Chapter 0, An Introduction to Python: Objects, Variables and Types

John M. Morrison

December 19, 2022

Contents

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 to. 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 has an abundance of useful information. 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 if you have an account on one. There are complete instructions on the site for installing and using Python on any platform. We will emphasize using Python in a command-line environment in this book. The video linked at the end of this section is a very helpful guide to installing Python on your computer. Python is available in two versions, currently 2.7 and 3.11. As of 2022, Python 2.7 is at "end of life," so we will focus on Python3 in this book. However, it is good to be aware of Python 2, since you are likely to see in in OPC (other people's code).

After you get Python running, this chapter will introduce to the ways in which Python stores data, what kind of data it stores, and how it data available to you.

As you read this chapter, you will want to get a session of Python open and experiment with the things you see. Remember: to understand something, you must bend it, break it, and understand its strengths and limits.

Corey Schaefer's video contains complete instructions for MacOSX and Windoze. Make sure you check the box for updating your path and that you run the tests at the end of the video to ensure that Python is installed properly. One bonus of doing this is you will get a mini-introduction to Python. The Windoze version of installation begins at 5:30.

0.1 Your Wish is My Command

Throughout we will refer to a MacOS terminal, a cmd window, or a PowerShell window as a *command window*. Here is how to get this window to appear on all platforms.

On a Mac, Terminal is in `/Applications/Utilities`. Drag the terminal icon to your application dock. Double-click on it to launch it. You will see some text resembling this in it.

```
MAC:Wed Jun 02:17:54:python>
```

The text you see is called the *prompt*. When we discuss command windows, we will abbreviate this text with a $.

On a Windoze machine, type `cmd` or `PowerShell` in the search box and hit ENTER. This will cause a command window to come up that looks pretty much like its MacOSX cousin.

If you are running a Linux desktop such as Ubuntu, there is a terminal icon present on the desktop. If you are not sure where it is, type `terminal` in the search and it will appear. Once you do this you are ready for the next step.

0.2 The Nitty-Gritty: Launching Python on your Local Machine

To begin an interactive Python session, type

```
$ python3
```

in the command window. You will see something like this

```
Python 3.11.0 (v3.11.0:deaf509e8f, Oct 24 2022, 14:43:23)
[Clang 13.0.0 (clang-1300.0.29.30)] 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
>>>
```

If you have not seen the mod operator before, here is what it does. The expression `b%a` is evaluated by dividing `a` into `b` and computing the remainder.

To quit, type control-d on Mac or Linux, control-z in Windoze, or `quit()` on any platform. The characters control-d and control-z are end-of-file characters in their respective operating systems. 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.

A Road Map Python stores all data in regions of memory called *objects*. Objects are not just data; they are also "smart" in that they are aware of themselves and they can perform useful tasks. Python allows you to store objects under names called *variables*. This chapter will teach you about the basic types of of objects, how to get them to do work for you, and how to store them for later retrieval using variables. Knowing how they work and what services they can perform for you can save a lot of work.

You will combine variables and objects into *expressions*, and learn the grammar for writing expressions that Python can understand and evaluate.

As a result of reading this chapter, you will be able to write very simple programs that carry out the statements in them in the order in which the statements are shown.

We will begin by learning how Python manages such basic stuff as numbers, Booleans (`True`/`False`), and text. As you read this chapter, do plenty of experimenting. Deliberately "break" things and see how Python reacts. This first step is the most basic step in learning to program.

1 Types, Objects and Numbers

Computing is about the manipulation of data; all data in Python are represented by objects. 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 big 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.

1.1 Properties of Objects

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 object's physical presence in heap 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 `int` object is simply the integer it 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 just met in the session above exhibit the expected useful behavior in the presence of arithmetic operators. 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.

1.2 Python's Number Types

Objects of number type are aware of arithmetic operations, 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.

- `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 (64 bit) 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 in Python 2, and does not exist in Python3. In most languages, integers are restricted to a range, typically -2^{32} to $2^{32} - 1$. This was `int` in Python 2. Python 3 merges the `int` and `long` types. If you program in Python 3, you need not worry about the distinction between `long` and `int`.
- `complex` This is Python's complex number type.

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'>
```

Programming Exercises Do these right away to see some useful stuff.

1. At the prompt type `type(True)`. What do you see?
2. At the prompt type `type("cows cows cows")`. What do you see?

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 of the same type. 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 the 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 6.58e21 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 π with 3.14.
4. Find the surface area of the earth in square miles. Determine the equivalent in acres.

1.3 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; failure to do so is rewarded with an error messge. 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. Such as string is malformed and it triggers an error.

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
```

Programming 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")`

1.4 Introducing Variables

Now, we will see how to create the symbols that refer to objects in Python. There are two parts to this process. A *variable* in Python is a name that points to an object stored in heap memory; to wit, Python variables know how to find their objects on the heap. There is another important piece of memory for you to know about, the *stack*

Values of variables and information on the objects they point at is kept in the stack. The stack is a fixed-sized, relatively small piece of memory, where the heap is much larger and it can grow in size if needed (and the cranky operating system doesn't say no).

Here is a simple way to think of variables. Your telephone number is a separate entity from your telephone. A telephone number is like a variable: it is a means by which you can refer to, or send messages to, a 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. Take especial note of this fact: *Variables are typeless names that point at objects.* They are a means by which we gain access to objects. The information about variables and where they are pointing are stored in the stack. You will learn later the raison d'etre for this name.

1.5 Getting More Information about Objects

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.

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.

Let's 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 `=` sign in Python is called the `assignment operator`. 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.

What is a literal? A literal is a concrete object, as opposed to a vriable. Examples include these the following.

`"fooment"`	This is the string `"fooment"`.
`42`	This is the integer `42`.
`1.414`	This is the floating point number `1.414`.
`True`	This is the boolean constant `True`.

1.6 Reading Python Documentation

Begin by going to The Python Site. Click on the documentation link and select the Python 3.x documentation.

You will want to explore this site for tutorials and other information. This is a comprehensive reference on Python, which you will learn how to use. Now visit The Standard Types Page; this page gives a lot of detail on the built-in types, which includes strings. 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 Now you will get a chance to work with variables and strings.

1. Create a string `alpha = "abcdefg"`.
2. What happens when you type `alpha[0]`?
3. What happens when you type `alpha[1]`?
4. What happens when you type `alpha[2]`?
5. What happens when you type `alpha[-1]`?
6. What happens when you type `alpha[7]`?
7. What happens when you type `alpha[3:]`?
8. What happens when you type `alpha[:3]`?
9. What happens when you type `alpha[1:5]`?
10. 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?
11. 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 = "bugs bunny"
    >>> 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?
12. If there is an `endswith`, there is a `startswith`. Experiment with it.
13. What is happening here?

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

14. Make the string

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

15. What do `rstrip()`, `lstrip` and `strip()` do to it?
16. Try some of these out on various one-character strings. What happens if you use them on a string with more than one character? Do some experiments to figure this out.

    ```
    isspace()
    isdeciaml()
    islower()
    isalpha()
    ```

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

1.7 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, and constitute the two Boolean literals. The exercises shown here will take you on a guided tour of this type. It is important to do them before moving on.

Exercises

1. At the Python prompt enter `not True` and `not False`. What happens?
2. If `b` represents a boolean value, what is the relationship between `b` and `not not b`?
3. Since `not` is an operator, would you describe it as prefix, postfix or infix? Would you describe it as binary or unary?
4. 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?
5. There is a binary infix operator `or` for Booleans. Repeat the previous exercise for `or`.

2 Variables, Assignment, Operators and Type

We begin by discussing some 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 +, -, *, `not`, `or`, 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. We will see that there are other types of lvalues we progress.

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. Take a look at this Python session.

```
>>> x = 5
>>> 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. Python has a native exponentiation operator `**`. It is an infix binary operator. Try it out!

In Python 2, If you want fractional numbers, you must cast to 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. The inclusion of a decimal point makes a literal number a floating point number.

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. Why is this? We refer you to the F-word in computing, "finite." Python floating point numbers are 64 bits in size. What can we store in 64 bits of memory? The limit is $2^{64} = 18,446,744,073,709,551,616$ possible values. Hence, there are only finitely many floating-point numbers out there to represent the uncountable continuum of real numbers. Therefore, a `float` is an *approximation* of a real number.

Do not be disturbed by the presence of a wacky digit or two out in insignficiantdigitville. This phenomenon is not particular to Python. Rather is 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.

Programming Exercises

1. What happens if you use * on a string and a negative number? On zero?

2. How would you print out

```
ababab   ..... ab (50 times)
```

2.1 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's 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 *snake 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`. Snake notation is quite standard in the Python programming world.

Here are examples of legal variable names.

```
number_of_trials
first_name
lastName
social_security_number
number_of_cackles
x1
```

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
```

2.2 Language Keywords

Certain words are reserved by Python for critical functions; never use these for variable names or you will get mysterious error messages. Here is a table of them.

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	

An example of such a keyword is `lambda`. If you type `lambda` in a `.py` file in a text editor, it turns a special color (depending on your system and editor). Note that color; any word turning that color when typed is a Python keyword. You will be alerted to keywords as we proceed. Make sure you use a code-savvy editor if you are using a text editor to create Python code.

2.3 The Big Picture

In Python you have variables and objects. Variables are names that point at objects on the heap. The variables themselves and where they point on the heap is stored on the stack. To make a variable point at an object, use the assignment operator `=`. The right hand side can be a literal or it can be an expression composed of literals and variables. This expression is evaluated the result is placed on the heap and is pointed at by the variable on the left-hand side of the assignment.

Programming Problems

1. What happens if you type `x = y = 5` at the Python prompt?
2. What happens when you run these Python statements at the prompt?

```
a = 5
b = 2
print(a, b)
a, b = b, a
print(a, b)
```

3. What sort of tricks can you do with more than two variables with the technique you saw in the last problem?

2.4 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. This is a temporary request; it does not change the type of the object being cast or its value. Here we show show 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)` or `newType(expression)`, where `newType` is the new type you want the object or the result of evaluating the expression 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 object's type does not change; the cast is a *temporary* request for a change of context. You can, of course assign the result of a cast to a variable if you want to keep it around.

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")`?
5. What happens if you cast a `float` to an `int`. Try this for both positive and negative numbers.

2.5 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. They are valid in a variety of contexts.

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.

- The operator `>` is the greater than operator.
- The operator `<` is the less than operator.
- The operator `==` is the isequalto operator.
- The operator `!=` is the notequalto operator.
- The operator `<=`is less than or equal to.
- The operator `>=`is greater than or equal to.

Notice that `=` is not a relational operator; it is the assignment operator that makes a variable point at an object.

Here we show the relational operators in the context of numbers. Both operands must be numbers, or Python will hiss.

- `>` 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 notequalsto operator for numbers.
- `==` This checks for equality of its numerical operands.

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. Enter `True + True` in an interactive session? What happens? What about `True * 5`?
2. Try casting various strings to a Boolean. Do you ever get False? (You can if you choose the right string)
3. Try casting numbers to Boolean. When do you get `True` and when do you get `False`?
4. Let `x = "123"` and perform these casts.

   ```
   int(x)
   int(x, 4)
   int(x, 5)
   int(x, 6)
   int(x, 7)
   int(x, 8)
   int(x, 10)
   ```

   What is the meaning of the second arguments 4-10? Experiment and break the code!

3 String Conveniences

Python has a few features that you will find convenient as you progress. The simplest is the *triple-quoted string*, which allows line breaks inside of a literal string. We supply an example here

```
>>> x = """I can span several
... lines.  Since this is interactive
... there are three dots but they
... won't appear in the final product."""
>>> print(x)
I can span several
lines.  Since this is interactive
there are three dots but they
won't appear in the final product.
>>>
```

You can use single or double quotes to bound a triple-quoted string, but make sure you are using the same type throughout or you will get hissed at.

3.1 The Raw Bar

You should note that the character \ is what is called an "escape character." To make an actual backslash, you must use two backslashes like so: \\. This gets annoying if you are working with stuff such as Windoze file paths. For this purpose, Python has *raw strings*. We demonstrate this feature here.

```
>>> path_to_perdition = r"C:\Program Files\Bill Gates\hot_mess.py"
>>> print(path_to_perdition)
C:\Program Files\Bill_Gates\hot_mess.py
>>>
```

Preceding a string with an `r` makes it a raw string. For such strings, a backslash is just .... a backslash. And, yes, you can make a triple-quoted string be raw.

A Do-now Exercise Print the strings. This is important! Repeat, making them raw strings.

1. `"a\tb\tc\td"`
2. `"a\nb\nc\nd"`

3.2 f-strings

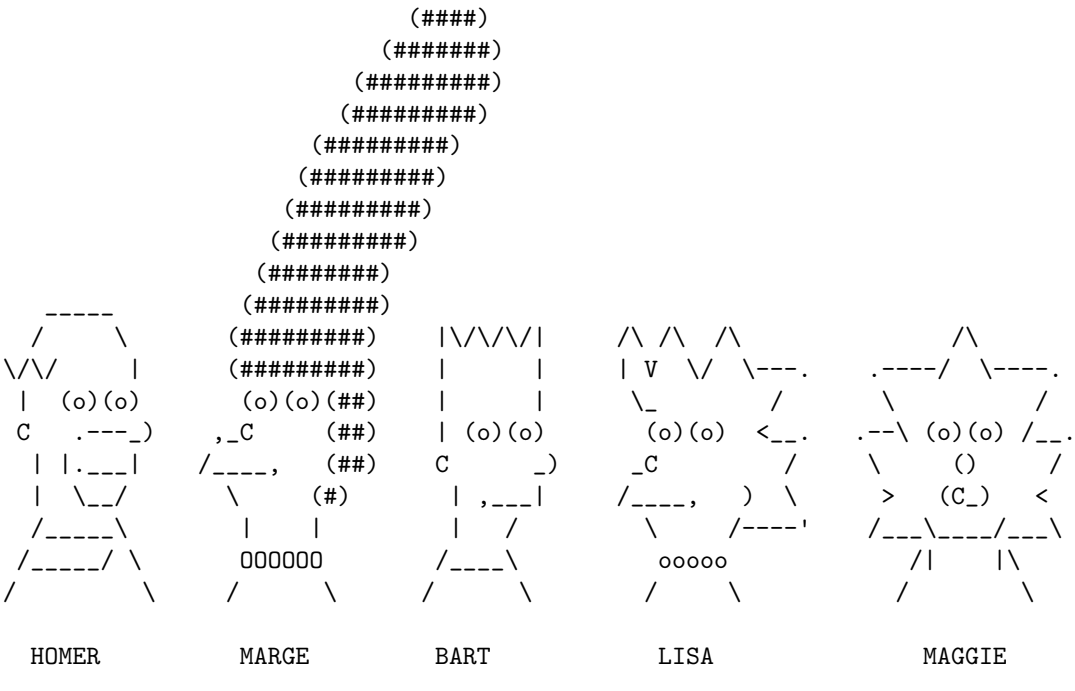
This elegant feature is new to Python 3.6. It makes code much more readable. These strings make formatting things a piece of cake. Suppose we have variables `x` and `y` and we want to print `x + y`. There are two ways to do this. Which do you prefer?

```
>>> x = 5
>>> y = 3
>>> print(str(x) + " + " + str(y) + " = " + str(x + y))
5 + 3 = 8
>>> print(f"{x} + {y} = {x + y}")
5 + 3 = 8
```

The second is accomplished with an f-string. The expressions inside of the curly braces are evaluated and then turned into strings and dropped neatly into place. Don't forget the `f` in front! This is not available in Python 2. We will use f-strings throughout.

For the first solution, did you forget a + sign and get a nasty error? Typing that first `print` statement is pure drudgery.

Programming Exercise Use a raw triple-quoted string to print this.



3.3 Formatting Numbers

Sometimes it's convenient to have control over how numbers, especially floats, are displayed. Consider this mess.

```
>>> from math import cos, radians
>>> for k in range(10):
...     print(cos(radians(k)))
...
1.0
0.9998476951563913
0.9993908270190958
0.9986295347545738
0.9975640502598242
0.9961946980917455
0.9945218953682733
0.992546151641322
0.9902680687415704
0.9876883405951378
>>>
```

Maybe we want just 4 places beyond the decimal and to have 1.0000 instead of 1.0. Check this out.

```
>>> for k in range(10):
...     print(f"{cos(radians(k)):.4f}")
...
1.0000
0.9998
0.9994
0.9986
0.9976
0.9962
0.9945
0.9925
0.9903
0.9877
```

The `:.4f` says, "round at four beyond the decimal point." Now let's do this. The — on either side shows how the number is formatted in its space.

```
>>> for k in range(10):
...     print(f"|{cos(radians(k)):10.4f}|")
...
|    1.0000|
```

```
|    0.9998|
|    0.9994|
|    0.9986|
|    0.9976|
|    0.9962|
|    0.9945|
|    0.9925|
|    0.9903|
|    0.9877|
```

The space for the number is 10 characters wide; this is a minimum. Note that each number is right-justified within its space.

Here we change the justification.

```
>>> for k in range(10):
...     print(f"|{cos(radians(k)):<10.4f}|")
...
|1.0000    |
|0.9998    |
|0.9994    |
|0.9986    |
|0.9976    |
|0.9962    |
|0.9945    |
|0.9925    |
|0.9903    |
|0.9877    |
>>> for k in range(10):
...     print(f"|{cos(radians(k)):^10.4f}|")
...
|  1.0000  |
|  0.9998  |
|  0.9994  |
|  0.9986  |
|  0.9976  |
|  0.9962  |
|  0.9945  |
|  0.9925  |
|  0.9903  |
|  0.9877  |
```

What happens if you don't give enough space? Observe the number you put before the space is a *minimum* width.

```
>>> for k in range(10):
...     print(f"|{cos(radians(k)):^2.4f}|")
```

```
...
|1.0000|
|0.9998|
|0.9994|
|0.9986|
|0.9976|
|0.9962|
|0.9945|
|0.9925|
|0.9903|
|0.9877|
```

Your chem teacher will be pleased by your tasteful use of significant digits.

Programming Exercises

1. Place this `print(f"{k:>2d}{k*k:>3d}")` in a for loop. What does it look like.
2. How do you left-justify the numbers in the columns? center?
3. It appears that `f` is for float and `d` is for integer (why? goes back to C and `%d`). How abut `s`. Make a variable `a` and have it point at a string. Then try this: `print(f"a:10s—")`—. Try to see how to change the justification and see if you can center it, too.
4. Do this assignment, `x, y, z = 3,4,5`. Now try this: `print(f"{x = }, {y = }, {z = }")`. What useful thing happens?

4 Sequence Types

Have you ever had a nocturnal itch to store a bunch of related items together? F'rinstance, 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, or possibly 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 as you stumble about in the morning from a lack of sleep!

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*; data structures 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. All three of these things have something in common. They store objects cheek-by-jowl "in a row."

- `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*. However, a variable is not wedded to a string; it can be assigned to different string.
- `list` This is the list type which stores a sequence of Python objects. This sequence of objects (order counts) 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*. A list literal is enclosed inside of square brackets `[... ]`; inside this you put a comma-separated list of variables, expressions, or literals. An example of a list literal is something like this `[1, 2, "cat", "cow"]`.
- `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. Tuples look like lists but they are clothed in `( ... )` instead of `[ ... ]`.

Let us show a couple of interesting things here. First, Python acknowledges their types.

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

Second, even though the contents of `moose` and `regalis` are identical, they are of different types so they fail the "species test" and are found to be unequal.

```
>>> moose == regalis
False
>>>
```

Observe that a list can never equal a tuple, and vice versa. Objects of different types are never equal; this is the species test at work. 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)
```

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. For now, you might want to think about your refrigerator freshly stocked with some cold, creamy bottles of Maple View chocolate milk. This is mutable. You might have a roommate. What do you think can happen?

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 give some grammatical reminders 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. Lists and tuples can contain Python objects of any types; we say that these two types are heterogeneous sequences.

Here we demonstrate `+` and `len`.

```
>>> 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
>>>
```

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.

A Roundup of Useful Stuff 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`.
- `in` This is an infix binary operator; the left operand is a Python object, the right-hand object is a Python sequence. 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 sequences 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 tuple and a list. Multiply them by a positive integers. What happens?
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! Can you Google to find out how to sort a list in Python?

5 On the Importance of Type

We've now seen the action of a variety of operators on numbers. Recall that 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)`, or by using an f-string.

The * operator has a useful behavior when it operates on an integer and a sequence. Look at this session. This is a re-run if you have done all of the exercises.

```
>>> 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. What do you see?

2. You can see the numerical (ASCII) value for any character by placing it in a one-character 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.

6 Text Editors

In preparation for writing programs, you will need to install a text editor. These programs edit plain text. Word processor files contain all sorts of invisible information; open one of these with Python and a cascade of error messages will be spewed at you.

The VSCode editor is an excellent and powerful choice. It is free and it runs on all platforms.

7 Making your first Python Program `hello.py`

So far, we have only 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 your text editor.

```
#!/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 (Yes, that Mac is a UNIX computer).

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.

7.1 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 serous plumbing here we don't have to plumb.

7.2 Running Your Program

Now, thank your lucky stars, and save the program in your text editor. To run the program, do this in a command window.

```
 $ python3 hello.py
```

When you are done you will see this.

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

The action of our program is to take the string `"Hello World!"` and to put it to `stdout`. The command `python` behaves as a UNIX (or Windoze) command. It takes as an argument the name of the file containing the program you wish to run. You may, if you are using MaC or Linux, 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 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) on MacOSX or Linux 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.

Note for VSCode Users Take a peek in Appendix A. You can run your program by hitting the little triangle button. Appendix A will explain to you succinctly and in detail how it all works.

8 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 your 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:  21 June 2022
#    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.

9 Useful Formatting Tools

The `print` function in Python3 has some useful features that can be helpful to you. You can print any comma-separated list of objects. Bear witness.

```
>>> a = 1
>>> b = 2
>>> c = "cows"
>>> d = True
>>> print(a,b,c,d)
1 2 cows True
>>>
```

When printed, the items are separated by a space. You can, however, separate them with any string you would like to, as in this example here.

```
>>> print(a,b,c,d, sep=" MOO ")
1 MOO 2 MOO cows MOO True
```

The `print` function automatically places a newline when it is done. Want something else? Here is a means to that `end`.

```
>>> print(a,b,c,d, end=" No newline!")
1 2 cows True No newline!>>>
```

The optional arguments to `print` are called *keyword arguments*. The `sep` argument has a default value of `" "` and the `end` argument has a default value of `\n`.

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.
"""
```

You can even make it a raw string! Here is an ASCII art cow being printed.

```
>>> cow = r"""
... Art by Hayley Jane Wakenshaw

             /)  (\
        .-._((,~~.))_.-,
         `=.   99   ,='
           / ,o~~o. \
          { { .__. } }
           ) `~~~\' (
          /`-._  _\.-\
         /         )  \
       ,-X        #   X-.
hjw   /   \          /   \
```

```
     (     )| |  | |(     )
      \   / | |  | | \   /
       \_(.-( )--( )-.)_/
       /_,\ ) /  \ ( /._\
           /_,\  /._\
"""
>>> print(cow)

Art by Hayley Jane Wakenshaw

             /)  (\
         .-._((,~~.))_.-,
          `=.   99   ,='
            / ,o~~o. \
           { { .__. } }
            ) `~~~\' (
           /`-._  _\.-\
          /         )  \
        ,-X        #   X-.
hjw    /   \          /   \
     (     )| |  | |(     )
      \   / | |  | | \   /
       \_(.-( )--( )-.)_/
       /_,\ ) /  \ ( /._\
           /_,\  /._\

>>>
```

Moooooooo!!!! Credit goes to the ASCII Art Archive and Haley Jane Wakeenshaw.
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`?

10 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 locations of the objects they point at on the heapas "definitions." This table is stored on the stack; it's how the stack keeps track of varables and their pointees..

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 or list slices.

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.

```
x → 5
y → 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 and substitute them in where indicated.

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

10.1 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 at `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).

You are also familiar with this phenomenon from your Miss Wormwood days, PEMDAS. This is the algebraic order of operations: parentheses, exponents Multiplication and division (from left to right) and then addition and subtraction from left to right. We see that the arithmetic operations associate from left to right.

Assignment goes backwards. It associates from right to left. It is lower in precedence than any other arithmetic operation. So, let's see what happens when `a = b = 5` is encountered. First of all, the first two items `a` and `b` are lvalues, because they are on the left side of an assignment. They do not have to be evaluated, because lvalues are necessarily "atomic" expressions.

Once Python sees the second assignment, it evaluates the expression, in this case 5, to its right. It then makes `b` point at the integer object 5. Now the expression `b = 5` is evaluated. What value comes out of it? The result of evaluating the rvalue of the assignment, which is 5. So `b = 5` returns a 5 to the `a =` and then `a` now points at (likely the same) integer object 5.

10.2 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.

10.3 Python is a strongly, dynamically typed language.

Variables are typeless: the objects they point at actually have type. The type of a variable's 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 annie–mule!" Next we point x at a string. Since a string has behavior `upper()`, Python happily complies with our wish.

11 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.

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

Python will hiss is you attempt to use an index that is out-of-bounds. Here is a very common n00b 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. Indices are point to the character immediately to their right. This is extremely handy when we talk about taking slices from a sequence.

11.2 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 in a string slice. Here we get the slice of `x` starting at index 5 and ending at index 7.

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

Notice how the design of indices make things spiffy.

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

11.3 The `in` Keyword

Here we show this keyword at work.

```
>>>  "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. Take note of the fact that `in` is a language keyword! Type it in a `.py` file in a text editor

```
>>>  "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`.

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.

```
>>> 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 subsequence or a single item in a 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 `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` → `"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` → `"moo cow"`

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

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.

12 The Pointing Relationship

This section gives a second, deeper look at the pointing relationship between variables and objects.

To begin, let's talk a little bit about RAM. 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 process's virtual address space is like a little, private computer for your process that other process may not see or tamper with. It provides you happy little program with the illusion it is operating on its own computer.

What follows is an incomplete model, but it provides an excellent insight into how Python (and other languages as well) actually work.

As part of this process, a fixed-sized chunk of memory is devoted to two important functions: the stack, which manages function calls and maintains tables of variables and where they are pointing.

The other important piece of memory is the heap, which is your program's object warehouse. In Python, all objects are stored on the heap. Each byte of

memory, incuding the stack and the heap, has a unique memory address.

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.

To create a variable of integer type in C, you enter this.

```
int x = 42;
```

In C, when a variable is created, it holds the value 42 directly as a bit-pattern. The `int` type is a 32 bit integer, so it stores the bit pattern `00000000000000000000000000101010`. So, in C the value is stored directly in a little box of memory sufficient to hold 32 bits. In this case, the memory is allocated on the stack inside of the function's stack frame that created it. The variable has type; this is necessary because there are only 32 bits of storage for its value. The variable's type is an immutable property of that variable.

Now consider Python. To create a variable and have it point at the value 42, here is what happens. First, the integer object representing 42 is put on the heap. Secondly, the variable `x` gets the memory address of the integer object. Memory address are just integers; so `x` is just storing an integer. The variable `x`, along with the memory address it is storing is kept in the stack frame of the function that created that variable. If the variable was created in the global scope, it goes in the global frame.

This is the magic of *indirection*. Python variables refer to their objects via a memory address that is assigned to them. This is the "pointing relationship" we have been speaking of.

So, in creating varable, the stack frame holding it inscribes it in its symbol table, along with the memory address of its object. Note that this address is just an integer.

You have seen that 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.

Assigning the Bible to `x` causes a copy of the Bible (which you probably read from a file) to be placed on the heap. Then, the memory address where the Bible starts is handed to `x`, which stores it on the stack. Because `x` only stores a memory address, and all memory addresses are just integers, this arabesque is possible.

You have seen indirection in your everyday life. Think about 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).

Objects are quite complex. They have data (such as 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.

A Reprise 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 on the heap. It then gives `x` the heap address where we can find the Bible.

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 heap of your Python session. If your session runs out of heap 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.

13 Mutability and its Dangers

Consider this innocent little act.

```
>>> cats = ["burmese", "siamese", "russian blue"]
>>> meowers = cats
>>> cats
['burmese', 'siamese', 'russan blue']
>>> meowers
['burmese', 'siamese', 'russan 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 burmese cats to calico.

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

Prepare yourself. . .

```
>>> meowers
['calico', 'siamese', 'russan 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, so the memory address of `meowers` got copied into `cats`. 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 is 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 `dogs`, 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? Add more dogs to the list and repeat this.

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 `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?

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]
>>> xtuple = (1,2,3)
>>> ylist = xlist
>>> ytuple = xtuple
>>> xlist is ylist
True
>>> xtuple is ytuple
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]
>>> xtuple += (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 `xtuple` and `ytuple`.

```
>>> xtuple is ytuple
False
>>> xtuple
(1, 2, 3, 4)
>>> ytuple
(1, 2, 3)
```

A new object is constructed for `xtuple` and `ytuple` 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.

14 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 you 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 aliases 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 is 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?

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.

15 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.

16 Terminolgy Roundup

- **Boolean** This refers to the true/false data type. The Boolean objects in Python are `True` and `False`
- **behavior** This is the property of an object that describes what an object does.
- **binary operator** This combines two objects of the same type and returns a third object
- **character string** This is a sequence of characers, a blob of text.
- **command window** On a Mac or UNIX machine, it refers to the terminal. In a Windoze box, it refers to a PowerShell or cmd window.
- **comment box** This is a comment you place at the top of a program to tell the user how to run it.
- **concatenate** To glue together, as in strings.

- **data structure** This is a container that can hold many objects under a single name.
- **document** Documenting a program tells other programmers using your code how to properly use and run it.
- **entries** These are the items in a list or a tuple.
- **evaluatione** This is the process of extracting meaning from an expresson. *syn:* parse.
- **expression** This is a combination of objects and operators. Note that expressions have certain grammatical rules.
- **fetch** To read in values from an a symbol table.
- **float** This is Python's floating-point numerical type. It is an IEEE 64-bit 754 flaoting point number.
- **garbage collector** This is Python's janitor; it deallocates memory being used by orphaned objects.
- **immutable** An object is immutiable if it cannot be changed in memory subsequent to its creation. Strings, integers, and Booleans are all immutable objects in Python.
- **immutable sequence type** This is a tuple.
- **indirection** This is the process by which variables refer to their objects. Variables store a heap memory address which is really just an integer. The object itself resides the heap, whilst the variable and the address it stores lives in the stack.
- **indes** Indices are integers that denumerate the elements of a sequence. They begin a 0 and they live *between* the sequence's entries. The last index points off the end of the sequence.
- **infix** A binary operator is infix if it occurs between its operands
- **items** Synonom of entries; list or tuple elements.
- **keyword arguments** These are optional arguments that occur at the end of a function's argument list. They are specifed by name. Examples inclue `sep` and `end` for the `print` function.
- **lvalue** An lvalue refers to writeable memory. Varibles are lvalues because you can assign value to them.
- **object** This is a named region of storage in a computer's memory.
- **operand** This is something being acted upon by an operator.
- **operator** Operators are commands that operate on objects. They are part of expressions.
- **order of operations** This is an order of precedence for operators (who goes first?) PEMDAS that you learned in Algebra I is an example of an order of operations.

- **orphan** An object is orphaned if no variable is pointing at it.
- **pooling** This refers to the cacheing of objects in memory for later re-use. Strings and small integers are pooled in Python.
- **prefix** An operator is a prefix operator if it occurs before its operand.
- **program** This is a set of instructions to be carried out by a computer.
- **programming language** This is a language for communicating with a computer.
- **prompt** This is a symbol in a command window indicating your computer is ready to receive commands.
- **raw strings** A raw string is created in Python by preceding the first quote with an `r`.
- **relational** This describes a boolean-valued operator that compares two comparable objects. Examples include `<`, `==`, and friends.
- **rvalue** This is the result of evaluating a valid expression
- **slice** This is a piece of a sequence between two indices.
- **snake notation** This refers to using underscores to separate words in variable names. Example: `number_of_rabid_cows`.
- **stack** This is the area of memory where variables and the heap addresses they store are kept.
- **state** This is what an object knows.
- **string intern pool** This is an area of memory in the heap where strings are stored. Strings kept here are not orphaned so they can be recycled.
- **heap** This is the area of memory where objects are stored.
- **triple-quoted string** This is a string surronded by sets of three quotes; such strings can span many lines.
- **type** This is the "species" of an object.
- **variable** This is a name that can point at an object