguous! Hence the `False`. Oops! Hence the `False`.

The behavior of the `in` operator is different for lists and tuples; in this case, it a check for membership in the tuple or list. This mechanism works identically for tuples or lists; here we show it working on a tuple. The left operand here is an object and the right operand is a list or tuple.

```
>>> cows = ("guernsey", "brahmin", "texas longhorn")
>>> "siamese" in cows
False
>>> "brahmin" in cows
True
>>> 56 in cows
False
>>>
```

**A Formal Description of Sequence Operators** We gather what we have learned so far all in one handy place. You will notice that things work very similarly for strings, tuples and lists. This sort of parallelism makes for some pleasing economy of thought.

**Entry Access** `[]` The square bracket operator allows us to extract a sub-sequence or a single item in a sequence. If `x` is a sequence and `a` and `b` are integers, then

- `x[a:b]` is the string starting at index `a` of `x` and ending at index `b`. If `a >= b`, then the result is an empty string. It is an error to try to use indices that are out of bounds.
- `x[:b]` is the string starting at the beginning `x` and ending at index `b`. It is an error to use a value for `b` that is out of bounds.
- `x[a:]` is the string starting at the `a`th character of `x` and ending at the end of `x`. It is an error to use a value for `a` that is out of bounds.

**String and Tuple Immutability** All of the slicing methods hand you a *copy* of the indicated subset of a string or tuple. Strings in Python are *immutable*; once you create a string object, you cannot change it!

```
>>> x = "moo"
>>> x[0] = "f"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>>
```

Python does not allow you to alter the entries of strings. Hence we say that a string is an *immutable sequence type*. Strings are sequences of characters that are “written in ink.” Once you create them, you cannot change their entries.

However, you can create a new string object and orphan an old one; it is common to “frankenstring” new strings from existing ones using the `[]` operator.

Here is a simple example continuing on the last Python session. The symbol table for as of now is

$x \rightarrow \text{"moo"}$

```
>>> x = x + " cow"
>>> print x
moo cow
```

In this process here is what is happening. In the line

`x = x + " cow"`

we fetch the value `"moo"` from the symbol table for `x` and `x + "cow"` evaluates to `"moo cow"`. We then tell `x` to point at this new string. The symbol table becomes

$x \rightarrow \text{"moo cow"}$

The object `"moo"` is now orphaned. It awaits the coming of the garbage collector. It is no longer accessible to you.

It is useful to point out that `float`, `bool`, and `int` are all immutable Python objects. As a result, when you have a piece of code like

```
x = 5
x = x + 1
```

the following happens. The integer object `5` is created in memory, and the variable `x` points at it. On the next line an assignment occurs. The value `5` is fetched from the symbol table under the name `x`. The right-hand side is evaluated to `6`. Then `x` not points at the value `6`, and the object `5` is orphaned.

So far the only mutable type we have encountered is `list`. When you slice a list, you are handed the actual sequence of objects. You can, for instance, assign an empty list to a slice of a list and that slice will be deleted from the original list. You can assign a list to an empty slice and it will be spliced into your list. You can read about this in the section entitled “Mutability and its Dangers.”

## 10 Advanced Topic: Understanding the Pointing Relationship

We have said that “variables point at objects.” We need to elaborate on this relationship to fully understand what it means for us when we program from



a practical standpoint. Full understanding of this phenomenon makes some seemingly confusing issues that will crop up later crystal clear. You should think of a variable as a name. Variables are brought into existence with some kind of assignment statement. For example `x = 42` causes the variable `x` to point at the integer object 42. In C or Java, you would have to say

```
int x = 42;
```

In the C language, this creates an integer variable holding the bytes for the number 42. An integer on a standard computer today is 32 or 64 bits. That size is not changeable.

In Python we could subsequently say `x = "In the beginning ..... (the whole Bible)..."`. We know the Bible is huge; as a text file is several megabytes. In C or Java, assigning `x` to such a thing is an error. How can you expect to cram an entire Bible into a space of 32 or 64 bits? You cannot! How does Python circumnavigate this seemingly impossible problem? The solution relies in the magical process of indirection, which we are about to meet.

What follows is an incomplete model, but it provides an excellent insight into how Python actually works. You should recall that your RAM is divided into bytes, each of which have an address. Every program that runs on your box gets a virtual address space. This is a “sandbox” of memory the operating system gives to your program. The memory in RAM is not necessarily a contiguous block of addresses. However, in a beautiful feat of abstraction, the OS gives your program virtual addresses, which appear to your program to be contiguous. The OS handles the ugly problem of translating the virtual addresses into real hardware addresses. It controls all processes (running programs) on your machine and manages their virtual address spaces. Your processes’s virtual address space is like a little, private computer for your process that other process may not see or tamper with.

All access of values via variables is accomplished by the miracle of indirection. What a variable stores is not an actual value. What it stores is a virtual memory address where the object it is pointing at can be found. This behaves much like the telephone system. Your cell phone has a number, which is its address. Calling that phone causes it to ring and to (hopefully) cause you to answer it. Your physical phone, in this analogy is the object. The caller might be your mother, who could send you messages via your phone you are prepared to understand, such as “come home” or “tell me what time to expect you for dinner.” Your phone is the object here; you receive on and act upon the message. Your mother gets access to you via your phone number. In fact, there are three levels of indirection here: your mother uses her phone (level 1) to dial your phone (level 2) to send a message to you (level 3).

The act of assigning a variable triggers a complex of events. Room for its object is allocated in your program’s memory and its object (string, list, number, etc.) is created. The variable stores the object’s virtual memory address. This

is how it points at its object. Virtual memory addresses are just integers that are displayed in hex code. When you point the variable at another object, you cause it to store the new object's address. If you have no variable pointing at the old object, the old object is orphaned.

Objects are quite complex. They have data (like the characters in a string) and they know how to do things to themselves. Objects are smart in that they are aware of their type and identity and they have the code they need to carry out the tasks entailed in messages sent them.

So let's go back to the example of `x` being 42 and then being a string with the entire Bible in it. In the beginning, the value 42 is an integer object stored in memory. When we point `x` at the bible, Python makes space for the string containing the Bible and places it in memory. It then gives `x` the memory address where we can find the Bible.

So far the only mutable type we have encountered is `list`. When you slice a list, you are handed the actual sequence of objects. You can, for instance, assign an empty list to a slice of a list and that slice will be deleted from the original list. You can assign a list to an empty slice and it will be spliced into your list. To wit, a slice of a list is an lvalue or any entry in a list is an lvalue.

An integer is a very simple object; it remembers its datum (such as the number 42) and its type. Now when we assign the variable `x` to the string with the Bible in it the following happens.

1. The Bible is placed into the virtual address space of your Python session. If your session runs out of space, then Python will request more room from the OS; if this is not granted you will get some horrid memory error (unlikely in this case).
2. The code that makes a string smart is placed right next to it.
3. `x` stores the first memory address of this whole leviathan.
4. The old object, 42 is left behind.

You can think of this complex process very simply. Variables know *where* to find their objects. They do not know anything *about* their objects. The specification of an object's location is just a memory address, which is just an integer. So, variables only really store integers. The magic is in the indirection: the integer tells you where a complex object that is very smart is located. What is nice for you to know is that you do not have to understand the inner workings of a smart object to get it to do work for you.

## 11 Mutability and its Dangers

Consider this innocent little act.

```
>>> cats = ["bermese", "siamese", "russian blue"]
>>> meowers = cats
>>> cats
['bermese', 'siamese', 'russian blue']
>>> meowers
['bermese', 'siamese', 'russian blue']
>>>
```

Now, what is interesting is that a list is mutable. Entries and slices of lists are lvalues; we can assign lists to them, and thereby change items. Here, let's change bermese cats to calico.

```
>>> cats[0] = "calico"
>>> cats
['calico', 'siamese', 'russian blue']
>>>
```

Prepare yourself. . .

```
>>> meowers
['calico', 'siamese', 'russian blue']
>>>
```

Both lists changed! What happened? What we see here is a phenomenon called *aliasing*. The seemingly innocent step `cats = meowers` provides the clue. Remember: variables store the address of objects. We actually access those objects via indirection. What we did here is make `cats` and `meowers` point at the same object; this is so because both store the same memory address. Since lists are mutable, any variable pointing at a list can change its state. This can be dangerous and can produce unexpected and undesired results. The perils multiply in the next chapter when we begin using functions. Nonetheless, mutability can be very convenient and can add to performance. We must respect its power and its perils, much as we do any powerful tool.

**Programming Exercises** In these exercises, you will explore the world of sequences. These exercises convey some important information we will use later.

1. . Create an empty list named `dogs` by entering

```
>>> dogs = []
```

Now enter this command.

```
>>> dogs.append("standard poodle")
```

What does this do? Use it to populate the list with more breeds. The dot (`.`) says, “Object dog, append the object I give you to yourself.”

2. If you type

```
>>> dogs.sort()
```

what happens to the list `dogs`? What message are you sending the list?

3. Make a tuple and try to use `append` and `sort` on it. Explain what happens. Try these operations on a string and take note of the results.
4. Create an list named `l`, a string named `s` and a tuple named `t`; make sure these contain at least ten elements. Then enter `l[:2]` at the command line. Does it modify `l`? Try placing numbers between the two colons. What happens. Try this for `tt` `s` and `t` as well. Does doing this modify any of `l`, `s`, or `t`?
5. Now enter `l[len(l) - 1, -1, -1]`. What happens? What else can you do?
6. You can cast anything to a string. What does casting a list or a tuple to a string do? What happens if you cast a string to a list or a tuple?
7. Create a string as follows.

```
>>> x = "abcdefghijklmnopqrstuvwxyz"
>>> x = x + "ABCDEFGHIJKLMNOPQRSTUVWXYZ, !@#$1234"
```

Send a string the messages `lower()` and `upper()` by using the dot notation, `x.lower()` and `x.upper()` Do they affect the original string? What do you see? What can you say a string knows how to do from what you have seen here?

8. . What happens if you assign an empty list to a slice of a list? Can you assign a list to a slice of a list? What is the result? What happens for a tuple?

**The Operator `+=` for Sequences, the Keyword `is`, Pooling, and Mutability** We said earlier that an object has state, identity and behavior. We have `==` to check for equality. Can we check for equality of identity? The answer is “yes;” to do so use the keyword `is`. If you have variables `x` and `y`, `x` is `y` returns `True` if `x` and `y` both point at the same object. Let us illustrate with a simple example. We create a string and assign it to another variable. There are no surprises here.

```
>>> x = "some"
>>> y = x
>>> x is y
True
>>>
```

Here is where we see something interesting. Strings are immutable, so the code on the first line `x = x + "thing"` causes `x` to point at the string

"something". The string y is unaffected. Since strings are immutable, Python cannot modify the object that x is pointing it. Instead, it creates a whole new string, "something" and has x point at it. The variable y is still pointing at "some".

```
>>> x = x + "thing"
>>> x
'something'
>>> y
'some'
>>> x == y
False
>>> x is y
False
>>>
```

In Python, the += operator appends sequences to sequences. Here we show it working on strings. This operator had the same action as

```
x = x + "thing".
```

```
>>> x = "some"
>>> y = x
>>> x is y
True
>>> x += "thing"
>>> x
'something'
>>> y
'some'
>>> x == y
False
>>> x is y
False
>>>
```

Now we examine this behavior on a list and a tuple. A list, in contrast to a string, is a mutable sequence type. A tuple, in like a string, is an immutable sequence type.

```
>>> xlist = [1,2,3]
>>> xtupple = (1,2,3)
>>> ylist = xlist
>>> ytupple = xtupple
```

```
>>> xlist is ylist
True
>>> xtupple is ytupple
True
```

We see no surprises. Now we will use += to tack on a new element for each. Note that a singleton tuple requires the comma after the value to be recognized as a tuple.

```
>>> xlist += [4]
>>> xtupple += (4,)
```

You see that `xlist` and `ylist` still point to the same object.

```
>>> xlist is ylist
True
>>> xlist
[1, 2, 3, 4]
>>> ylist
[1, 2, 3, 4]
```

Contrast this to the fate of `xtupple` and `ytupple`.

```
>>> xtupple is ytupple
False
>>> xtupple
(1, 2, 3, 4)
>>> ytupple
(1, 2, 3)
```

A new object is constructed for `xtupple` and `ytupple` is unaffected. No aliasing occurs here. The compound assignment operator += works for all types of sequences. Its action, however is affected by the mutability of the sequence.

## 12 Advanced Topic: Pooling

Certain types of objects in Python are *pooled*, or cached in memory. An example of this is string objects. Here is how it works. Python maintains a set of all reasonably small strings used in the lifetime of your program. Instead of orphaning them, it keeps them in an area of memory called the **string intern pool**. A string is never included in the pool twice. The pool is an area of memory organized for the strings your program uses. These objects get recycled. To see evidence of this observe the following contrast of list and string behavior. We begin by making a string and a list.

```
>>> pool = ["swimsuit", "sunscreen", "rubber duck"]
>>> spool = "fishing line"
```

Now if we take a slice, we know we should get a copy of each sequence.

```
>>> poolcopy = pool[:]
>>> spoolcopy = spool[:]
```

But when we test for equality of identity, we are in for a surprise!

```
>>> pool is poolcopy
False
>>> spool is spoolcopy
True
```

The lists `pool` and `poolcopy` are separate. This must be done, since a list is mutable. Were they to point at the same object, they would become alias for each other. This would violate the requirement that slices return copies of their segment of a sequence!

Since strings are immutable, no second copy is needed. Python is very clever; it just tells `spool` and `spoolcopy` to point at the same string! Since neither can modify this object, it's perfectly safe and it saves memory. Mutable objects may never be pooled.

There is an added bonus here. If two strings are pooled, checking string equality is simple: Python just has to check for equality of identity (this is an integer comparison). It does not have to move through each string, checking the equality of characters. Since this operation of comparing all characters in a string is proportional to the length of the string, you can see that considerable economies are achieved here.

Strings are pooled because nearly every program traffics in them. Python gains efficiency from this feature. Below, you will check out other types and see if they are pooled. What types make the most sense for pooling?

## Programming Exercises

1. You saw how to test if a type of objects is pooled; so far we only know strings are pooled. What about `bool`, `int`, `long` and `float`?
2. How about tuples?
3. What is the largest number of Boolean objects Python will ever actually store?

## 13 Useful Learning Resources

The best source of information on Python is on the Python site, <http://www.python.org>. The documentation can be found at <http://docs.python.org>.

Wikipedia has an article on string pooling at <http://en.wikipedia.org/wiki/Stringinterning>. The Java and Ruby languages also pool strings. Since strings are mutable in C++, they are not pooled in that language. You can download the complete documentation to Python and store it on your box. The documentation provides a complete guide to all of the Python language features, libraries and modules.