**Python**

1. **Variables, Expressions & Statements**

**Variables**

* One of the most powerful features of a programming language is the ability to manipulate **variables**. A variable is a name that refers to a value.
* The **assignment statement** creates new variables and gives them values:

>>> message = “Hello, World"   
>>> n = 17   
>>> pi = 3.14159

* This example makes three assignments:
  + The first assigns the string "Hello,World" to a new variable named message.
  + The second gives the integer 17 to n
  + The third gives the floating-point number 3.14159 to pi.
* Notice that the first statement uses double quotes to enclose the string. In general, single and double quotes do the same thing, but if the string contains a single quote (or an apostrophe, which is the same character), you have to use double quotes to enclose it.
* The print statement also works with variables.

>>> print message   
Hello, World   
>>> print n   
17   
>>> print pi   
3.14159

* In each case the result is the value of the variable. Variables also have types; again, we can ask the interpreter what they are.

>>> type(message)   
<type 'str'>   
>>> type(n)   
<type 'int'>   
>>> type(pi)   
<type 'float'>

The type of a variable is the type of the value it refers to.

**Variable names and keywords**

* Programmers generally choose names for their variables that are meaningful  they document what the variable is used for.
* Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter. Although it is legal to use uppercase letters, by convention we don't. If you do, remember that case matters. Bruce and bruce are different variables.
* The underscore character (\_) can appear in a name. It is often used in names with multiple words, such as my\_name or price\_of\_tea\_in\_china.
* If you give a variable an illegal name, you get a syntax error:

>>> 76trombones = 'big parade'   
SyntaxError: invalid syntax   
>>> more$ = 1000000   
SyntaxError: invalid syntax   
>>> class = 'Computer Science 101'   
SyntaxError: invalid syntax

* 76trombones is illegal because it does not begin with a letter. more$ is illegal because it contains an illegal character, the dollar sign. But what's wrong with class?
* It turns out that class is one of the Python **keywords**. Keywords define the language's rules and structure, and they cannot be used as variable names.
* Python has twenty-nine keywords:
  + and
  + def
  + exec
  + if
  + not
  + return
  + assert
  + del
  + finally
  + import
  + or
  + try
  + break
  + elif
  + for
  + in
  + pass
  + while
  + class
  + else
  + from
  + is
  + print
  + yield
  + continue
  + except
  + global
  + lambda
  + raise
* You might want to keep this list handy. If the interpreter complains about one of your variable names and you don't know why, see if it is on this list.

**Statements**

* A statement is an instruction that the Python interpreter can execute. We have seen two kinds of statements: **print** and **assignment**.
* When you type a statement on the command line, Python executes it and displays the result, if there is one. The result of a print statement is a value. Assignment statements don't produce a result.
* A script usually contains a sequence of statements. If there is more than one statement, the results appear one at a time as the statements execute.
* For example, the script

print 1   
x = 2   
print x

produces the output

1   
2

Again, the assignment statement produces no output.

**Evaluating expressions**

* An expression is a combination of values, variables, and operators. If you type an expression on the command line, the interpreter evaluates it and displays the result:

>>> 1 + 1   
2

* Although expressions contain values, variables, and operators, not every expression contains all of these elements. A value all by itself is considered an expression, and so is a variable.

>>> 17   
17   
>>> x   
2

* Confusingly, evaluating an expression is not quite the same thing as printing a value.

>>> message = 'Hello, World!'   
>>> message   
'Hello, World!'   
>>> print message   
Hello, World!

* When the Python interpreter displays the value of an expression, it uses the same format you would use to enter a value.
* In the case of strings, that means that it includes the quotation marks. But if you use a print statement, Python displays the contents of the string without the quotation marks.

**Operators and operands**

* **Operators** are special symbols that represent computations like addition and multiplication.
* The values the operator uses are called **operands**.
* The following are all legal Python expressions whose meaning is more or less clear:

20+32   hour-1   hour\*60+minute   minute/60   5\*\*2   (5+9)\*(15-7)

* The symbols +, -, and /, and the use of parenthesis for grouping, mean in Python what they mean in mathematics. The asterisk (\*) is the symbol for multiplication, and \*\* is the symbol for exponentiation.
* When a variable name appears in the place of an operand, it is replaced with its value before the operation is performed.
* Addition, subtraction, multiplication, and exponentiation all do what you expect, but you might be surprised by division. The following operation has an unexpected result:

>>> minute = 59   
>>> minute/60   
0

* The value of minute is 59, and in conventional arithmetic 59 divided by 60 is 0.98333, not 0. The reason for the discrepancy is that Python is performing **integer division**.
* When both of the operands are integers, the result must also be an integer, and by convention, integer division always rounds *down*, even in cases like this where the next integer is very close.
* A possible solution to this problem is to calculate a percentage rather than a fraction:

>>> minute\*100/60   
98

* Again the result is rounded down, but at least now the answer is approximately correct.

**Order of operations**

* When more than one operator appears in an expression, the order of evaluation depends on the **rules of precedence**.
* Python follows the same precedence rules for its mathematical operators that mathematics does. The acronym **PEMDAS** is a useful way to remember the order of operations:
  + **P**arentheses have the highest precedence and can be used to force an expression to evaluate in the order you want. Since expressions in parentheses are evaluated first, 2 \* (3-1) is 4, and (1+1)\*\*(5-2) is 8. You can also use parentheses to make an expression easier to read, as in (minute \* 100) / 60, even though it doesn't change the result.
  + **E**xponentiation has the next highest precedence, so 2\*\*1+1 is 3 and not 4, and 3\*1\*\*3 is 3 and not 27.
  + **M**ultiplication and **D**ivision have the same precedence, which is higher than **A**ddition and **S**ubtraction, which also have the same precedence. So 2\*3-1 yields 5 rather than 4, and 2/3-1 is -1, not 1 (remember that in integer division, 2/3=0).
  + Operators with the same precedence are evaluated from left to right. So in the expression minute\*100/60, the multiplication happens first, yielding 5900/60, which in turn yields 98. If the operations had been evaluated from right to left, the result would have been 59\*1, which is 59, which is wrong.

**Operations on strings**

* In general, you cannot perform mathematical operations on strings, even if the strings look like numbers. The following are illegal (assuming that message has type string):

message-1   'Hello'/123   message\*'Hello'   '15'+2

* Interestingly, the + operator does work with strings, although it does not do exactly what you might expect. For strings, the + operator represents **concatenation**, which means joining the two operands by linking them end-to-end. For example:

fruit = 'banana'   
bakedGood = ' nut bread'   
print fruit + bakedGood

* The output of the above program is banana nut bread. The space before the word nut is part of the string, and is necessary to produce the space between the concatenated strings.
* The \* operator also works on strings; it performs repetition. For example, 'Fun'\*3 is 'FunFunFun'. One of the operands has to be a string; the other has to be an integer.
* On one hand, this interpretation of + and \* makes sense by analogy with addition and multiplication. Just as 4\*3 is equivalent to 4+4+4, we expect 'Fun'\*3 to be the same as 'Fun'+'Fun'+'Fun', and it is. On the other hand, there is a significant way in which string concatenation and repetition are different from integer addition and multiplication. Can you think of a property that addition and multiplication have that string concatenation and repetition do not?

**Composition**

* So far, we have looked at the elements of a program  variables, expressions, and statements  in isolation, without talking about how to combine them.
* One of the most useful features of programming languages is their ability to take small building blocks and **compose** them.
* For example, we know how to add numbers and we know how to print; it turns out we can do both at the same time:

>>>  print 17 + 3   
20

* In reality, the addition has to happen before the printing, so the actions aren't actually happening at the same time.
* The point is that any expression involving numbers, strings, and variables can be used inside a print statement. You've already seen an example of this:

print 'Number of minutes since midnight: ', hour\*60+minute

* You can also put arbitrary expressions on the right-hand side of an assignment statement:

percentage = (minute \* 100) / 60

* This ability may not seem impressive now, but you will see other examples where composition makes it possible to express complex computations neatly and concisely.
* **Warning**: There are limits on where you can use certain expressions. For example, the left-hand side of an assignment statement has to be a *variable* name, not an expression. So, the following is illegal: minute+1 = hour.

**Comments**

* As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.
* For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called **comments**, and they are marked with the # symbol:

# compute the percentage of the hour that has elapsed   
percentage = (minute \* 100) / 60

* In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

percentage = (minute \* 100) / 60     # caution: integer division

* Everything from the # to the end of the line is ignored  it has no effect on the program.
* The message is intended for the programmer or for future programmers who might use this code.

1. **Functions**

**Function calls**

* You have already seen one example of a **function call**:

>>> type("32")   
<type 'str'>

* The name of the function is type, and it displays the type of a value or variable.
* The value or variable, which is called the **argument** of the function, has to be enclosed in parentheses.
* It is common to say that a function "takes" an argument and "returns" a result. The result is called the **return value**.
* Instead of printing the return value, we could assign it to a variable:

>>> betty = type("32")   
>>> print betty   
<type 'str'>

* As another example, the id function takes a value or a variable and returns an integer that acts as a unique identifier for the value:

>>> id(3)   
134882108   
>>> betty = 3   
>>> id(betty)   
134882108

* Every value has an id, which is a unique number related to where it is stored in the memory of the computer. The id of a variable is the id of the value to which it refers.

**Type conversion**

* Python provides a collection of built-in functions that convert values from one type to another. The int function takes any value and converts it to an integer, if possible, or complains otherwise:

>>> int("32")   
32   
>>> int("Hello")   
ValueError: invalid literal for int(): Hello

* int can also convert floating-point values to integers, but remember that it truncates the fractional part:

>>> int(3.99999)   
3   
>>> int(-2.3)   
-2

* The float function converts integers and strings to floating-point numbers:

>>> float(32)   
32.0   
>>> float("3.14159")   
3.14159

* Finally, the str function converts to type string:

>>> str(32)   
'32'   
>>> str(3.14149)   
'3.14149'

* It may seem odd that Python distinguishes the integer value 1 from the floating-point value 1.0. They may represent the same number, but they belong to different types. The reason is that they are represented differently inside the computer.

**Type coercion**

* Now that we can convert between types, we have another way to deal with integer division.
* Suppose we want to calculate the fraction of an hour that has elapsed. The most obvious expression, minute / 60, does integer arithmetic, so the result is always 0, even at 59 minutes past the hour.
* One solution is to convert minute to floating-point and do floating-point division:

>>> minute = 59   
>>> float(minute) / 60   
0.983333333333

* Alternatively, we can take advantage of the rules for automatic type conversion, which is called **type coercion**. For the mathematical operators, if either operand is a float, the other is automatically converted to a float:

>>> minute = 59   
>>> minute / 60.0   
0.983333333333

* By making the denominator a float, we force Python to do floating-point division.

**Math functions**

* Python has a math module that provides most of the familiar mathematical functions. A **module** is a file that contains a collection of related functions grouped together.
* Before we can use the functions from a module, we have to import them:

>>> import math

* To call one of the functions, we have to specify the name of the module and the name of the function, separated by a dot, also known as a period. This format is called **dot notation**.

>>> decibel = math.log10 (17.0)   
>>> angle = 1.5   
>>> height = math.sin(angle)

* The first statement sets decibel to the logarithm of 17, base 10. There is also a function called log that takes logarithm base e.
* The third statement finds the sine of the value of the variable angle. sin and the other trigonometric functions (cos, tan, etc.) take arguments in radians. To convert from degrees to radians, divide by 360 and multiply by 2\*pi. For example, to find the sine of 45 degrees, first calculate the angle in radians and then take the sine:

>>> degrees = 45   
>>> angle = degrees \* 2 \* math.pi / 360.0   
>>> math.sin(angle)   
0.707106781187

* The constant pi is also part of the math module. If you know your geometry, you can check the previous result by comparing it to the square root of two divided by two:

>>> math.sqrt(2) / 2.0   
0.707106781187

**Composition**

* Just as with mathematical functions, Python functions can be composed, meaning that you use one expression as part of another. For example, you can use any expression as an argument to a function:

>>> x = math.cos(angle + math.pi/2)

* This statement takes the value of pi, divides it by 2, and adds the result to the value of angle. The sum is then passed as an argument to the cos function.
* You can also take the result of one function and pass it as an argument to another:

>>> x = math.exp(math.log(10.0))

* This statement finds the log base e of 10 and then raises e to that power. The result gets assigned to x.

**Adding new functions**

* So far, we have only been using the functions that come with Python, but it is also possible to add new functions.
* Creating new functions to solve your particular problems is one of the most useful things about a general-purpose programming language.
* In the context of programming, a **function** is a named sequence of statements that performs a desired operation. This operation is specified in a **function definition**.
* The functions we have been using so far have been defined for us, and these definitions have been hidden. This is a good thing, because it allows us to use the functions without worrying about the details of their definitions.
* The syntax for a function definition is:

def NAME( LIST OF PARAMETERS ):   
  STATEMENTS

* You can make up any names you want for the functions you create, except that you can't use a name that is a Python keyword. The list of parameters specifies what information, if any, you have to provide in order to use the new function.
* There can be any number of statements inside the function, but they have to be indented from the left margin.
* The first couple of functions we are going to write have no parameters, so the syntax looks like this:

def newLine():   
  print

* This function is named newLine. The empty parentheses indicate that it has no parameters. It contains only a single statement, which outputs a newline character. (That's what happens when you use a print command without any arguments.)
* The syntax for calling the new function is the same as the syntax for built-in functions:

print "First Line."   
newLine()   
print "Second Line."

The output of this program is:

First line.   
  
Second line.

* Notice the extra space between the two lines. What if we wanted more space between the lines? We could call the same function repeatedly:

print "First Line."   
newLine()   
newLine()   
newLine()   
print "Second Line."

* Or we could write a new function named threeLines that prints three new lines:

def threeLines():   
  newLine()   
  newLine()   
  newLine()   
  
print "First Line."   
threeLines()   
print "Second Line."

* This function contains three statements, all of which are indented by two spaces. Since the next statement is not indented, Python knows that it is not part of the function.
* You should notice a few things about this program:

1. You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
2. You can have one function call another function; in this case threeLines calls newLine.

* So far, it may not be clear why it is worth the trouble to create all of these new functions. Actually, there are a lot of reasons, but this example demonstrates two:
  + Creating a new function gives you an opportunity to name a group of statements. Functions can simplify a program by hiding a complex computation behind a single command and by using English words in place of arcane code.
  + Creating a new function can make a program smaller by eliminating repetitive code. For example, a short way to print nine consecutive new lines is to call threeLines three times.

**Parameters and arguments**

* Some of the built-in functions you have used require arguments, the values that control how the function does its job.
* For example, if you want to find the sine of a number, you have to indicate what the number is. Thus, sin takes a numeric value as an argument.
* Some functions take more than one argument. For example, pow takes two arguments, the base and the exponent. Inside the function, the values that are passed get assigned to variables called **parameters**.
* Here is an example of a user-defined function that has a parameter:

def printTwice(bruce):   
  print bruce, bruce

* This function takes a single argument and assigns it to a parameter named bruce.
* The value of the parameter (at this point we have no idea what it will be) is printed twice, followed by a newline.
* The name bruce was chosen to suggest that the name you give a parameter is up to you, but in general, you want to choose something more illustrative than bruce.
* The function printTwice works for any type that can be printed:

>>> printTwice('Spam')   
Spam Spam   
>>> printTwice(5)   
5 5   
>>> printTwice(3.14159)   
3.14159 3.14159

* In the first function call, the argument is a string. In the second, it's an integer. In the third, it's a float.
* The same rules of composition that apply to built-in functions also apply to user-defined functions, so we can use any kind of expression as an argument for printTwice:

>>> printTwice('Spam'\*4)   
SpamSpamSpamSpam SpamSpamSpamSpam   
>>> printTwice(math.cos(math.pi))   
-1.0 -1.0

* As usual, the expression is evaluated before the function is run, so printTwice prints SpamSpamSpamSpam SpamSpamSpamSpam instead of 'Spam'\*4 'Spam'\*4.
* We can also use a variable as an argument:

>>> michael = 'Eric, the half a bee.'   
>>> printTwice(michael)   
Eric, the half a bee. Eric, the half a bee.

* Notice something very important here. The name of the variable we pass as an argument (michael) has nothing to do with the name of the parameter (bruce). It doesn't matter what the value was called back home (in the caller); here in printTwice, we call everybody bruce.

**Variables and parameters are local**

* When you create a **local variable** inside a function, it only exists inside the function, and you cannot use it outside. For example:

def catTwice(part1, part2):   
  cat = part1 + part2   
  printTwice(cat)

* This function takes two arguments, concatenates them, and then prints the result twice. We can call the function with two strings:

>>> chant1 = "Pie Jesu domine, "   
>>> chant2 = "Dona eis requiem."   
>>> catTwice(chant1, chant2)   
Pie Jesu domine, Dona eis requiem. Pie Jesu domine, Dona eis requiem.

* When catTwice terminates, the variable cat is destroyed. If we try to print it, we get an error:

>>> print cat   
NameError: cat

* Parameters are also local.
* For example, outside the function printTwice, there is no such thing as bruce. If you try to use it, Python will complain.

**Functions with results**

* You might have noticed by now that some of the functions we are using, such as the math functions, yield results.
* Other functions, like newLine, perform an action but don't return a value. That raises some questions:
  + What happens if you call a function and you don't do anything with the result (i.e., you don't assign it to a variable or use it as part of a larger expression)?
  + What happens if you use a function without a result as part of an expression, such as newLine() + 7?
  + Can you write functions that yield results, or are you stuck with simple function like newLine and printTwice?

1. **Conditionals & Recursion**

### The modulus operator

* The **modulus operator** works on integers (and integer expressions) and yields the remainder when the first operand is divided by the second.
* In Python, the modulus operator is a percent sign (%). The syntax is the same as for other operators:

>>> quotient = 7 / 3   
>>> print quotient   
2   
>>> remainder = 7 % 3   
>>> print remainder   
1

* So 7 divided by 3 is 2 with 1 left over.
* The modulus operator turns out to be surprisingly useful.
* For example, you can check whether one number is divisible by another  if x % y is zero, then x is divisible by y.
* Also, you can extract the right-most digit or digits from a number.
* For example, x % 10 yields the right-most digit of x (in base 10). Similarly x % 100 yields the last two digits.

### Boolean expressions

* A **boolean expression** is an expression that is either true or false.
* One way to write a boolean expression is to use the operator ==, which compares two values and produces a boolean value:

>>> 5 == 5   
True   
>>> 5 == 6   
False

* In the first statement, the two operands are equal, so the value of the expression is True; in the second statement, 5 is not equal to 6, so we get False. True and False are special values that are built into Python.
* The == operator is one of the **comparison operators**; the others are:

      x != y               # x is not equal to y   
      x > y                # x is greater than y   
      x < y                # x is less than y   
      x >= y               # x is greater than or equal to y   
      x <= y               # x is less than or equal to y

* Although these operations are probably familiar to you, the Python symbols are different from the mathematical symbols.
* A common error is to use a single equal sign (=) instead of a double equal sign (==).
* Remember that = is an assignment operator and == is a comparison operator. Also, there is no such thing as =< or =>.

### Logical operators

* There are three **logical operators**: and, or, and not.
* The semantics (meaning) of these operators is similar to their meaning in English.
* For example, x > 0 and x < 10 is true only if x is greater than 0 *and* less than 10.
* n%2 == 0 or n%3 == 0 is true if *either* of the conditions is true, that is, if the number is divisible by 2 *or* 3.
* Finally, the not operator negates a boolean expression, so not(x > y) is true if (x > y) is false, that is, if x is less than or equal to y.
* Strictly speaking, the operands of the logical operators should be boolean expressions, but Python is not very strict. Any nonzero number is interpreted as "true."

>>>  x = 5   
>>>  x and 1   
1   
>>>  y = 0   
>>>  y and 1   
0

* In general, this sort of thing is not considered good style. If you want to compare a value to zero, you should do it explicitly.

### Conditional execution

* In order to write useful programs, we almost always need the ability to check conditions and change the behavior of the program accordingly.
* **Conditional statements** give us this ability. The simplest form is the if statement:

if x > 0:   
  print "x is positive"

* The boolean expression after the if statement is called the **condition**. If it is true, then the indented statement gets executed. If not, nothing happens.
* Like other compound statements, the if statement is made up of a header and a block of statements:

HEADER:   
  FIRST STATEMENT   
  ...   
  LAST STATEMENT

* The header begins on a new line and ends with a colon (:).
* The indented statements that follow are called a **block**.
* The first unindented statement marks the end of the block.
* A statement block inside a compound statement is called the **body** of the statement.
* There is no limit on the number of statements that can appear in the body of an if statement, but there has to be at least one.
* Occasionally, it is useful to have a body with no statements (usually as a place keeper for code you haven't written yet).
* In that case, you can use the pass statement, which does nothing.

### Alternative execution

* A second form of the if statement is alternative execution, in which there are two possibilities and the condition determines which one gets executed. The syntax looks like this:

if x%2 == 0:   
  print x, "is even"   
else:   
  print x, "is odd"

* If the remainder when x is divided by 2 is 0, then we know that x is even, and the program displays a message to that effect.
* If the condition is false, the second set of statements is executed.
* Since the condition must be true or false, exactly one of the alternatives will be executed.
* The alternatives are called **branches**, because they are branches in the flow of execution.
* As an aside, if you need to check the parity (evenness or oddness) of numbers often, you might "wrap" this code in a function:

def printParity(x):   
  if x%2 == 0:   
    print x, "is even"   
  else:   
    print x, "is odd"

* For any value of x, printParity displays an appropriate message. When you call it, you can provide any integer expression as an argument.

>>> printParity(17)   
17 is odd   
>>> y = 17   
>>> printParity(y+1)   
18 is even

### Chained conditionals

* Sometimes there are more than two possibilities and we need more than two branches. One way to express a computation like that is a **chained conditional**:

if x < y:   
  print x, "is less than", y   
elif x > y:   
  print x, "is greater than", y   
else:   
  print x, "and", y, "are equal"

* elif is an abbreviation of "else if."
* Again, exactly one branch will be executed. There is no limit of the number of elif statements, but the last branch has to be an else statement:

if choice == 'A':   
  functionA()   
elif choice == 'B':   
  functionB()   
elif choice == 'C':   
  functionC()   
else:   
  print "Invalid choice."

* Each condition is checked in order. If the first is false, the next is checked, and so on. If one of them is true, the corresponding branch executes, and the statement ends.
* Even if more than one condition is true, only the first true branch executes.

### Nested conditionals

* One conditional can also be nested within another. We could have written the trichotomy example as follows:

if x == y:   
  print x, "and", y, "are equal"   
else:   
  if x < y:   
    print x, "is less than", y   
  else:   
    print x, "is greater than", y

* The outer conditional contains two branches.
* The first branch contains a simple output statement. The second branch contains another if statement, which has two branches of its own. Those two branches are both output statements, although they could have been conditional statements as well.
* Although the indentation of the statements makes the structure apparent, nested conditionals become difficult to read very quickly.
* In general, it is a good idea to avoid them when you can.
* Logical operators often provide a way to simplify nested conditional statements. For example, we can rewrite the following code using a single conditional:

if 0 < x:   
  if x < 10:   
    print "x is a positive single digit."

* The print statement is executed only if we make it past both the conditionals, so we can use the and operator:

if 0 < x and x < 10:   
  print "x is a positive single digit."

* These kinds of conditions are common, so Python provides an alternative syntax that is similar to mathematical notation:

if 0 < x < 10:   
  print "x is a positive single digit."

* This condition is semantically the same as the compound boolean expression and the nested conditional.

### The return statement

* The return statement allows you to terminate the execution of a function before you reach the end. One reason to use it is if you detect an error condition:

import math   
  
def printLogarithm(x):   
  if x <= 0:   
    print "Positive numbers only, please."   
    return   
  
  result = math.log(x)   
  print "The log of x is", result

* The function printLogarithm has a parameter named x.
* The first thing it does is check whether x is less than or equal to 0, in which case it displays an error message and then uses return to exit the function.
* The flow of execution immediately returns to the caller, and the remaining lines of the function are not executed.

Remember that to use a function from the math module, you have to import it.

### Recursion

* We mentioned that it is legal for one function to call another, and you have seen several examples of that.
* We neglected to mention that it is also legal for a function to call itself. I
* t may not be obvious why that is a good thing, but it turns out to be one of the most magical and interesting things a program can do. For example, look at the following function:

def countdown(n):   
  if n == 0:   
    print "Blastoff!"   
  else:   
    print n   
    countdown(n-1)

* countdown expects the parameter, n, to be a positive integer.
* If n is 0, it outputs the word, "Blastoff!" Otherwise, it outputs n and then calls a function named countdown  itself  passing n-1 as an argument.
* What happens if we call this function like this:

>>> countdown(3)

The execution of countdown begins with n=3, and since n is not 0, it outputs the value 3, and then calls itself...

The execution of countdown begins with n=2, and since n is not 0, it outputs the value 2, and then calls itself...

The execution of countdown begins with n=1, and since n is not 0, it outputs the value 1, and then calls itself...

The execution of countdown begins with n=0, and since n is 0, it outputs the word, "Blastoff!" and then returns.

The countdown that got n=1 returns.

The countdown that got n=2 returns.

The countdown that got n=3 returns.

* And then you're back in \_\_main\_\_ (what a trip). So, the total output looks like this:

3   
2   
1   
Blastoff!

* As a second example, look again at the functions newLine and threeLines:

def newline():   
  print   
  
def threeLines():   
  newLine()   
  newLine()   
  newLine()

* Although these work, they would not be much help if we wanted to output 2 newlines, or 106. A better alternative would be this:

def nLines(n):   
  if n > 0:   
    print   
    nLines(n-1)

* This program is similar to countdown; as long as n is greater than 0, it outputs one newline and then calls itself to output n-1 additional newlines.
* Thus, the total number of newlines is 1 + (n - 1) which, if you do your algebra right, comes out to n.
* The process of a function calling itself is **recursion**, and such functions are said to be recursive.

### Infinite recursion

* If a recursion never reaches a base case, it goes on making recursive calls forever, and the program never terminates. This is known as **infinite recursion**, and it is generally not considered a good idea.
* Here is a minimal program with an infinite recursion:

def recurse():   
  recurse()

* In most programming environments, a program with infinite recursion does not really run forever.
* Python reports an error message when the maximum recursion depth is reached:

  File "<stdin>", line 2, in recurse   
  (98 repetitions omitted)   
  File "<stdin>", line 2, in recurse   
RuntimeError: Maximum recursion depth exceeded

* This traceback is a little bigger than the one we saw in the previous chapter. When the error occurs, there are 100 recurse frames on the stack!

### Keyboard input

* The programs we have written so far are a bit rude in the sense that they accept no input from the user. They just do the same thing every time.
* Python provides built-in functions that get input from the keyboard.
* The simplest is called raw\_input. When this function is called, the program stops and waits for the user to type something.
* When the user presses Return or the Enter key, the program resumes and raw\_input returns what the user typed as a **string**:

>>> input = raw\_input ()   
What are you waiting for?   
>>> print input   
What are you waiting for?

* Before calling raw\_input, it is a good idea to print a message telling the user what to input. This message is called a **prompt**. We can supply a prompt as an argument to raw\_input:

>>> name = raw\_input ("What...is your name? ")   
What...is your name? Arthur, King of the Britons!   
>>> print name   
Arthur, King of the Britons!

* If we expect the response to be an integer, we can use the input function:

prompt = "What...is the airspeed velocity of an unladen swallow?\n"   
speed = input(prompt)

* The sequence \n at the end of the string represents a newline, so the user's input appears below the prompt.
* If the user types a string of digits, it is converted to an integer and assigned to speed. Unfortunately, if the user types a character that is not a digit, the program crashes:

>>> speed = input (prompt)   
What...is the airspeed velocity of an unladen swallow?   
What do you mean, an African or a European swallow?   
SyntaxError: invalid syntax

* To avoid this kind of error, it is generally a good idea to use raw\_input to get a string and then use conversion functions to convert to other types.

1. **Iteration**

**Multiple assignment**

* As you may have discovered, it is legal to make more than one assignment to the same variable. A new assignment makes an existing variable refer to a new value (and stop referring to the old value).

bruce = 5   
print bruce,   
bruce = 7   
print bruce

* The output of this program is 5 7, because the first time bruce is printed, his value is 5, and the second time, his value is 7. The comma at the end of the first print statement suppresses the newline after the output, which is why both outputs appear on the same line.
* With multiple assignment it is especially important to distinguish between an assignment operation and a statement of equality. Because Python uses the equal sign (=) for assignment, it is tempting to interpret a statement like a = b as a statement of equality. It is not!
* First, equality is commutative and assignment is not. For example, in mathematics, if a = 7 then 7 = a. But in Python, the statement a = 7 is legal and 7 = a is not.
* Furthermore, in mathematics, a statement of equality is always true. If a = b now, then a will always equal b. In Python, an assignment statement can make two variables equal, but they don't have to stay that way:

a = 5   
b = a    # a and b are now equal   
a = 3    # a and b are no longer equal

* The third line changes the value of a but does not change the value of b, so they are no longer equal. (In some programming languages, a different symbol is used for assignment, such as <- or :=, to avoid confusion.)
* Although multiple assignment is frequently helpful, you should use it with caution. If the values of variables change frequently, it can make the code difficult to read and debug.

**The while statement**

* Computers are often used to automate repetitive tasks.
* Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.
* We have seen two programs, nLines and countdown, that use recursion to perform repetition, which is also called **iteration**.
* Because iteration is so common, Python provides several language features to make it easier. The first feature we are going to look at is the while statement.
* Here is what countdown looks like with a while statement:

def countdown(n):   
  while n > 0:   
    print n   
    n = n-1   
  print "Blastoff!"

* Since we removed the recursive call, this function is not recursive.
* You can almost read the while statement as if it were English. It means, "While n is greater than 0, continue displaying the value of n and then reducing the value of n by 1. When you get to 0, display the word Blastoff!"
* More formally, here is the flow of execution for a while statement:

1. Evaluate the condition, yielding 0 or 1.
2. If the condition is false (0), exit the while statement and continue execution at the next statement.
3. If the condition is true (1), execute each of the statements in the body and then go back to step 1.

* The body consists of all of the statements below the header with the same indentation.
* This type of flow is called a **loop** because the third step loops back around to the top. Notice that if the condition is false the first time through the loop, the statements inside the loop are never executed.
* The body of the loop should change the value of one or more variables so that eventually the condition becomes false and the loop terminates. Otherwise the loop will repeat forever, which is called an **infinite loop**. An endless source of amusement for computer scientists is the observation that the directions on shampoo, "Lather, rinse, repeat," are an infinite loop.
* In the case of countdown, we can prove that the loop terminates because we know that the value of n is finite, and we can see that the value of n gets smaller each time through the loop, so eventually we have to get to 0. In other cases, it is not so easy to tell:

def sequence(n):   
  while n != 1:   
    print n,   
    if n%2 == 0:        # n is even   
      n = n/2   
    else:               # n is odd   
      n = n\*3+1

* The condition for this loop is n != 1, so the loop will continue until n is 1, which will make the condition false.
* Each time through the loop, the program outputs the value of n and then checks whether it is even or odd. If it is even, the value of n is divided by 2. If it is odd, the value is replaced by n\*3+1. For example, if the starting value (the argument passed to sequence) is 3, the resulting sequence is 3, 10, 5, 16, 8, 4, 2, 1.
* Since n sometimes increases and sometimes decreases, there is no obvious proof that n will ever reach 1, or that the program terminates. For some particular values of n, we can prove termination. For example, if the starting value is a power of two, then the value of n will be even each time through the loop until it reaches 1. The previous example ends with such a sequence, starting with 16.
* Particular values aside, the interesting question is whether we can prove that this program terminates for *all positive values* of n. So far, no one has been able to prove it *or* disprove it!

1. **Strings**

### A compound data type

* So far we have seen three types: int, float, and string.
* Strings are qualitatively different from the other two because they are made up of smaller pieces  characters.
* Types that comprise smaller pieces are called **compound data types**.
* Depending on what we are doing, we may want to treat a compound data type as a single thing, or we may want to access its parts. This ambiguity is useful.
* The bracket operator selects a single character from a string.

>>> fruit = "banana"   
>>> letter = fruit[1]   
>>> print letter

* The expression fruit[1] selects character number 1 from fruit. The variable letter refers to the result. When we display letter, we get a surprise:

a

* The first letter of "banana" is not a.
* Unless you are a computer scientist. In that case you should think of the expression in brackets as an offset from the beginning of the string, and the offset of the first letter is zero.
* So b is the 0th letter ("zero-eth") of "banana", a is the 1th letter ("one-eth"), and n is the 2th ("two-eth") letter.
* To get the first letter of a string, you just put 0, or any expression with the value 0, in the brackets:

>>> letter = fruit[0]   
>>> print letter   
b

* The expression in brackets is called an **index**.
* An index specifies a member of an ordered set, in this case the set of characters in the string.
* The index *indicates* which one you want, hence the name. It can be any integer expression.

### Length

* The len function returns the number of characters in a string:

>>> fruit = "banana"   
>>> len(fruit)   
6

* To get the last letter of a string, you might be tempted to try something like this:

length = len(fruit)   
last = fruit[length]       # ERROR!

* That won't work. It causes the runtime error   
  IndexError: string index out of range.
* The reason is that there is no 6th letter in "banana". Since we started counting at zero, the six letters are numbered 0 to 5. To get the last character, we have to subtract 1 from length:

length = len(fruit)   
last = fruit[length-1]

* Alternatively, we can use negative indices, which count backward from the end of the string. The expression fruit[-1] yields the last letter, fruit[-2] yields the second to last, and so on.

### Traversal and the for loop

* A lot of computations involve processing a string one character at a time.
* Often they start at the beginning, select each character in turn, do something to it, and continue until the end.
* This pattern of processing is called a **traversal**. One way to encode a traversal is with a while statement:

index = 0   
while index < len(fruit):   
  letter = fruit[index]   
  print letter   
  index = index + 1

* This loop traverses the string and displays each letter on a line by itself.
* The loop condition is index < len(fruit), so when index is equal to the length of the string, the condition is false, and the body of the loop is not executed.
* The last character accessed is the one with the index len(fruit)-1, which is the last character in the string.
* Using an index to traverse a set of values is so common that Python provides an alternative, simpler syntax  the for loop:

for char in fruit:   
  print char

* Each time through the loop, the next character in the string is assigned to the variable char. The loop continues until no characters are left.
* The following example shows how to use concatenation and a for loop to generate an abecedarian series. "Abecedarian" refers to a series or list in which the elements appear in alphabetical order. For example, in Robert McCloskey's book *Make Way for Ducklings*, the names of the ducklings are Jack, Kack, Lack, Mack, Nack, Ouack, Pack, and Quack. This loop outputs these names in order:

prefixes = "JKLMNOPQ"   
suffix = "ack"   
  
for letter in prefixes:   
  print letter + suffix

The output of this program is:

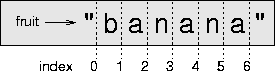
Jack   
Kack   
Lack   
Mack   
Nack   
Oack   
Pack   
Qack

### String slices

* A segment of a string is called a **slice**. Selecting a slice is similar to selecting a character:

>>> s = "Peter, Paul, and Mary"   
>>> print s[0:5]   
Peter   
>>> print s[7:11]   
Paul   
>>> print s[17:21]   
Mary

* The operator [n:m] returns the part of the string from the "n-eth" character to the "m-eth" character, including the first but excluding the last.
* This behavior is counterintuitive; it makes more sense if you imagine the indices pointing *between* the characters, as in the following diagram:



* If you omit the first index (before the colon), the slice starts at the beginning of the string. If you omit the second index, the slice goes to the end of the string. Thus:

>>> fruit = "banana"   
>>> fruit[:3]   
'ban'   
>>> fruit[3:]   
'ana'

### String comparison

* The comparison operators work on strings. To see if two strings are equal:

if word == "banana":   
  print  "Yes, we have no bananas!"

* Other comparison operations are useful for putting words in alphabetical order:

if word < "banana":   
  print "Your word," + word + ", comes before banana."   
elif word > "banana":   
  print "Your word," + word + ", comes after banana."   
else:   
  print "Yes, we have no bananas!"

* You should be aware, though, that Python does not handle upper- and lowercase letters the same way that people do. All the uppercase letters come before all the lowercase letters. As a result:

Your word, Zebra, comes before banana.

* A common way to address this problem is to convert strings to a standard format, such as all lowercase, before performing the comparison.
* A more difficult problem is making the program realize that zebras are not fruit.

### Strings are immutable

* It is tempting to use the [] operator on the left side of an assignment, with the intention of changing a character in a string. For example:

greeting = "Hello, world!"   
greeting[0] = 'J'            # ERROR!   
print greeting

* Instead of producing the output Jello, world!, this code produces the runtime error TypeError: object doesn't support item   
  assignment.
* Strings are **immutable**, which means you can't change an existing string. The best you can do is create a new string that is a variation on the original:

greeting = "Hello, world!"   
newGreeting = 'J' + greeting[1:]   
print newGreeting

* The solution here is to concatenate a new first letter onto a slice of greeting. This operation has no effect on the original string.

### Looping and counting

* The following program counts the number of times the letter a appears in a string:

fruit = "banana"   
count = 0   
for char in fruit:   
  if char == 'a':   
    count = count + 1   
print count

* This program demonstrates another pattern of computation called a **counter**.
* The variable count is initialized to 0 and then incremented each time an a is found. (To **increment** is to increase by one; it is the opposite of **decrement**, and unrelated to "excrement," which is a noun.)
* When the loop exits, count contains the result  the total number of a's.

### The string module

* The string module contains useful functions that manipulate strings. As usual, we have to import the module before we can use it:

>>> import string

* The string module includes a function named find that does the same thing as the function we wrote.
* To call it we have to specify the name of the module and the name of the function using dot notation.

>>> fruit = "banana"   
>>> index = string.find(fruit, "a")   
>>> print index   
1

* This example demonstrates one of the benefits of modules  they help avoid collisions between the names of built-in functions and user-defined functions.
* By using dot notation we can specify which version of find we want.
* Actually, string.find is more general than our version. First, it can find substrings, not just characters:

>>> string.find("banana", "na")   
2

* Also, it takes an additional argument that specifies the index it should start at:

>>> string.find("banana", "na", 3)   
4

* Or it can take two additional arguments that specify a range of indices:

>>> string.find("bob", "b", 1, 2)   
-1

* In this example, the search fails because the letter *b* does not appear in the index range from 1 to 2 (not including 2).

### Character classification

* It is often helpful to examine a character and test whether it is upper- or lowercase, or whether it is a character or a digit.
* The string module provides several constants that are useful for these purposes.
* The string string.lowercase contains all of the letters that the system considers to be lowercase.
* Similarly, string.uppercase contains all of the uppercase letters. Try the following and see what you get:

>>> print string.lowercase   
>>> print string.uppercase   
>>> print string.digits

* We can use these constants and find to classify characters.
* For example, if find(lowercase, ch) returns a value other than -1, then ch must be lowercase:

def isLower(ch):   
  return string.find(string.lowercase, ch) != -1

* Alternatively, we can take advantage of the in operator, which determines whether a character appears in a string:

def isLower(ch):   
  return ch in string.lowercase

* As yet another alternative, we can use the comparison operator:

def isLower(ch):   
  return 'a' <= ch <= 'z'

* If ch is between *a* and *z*, it must be a lowercase letter.
* Another constant defined in the string module may surprise you when you print it:

>>> print string.whitespace

* **Whitespace** characters move the cursor without printing anything.
* They create the white space between visible characters (at least on white paper).
* The constant string.whitespace contains all the whitespace characters, including space, tab (\t), and newline (\n).

1. **Lists**

* A **list** is an ordered set of values, where each value is identified by an index.
* The values that make up a list are called its **elements**.
* Lists are similar to strings, which are ordered sets of characters, except that the elements of a list can have any type.
* Lists and strings  and other things that behave like ordered sets  are called **sequences**.

### List values

* There are several ways to create a new list; the simplest is to enclose the elements in square brackets ([ and ]):

[10, 20, 30, 40]   
["spam", "bungee", "swallow"]

* The first example is a list of four integers.
* The second is a list of three strings.
* The elements of a list don't have to be the same type. The following list contains a string, a float, an integer, and (mirabile dictu) another list:

["hello", 2.0, 5, [10, 20]]

* A list within another list is said to be **nested**.
* Lists that contain consecutive integers are common, so Python provides a simple way to create them:

>>> range(1,5)   
[1, 2, 3, 4]

* The range function takes two arguments and returns a list that contains all the integers from the first to the second, including the first but not including the second!
* There are two other forms of range. With a single argument, it creates a list that starts at 0:

>>> range(10)   
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

* If there is a third argument, it specifies the space between successive values, which is called the **step size**. This example counts from 1 to 10 by steps of 2:

>>> range(1, 10, 2)   
[1, 3, 5, 7, 9]

* Finally, there is a special list that contains no elements. It is called the empty list, and it is denoted [].
* With all these ways to create lists, it would be disappointing if we couldn't assign list values to variables or pass lists as arguments to functions. We can.

vocabulary = ["ameliorate", "castigate", "defenestrate"]   
numbers = [17, 123]   
empty = []   
print vocabulary, numbers, empty   
['ameliorate', 'castigate', 'defenestrate'] [17, 123] []

### Accessing elements

* The syntax for accessing the elements of a list is the same as the syntax for accessing the characters of a string  the bracket operator ([]). The expression inside the brackets specifies the index. Remember that the indices start at 0:

print numbers[0]   
numbers[1] = 5

* The bracket operator can appear anywhere in an expression. When it appears on the left side of an assignment, it changes one of the elements in the list, so the one-eth element of numbers, which used to be 123, is now 5.
* Any integer expression can be used as an index:

>>> numbers[3-2]   
5   
>>> numbers[1.0]   
TypeError: sequence index must be integer

* If you try to read or write an element that does not exist, you get a runtime error:

>>> numbers[2] = 5   
IndexError: list assignment index out of range

* If an index has a negative value, it counts backward from the end of the list:

>>> numbers[-1]   
5   
>>> numbers[-2]   
17   
>>> numbers[-3]   
IndexError: list index out of range

* numbers[-1] is the last element of the list, numbers[-2] is the second to last, and numbers[-3] doesn't exist.
* It is common to use a loop variable as a list index.

horsemen = ["war", "famine", "pestilence", "death"]   
  
i = 0   
while i < 4:   
  print horsemen[i]   
  i = i + 1

* This while loop counts from 0 to 4. When the loop variable i is 4, the condition fails and the loop terminates. So the body of the loop is only executed when i is 0, 1, 2, and 3.
* Each time through the loop, the variable i is used as an index into the list, printing the i-eth element. This pattern of computation is called a **list traversal**.

### List length

* The function len returns the length of a list. It is a good idea to use this value as the upper bound of a loop instead of a constant. That way, if the size of the list changes, you won't have to go through the program changing all the loops; they will work correctly for any size list:

horsemen = ["war", "famine", "pestilence", "death"]   
  
i = 0   
while i < len(horsemen):   
  print horsemen[i]   
  i = i + 1

* The last time the body of the loop is executed, i is len(horsemen) - 1, which is the index of the last element. When i is equal to len(horsemen), the condition fails and the body is not executed, which is a good thing, because len(horsemen) is not a legal index.
* Although a list can contain another list, the nested list still counts as a single element. The length of this list is four:

['spam!', 1, ['Brie', 'Roquefort', 'Pol le Veq'], [1, 2, 3]]

### List membership

* in is a boolean operator that tests membership in a sequence.

>>> horsemen = ['war', 'famine', 'pestilence', 'death']   
>>> 'pestilence' in horsemen   
True   
>>> 'debauchery' in horsemen   
False

* Since "pestilence" is a member of the horsemen list, the in operator returns true. Since "debauchery" is not in the list, in returns false.
* We can use the not in combination with in to test whether an element is not a member of a list:

>>> 'debauchery' not in horsemen   
True

### Lists and for loops

* The generalized syntax of a for loop is:

for VARIABLE in LIST:   
  BODY

This statement is equivalent to:

i = 0   
while i < len(LIST):   
  VARIABLE = LIST[i]   
  BODY   
  i = i + 1

* The for loop is more concise because we can eliminate the loop variable, i. Here is the previous loop written with a for loop.

for horseman in horsemen:   
  print horseman

* It almost reads like English: "For (every) horseman in (the list of) horsemen, print (the name of the) horseman."
* Any list expression can be used in a for loop:

for number in range(20):   
  if number % 2 == 0:   
    print  number   
  
for fruit in ["banana", "apple", "quince"]:   
  print "I like to eat " + fruit + "s!"

* The first example prints all the even numbers between zero and nineteen. The second example expresses enthusiasm for various fruits.

### List operations

* The + operator concatenates lists:

>>> a = [1, 2, 3]   
>>> b = [4, 5, 6]   
>>> c = a + b   
>>> print c   
[1, 2, 3, 4, 5, 6]

* Similarly, the \* operator repeats a list a given number of times:

>>> [0] \* 4   
[0, 0, 0, 0]   
>>> [1, 2, 3] \* 3   
[1, 2, 3, 1, 2, 3, 1, 2, 3]

* The first example repeats [0] four times. The second example repeats the list [1, 2, 3] three times.

### List slices

* The slice operations we saw in [Section 7.4](http://www.greenteapress.com/thinkpython/thinkCSpy/html/chap07.html#4) also work on lists:

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']   
>>> list[1:3]   
['b', 'c']   
>>> list[:4]   
['a', 'b', 'c', 'd']   
>>> list[3:]   
['d', 'e', 'f']

* If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is really a copy of the whole list.

>>> list[:]   
['a', 'b', 'c', 'd', 'e', 'f']

### Lists are mutable

* Unlike strings, lists are mutable, which means we can change their elements. Using the bracket operator on the left side of an assignment, we can update one of the elements:

>>> fruit = ["banana", "apple", "quince"]   
>>> fruit[0] = "pear"   
>>> fruit[-1] = "orange"   
>>> print fruit   
['pear', 'apple', 'orange']

* With the slice operator we can update several elements at once:

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']   
>>> list[1:3] = ['x', 'y']   
>>> print list   
['a', 'x', 'y', 'd', 'e', 'f']

* We can also remove elements from a list by assigning the empty list to them:

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']   
>>> list[1:3] = []   
>>> print list   
['a', 'd', 'e', 'f']

* And we can add elements to a list by squeezing them into an empty slice at the desired location:

>>> list = ['a', 'd', 'f']   
>>> list[1:1] = ['b', 'c']   
>>> print list   
['a', 'b', 'c', 'd', 'f']   
>>> list[4:4] = ['e']   
>>> print list   
['a', 'b', 'c', 'd', 'e', 'f']

### List deletion

* Using slices to delete list elements can be awkward, and therefore error-prone. Python provides an alternative that is more readable.
* del removes an element from a list:

>>> a = ['one', 'two', 'three']   
>>> del a[1]   
>>> a   
['one', 'three']

* As you might expect, del handles negative indices and causes a runtime error if the index is out of range.
* You can use a slice as an index for del:

>>> list = ['a', 'b', 'c', 'd', 'e', 'f']   
>>> del list[1:5]   
>>> print list   
['a', 'f']

* As usual, slices select all the elements up to, but not including, the second index.

### Aliasing

* Since variables refer to objects, if we assign one variable to another, both variables refer to the same object:

>>> a = [1, 2, 3]   
>>> b = a

* Because the same list has two different names, a and b, we say that it is **aliased**. Changes made with one alias affect the other:

>>> b[0] = 5   
>>> print a   
[5, 2, 3]

* Although this behavior can be useful, it is sometimes unexpected or undesirable. In general, it is safer to avoid aliasing when you are working with mutable objects. Of course, for immutable objects, there's no problem. That's why Python is free to alias strings when it sees an opportunity to economize.

### Cloning lists

* If we want to modify a list and also keep a copy of the original, we need to be able to make a copy of the list itself, not just the reference.
* This process is sometimes called **cloning**, to avoid the ambiguity of the word "copy."
* The easiest way to clone a list is to use the slice operator:

>>> a = [1, 2, 3]   
>>> b = a[:]   
>>> print b   
[1, 2, 3]

* Taking any slice of a creates a new list. In this case the slice happens to consist of the whole list.
* Now we are free to make changes to b without worrying about a:

>>> b[0] = 5   
>>> print a   
[1, 2, 3]

### Nested lists

* A nested list is a list that appears as an element in another list. In this list, the three-eth element is a nested list:

>>> list = ["hello", 2.0, 5, [10, 20]]

* If we print list[3], we get [10, 20]. To extract an element from the nested list, we can proceed in two steps:

>>> elt = list[3]   
>>> elt[0]   
10

* Or we can combine them:

>>> list[3][1]   
20

* Bracket operators evaluate from left to right, so this expression gets the three-eth element of list and extracts the one-eth element from it.

### Matrices

* Nested lists are often used to represent matrices. For example, the matrix:

http://www.greenteapress.com/thinkpython/thinkCSpy/html/illustrations/matrix.png

might be represented as:

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

* matrix is a list with three elements, where each element is a row of the matrix. We can select an entire row from the matrix in the usual way:

>>> matrix[1]   
[4, 5, 6]

* Or we can extract a single element from the matrix using the double-index form:

>>> matrix[1][1]   
5

* The first index selects the row, and the second index selects the column. Although this way of representing matrices is common, it is not the only possibility. A small variation is to use a list of columns instead of a list of rows. Later we will see a more radical alternative using a dictionary.

### Strings and lists

* Two of the most useful functions in the string module involve lists of strings.
* The split function breaks a string into a list of words.
* By default, any number of whitespace characters is considered a word boundary:

>>> import string   
>>> song = "The rain in Spain..."   
>>> string.split(song)   
['The', 'rain', 'in', 'Spain...']

* An optional argument called a **delimiter** can be used to specify which characters to use as word boundaries. The following example uses the string ai as the delimiter:

>>> string.split(song, 'ai')   
['The r', 'n in Sp', 'n...']

* Notice that the delimiter doesn't appear in the list.
* The join function is the inverse of split. It takes a list of strings and concatenates the elements with a space between each pair:

>>> list = ['The', 'rain', 'in', 'Spain...']   
>>> string.join(list)   
'The rain in Spain...'

* Like split, join takes an optional delimiter that is inserted between elements:

>>> string.join(list, '\_')   
'The\_rain\_in\_Spain...'

1. **Tuples**

### Mutability and tuples

* So far, you have seen two compound types: strings, which are made up of characters; and lists, which are made up of elements of any type.
* One of the differences we noted is that the elements of a list can be modified, but the characters in a string cannot.
* In other words, strings are **immutable** and lists are **mutable**.
* There is another type in Python called a **tuple** that is similar to a list except that it is immutable. Syntactically, a tuple is a comma-separated list of values:

>>> tuple = 'a', 'b', 'c', 'd', 'e'

* Although it is not necessary, it is conventional to enclose tuples in parentheses:

>>> tuple = ('a', 'b', 'c', 'd', 'e')

* To create a tuple with a single element, we have to include the final comma:

>>> t1 = ('a',)   
>>> type(t1)   
<type 'tuple'>

* Without the comma, Python treats ('a') as a string in parentheses:

>>> t2 = ('a')   
>>> type(t2)   
<type 'str'>

* Syntax issues aside, the operations on tuples are the same as the operations on lists. The index operator selects an element from a tuple.

>>> tuple = ('a', 'b', 'c', 'd', 'e')   
>>> tuple[0]   
'a'

* And the slice operator selects a range of elements.

>>> tuple[1:3]   
('b', 'c')

* But if we try to modify one of the elements of the tuple, we get an error:

>>> tuple[0] = 'A'   
TypeError: object doesn't support item assignment

* Of course, even if we can't modify the elements of a tuple, we can replace it with a different tuple:

>>> tuple = ('A',) + tuple[1:]   
>>> tuple   
('A', 'b', 'c', 'd', 'e')

### Tuple assignment

* Once in a while, it is useful to swap the values of two variables. With conventional assignment statements, we have to use a temporary variable. For example, to swap a and b:

>>> temp = a   
>>> a = b   
>>> b = temp

* If we have to do this often, this approach becomes cumbersome. Python provides a form of **tuple assignment** that solves this problem neatly:

>>> a, b = b, a

* The left side is a tuple of variables; the right side is a tuple of values. Each value is assigned to its respective variable. All the expressions on the right side are evaluated before any of the assignments. This feature makes tuple assignment quite versatile.
* Naturally, the number of variables on the left and the number of values on the right have to be the same:

>>> a, b, c, d = 1, 2, 3   
ValueError: unpack tuple of wrong size

### Tuples as return values

* Functions can return tuples as return values. For example, we could write a function that swaps two parameters:

def swap(x, y):   
  return y, x

* Then we can assign the return value to a tuple with two variables:

a, b = swap(a, b)

* In this case, there is no great advantage in making swap a function. In fact, there is a danger in trying to encapsulate swap, which is the following tempting mistake:

def swap(x, y):      # incorrect version   
  x, y = y, x

* If we call this function like this:

swap(a, b)

* then a and x are aliases for the same value. Changing x inside swap makes x refer to a different value, but it has no effect on a in \_\_main\_\_. Similarly, changing y has no effect on b.
* This function runs without producing an error message, but it doesn't do what we intended. This is an example of a semantic error.

1. **Dictionaries**

* The compound types you have learned about  strings, lists, and tuples  use integers as indices. If you try to use any other type as an index, you get an error.
* **Dictionaries** are similar to other compound types except that they can use any immutable type as an index.
* As an example, we will create a dictionary to translate English words into Spanish. For this dictionary, the indices are strings.
* One way to create a dictionary is to start with the empty dictionary and add elements. The empty dictionary is denoted {}:

>>> eng2sp = {}   
>>> eng2sp['one'] = 'uno'   
>>> eng2sp['two'] = 'dos'

* The first assignment creates a dictionary named eng2sp; the other assignments add new elements to the dictionary. We can print the current value of the dictionary in the usual way:

>>> print eng2sp   
{'one': 'uno', 'two': 'dos'}

* The elements of a dictionary appear in a comma-separated list. Each entry contains an index and a value separated by a colon. In a dictionary, the indices are called **keys**, so the elements are called **key-value pairs**.
* Another way to create a dictionary is to provide a list of key-value pairs using the same syntax as the previous output:

>>> eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres'}

* If we print the value of eng2sp again, we get a surprise:

>>> print eng2sp   
{'one': 'uno', 'three': 'tres', 'two': 'dos'}

* The key-value pairs are not in order! Fortunately, there is no reason to care about the order, since the elements of a dictionary are never indexed with integer indices. Instead, we use the keys to look up the corresponding values:

>>> print eng2sp['two']   
'dos'

* The key 'two' yields the value 'dos' even though it appears in the third key-value pair.

### Dictionary operations

* The del statement removes a key-value pair from a dictionary.
* For example, the following dictionary contains the names of various fruits and the number of each fruit in stock:

>>> inventory = {'apples': 430, 'bananas': 312, 'oranges': 525,   
'pears': 217}   
>>> print inventory   
{'oranges': 525, 'apples': 430, 'pears': 217, 'bananas': 312}

* If someone buys all of the pears, we can remove the entry from the dictionary:

>>> del inventory['pears']   
>>> print inventory   
{'oranges': 525, 'apples': 430, 'bananas': 312}

* Or if we're expecting more pears soon, we might just change the value associated with pears:

>>> inventory['pears'] = 0   
>>> print inventory   
{'oranges': 525, 'apples': 430, 'pears': 0, 'bananas': 312}

* The len function also works on dictionaries; it returns the number of key-value pairs:

>>> len(inventory)   
4

### Dictionary methods

* A **method** is similar to a function  it takes arguments and returns a value  but the syntax is different.
* For example, the keys method takes a dictionary and returns a list of the keys that appear, but instead of the function syntax keys(eng2sp), we use the method syntax eng2sp.keys().

>>> eng2sp.keys()   
['one', 'three', 'two']

* This form of dot notation specifies the name of the function, keys, and the name of the object to apply the function to, eng2sp. The parentheses indicate that this method has no parameters.
* A method call is called an **invocation**; in this case, we would say that we are invoking keys on the object eng2sp.
* The values method is similar; it returns a list of the values in the dictionary:

>>> eng2sp.values()   
['uno', 'tres', 'dos']

* The items method returns both, in the form of a list of tuples  one for each key-value pair:

>>> eng2sp.items()   
[('one','uno'), ('three', 'tres'), ('two', 'dos')]

* The syntax provides useful type information. The square brackets indicate that this is a list. The parentheses indicate that the elements of the list are tuples.
* If a method takes an argument, it uses the same syntax as a function call. For example, the method has\_key takes a key and returns true (1) if the key appears in the dictionary:

>>> eng2sp.has\_key('one')   
True   
>>> eng2sp.has\_key('deux')   
False

* If you try to call a method without specifying an object, you get an error. In this case, the error message is not very helpful:

>>> has\_key('one')   
NameError: has\_key

### Aliasing and copying

* Because dictionaries are mutable, you need to be aware of aliasing. Whenever two variables refer to the same object, changes to one affect the other.
* If you want to modify a dictionary and keep a copy of the original, use the copy method. For example, opposites is a dictionary that contains pairs of opposites:

>>> opposites = {'up': 'down', 'right': 'wrong', 'true': 'false'}   
>>> alias = opposites   
>>> copy = opposites.copy()

* alias and opposites refer to the same object; copy refers to a fresh copy of the same dictionary. If we modify alias, opposites is also changed:

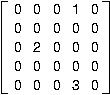
>>> alias['right'] = 'left'   
>>> opposites['right']   
'left'

* If we modify copy, opposites is unchanged:

>>> copy['right'] = 'privilege'   
>>> opposites['right']   
'left'

### Sparse matrices

* Consider a sparse matrix like this one:



* The list representation contains a lot of zeroes:

matrix = [ [0,0,0,1,0],   
           [0,0,0,0,0],   
           [0,2,0,0,0],   
           [0,0,0,0,0],   
           [0,0,0,3,0] ]

* An alternative is to use a dictionary. For the keys, we can use tuples that contain the row and column numbers. Here is the dictionary representation of the same matrix:

matrix = {(0,3): 1, (2, 1): 2, (4, 3): 3}

* We only need three key-value pairs, one for each nonzero element of the matrix. Each key is a tuple, and each value is an integer.
* To access an element of the matrix, we could use the [] operator:

matrix[0,3]   
1

* Notice that the syntax for the dictionary representation is not the same as the syntax for the nested list representation. Instead of two integer indices, we use one index, which is a tuple of integers.
* There is one problem. If we specify an element that is zero, we get an error, because there is no entry in the dictionary with that key:

>>> matrix[1,3]   
KeyError: (1, 3)

* The get method solves this problem:

>>> matrix.get((0,3), 0)   
1

* The first argument is the key; the second argument is the value get should return if the key is not in the dictionary:

>>> matrix.get((1,3), 0)   
0

* get definitely improves the semantics of accessing a sparse matrix. Shame about the syntax.

### Counting letters

* Dictionaries provide an elegant way to generate a histogram(number of occurrences of a character):

>>> letterCounts = {}   
>>> for letter in "Mississippi":   
...   letterCounts[letter] = letterCounts.get (letter, 0) + 1   
...   
>>> letterCounts   
{'M': 1, 's': 4, 'p': 2, 'i': 4}

* We start with an empty dictionary. For each letter in the string, we find the current count (possibly zero) and increment it. At the end, the dictionary contains pairs of letters and their frequencies.
* It might be more appealing to display the histogram in alphabetical order. We can do that with the items and sort methods:

>>> letterItems = letterCounts.items()   
>>> letterItems.sort()   
>>> print letterItems   
[('M', 1), ('i', 4), ('p', 2), ('s', 4)]

1. **Files**

* While a program is running, its data is in memory.
* When the program ends, or the computer shuts down, data in memory disappears.
* To store data permanently, you have to put it in a **file**. Files are usually stored on a hard drive, floppy drive, or CD-ROM.
* When there are a large number of files, they are often organized into **directories** (also called "folders"). Each file is identified by a unique name, or a combination of a file name and a directory name.
* By reading and writing files, programs can exchange information with each other and generate printable formats like PDF.
* Working with files is a lot like working with books.
* To use a book, you have to open it. When you're done, you have to close it. While the book is open, you can either write in it or read from it. In either case, you know where you are in the book. Most of the time, you read the whole book in its natural order, but you can also skip around.
* All of this applies to files as well. To open a file, you specify its name and indicate whether you want to read or write.
* Opening a file creates a file object. In this example, the variable f refers to the new file object.

>>> f = open("test.dat","w")   
>>> print f   
<open file 'test.dat', mode 'w' at fe820>

* The open function takes two arguments. The first is the name of the file, and the second is the mode. Mode "w" means that we are opening the file for writing.
* If there is no file named test.dat, it will be created. If there already is one, it will be replaced by the file we are writing.
* When we print the file object, we see the name of the file, the mode, and the location of the object.
* To put data in the file we invoke the write method on the file object:

>>> f.write("Now is the time")   
>>> f.write("to close the file")

* Closing the file tells the system that we are done writing and makes the file available for reading:

>>> f.close()

* Now we can open the file again, this time for reading, and read the contents into a string. This time, the mode argument is "r" for reading:

>>> f = open("test.dat","r")

* If we try to open a file that doesn't exist, we get an error:

>>> f = open("test.cat","r")   
IOError: [Errno 2] No such file or directory: 'test.cat'

* Not surprisingly, the read method reads data from the file. With no arguments, it reads the entire contents of the file:

>>> text = f.read()   
>>> print text   
Now is the timeto close the file

* There is no space between "time" and "to" because we did not write a space between the strings.
* read can also take an argument that indicates how many characters to read:

>>> f = open("test.dat","r")   
>>> print f.read(5)   
Now i

* If not enough characters are left in the file, read returns the remaining characters. When we get to the end of the file, read returns the empty string:

>>> print f.read(1000006)   
s the timeto close the file   
>>> print f.read()   
  
>>>

* The following function copies a file, reading and writing up to fifty characters at a time. The first argument is the name of the original file; the second is the name of the new file:

def copyFile(oldFile, newFile):   
  f1 = open(oldFile, "r")   
  f2 = open(newFile, "w")   
  while True:   
    text = f1.read(50)   
    if text == "":   
      break   
    f2.write(text)   
  f1.close()   
  f2.close()   
  return

* The break statement is new. Executing it breaks out of the loop; the flow of execution moves to the first statement after the loop.
* In this example, the while loop is infinite because the value True is always true. The *only* way to get out of the loop is to execute break, which happens when text is the empty string, which happens when we get to the end of the file.

### Text files

* A **text file** is a file that contains printable characters and whitespace, organized into lines separated by newline characters.
* Since Python is specifically designed to process text files, it provides methods that make the job easy.
* To demonstrate, we'll create a text file with three lines of text separated by newlines:

>>> f = open("test.dat","w")   
>>> f.write("line one\nline two\nline three\n")   
>>> f.close()

* The readline method reads all the characters up to and including the next newline character:

>>> f = open("test.dat","r")   
>>> print f.readline()   
line one   
  
>>>

* readlines returns all of the remaining lines as a list of strings:

>>> print f.readlines()   
['line two\012', 'line three\012']

* In this case, the output is in list format, which means that the strings appear with quotation marks and the newline character appears as the escape sequence <br>012.
* At the end of the file, readline returns the empty string and readlines returns the empty list:

>>> print f.readline()   
  
>>> print f.readlines()   
[]

* The following is an example of a line-processing program. filterFile makes a copy of oldFile, omitting any lines that begin with #:

def filterFile(oldFile, newFile):   
  f1 = open(oldFile, "r")   
  f2 = open(newFile, "w")   
  while True:   
    text = f1.readline()   
    if text == "":   
      break   
    if text[0] == '#':   
      continue   
    f2.write(text)   
  f1.close()   
  f2.close()   
  return

* The continue statement ends the current iteration of the loop, but continues looping. The flow of execution moves to the top of the loop, checks the condition, and proceeds accordingly.
* Thus, if text is the empty string, the loop exits. If the first character of text is a hash mark, the flow of execution goes to the top of the loop. Only if both conditions fail do we copy text into the new file.

### Writing variables

* The argument of write has to be a string, so if we want to put other values in a file, we have to convert them to strings first. The easiest way to do that is with the str function:

>>> x = 52   
>>> f.write (str(x))

* An alternative is to use the **format operator** %. When applied to integers, % is the modulus operator. But when the first operand is a string, % is the format operator.
* The first operand is the **format string**, and the second operand is a tuple of expressions. The result is a string that contains the values of the expressions, formatted according to the format string.
* As a simple example, the **format sequence** "%d" means that the first expression in the tuple should be formatted as an integer. Here the letter *d* stands for "decimal":

>>> cars = 52   
>>> "%d" % cars   
'52'

* The result is the string '52', which is not to be confused with the integer value 52.
* A format sequence can appear anywhere in the format string, so we can embed a value in a sentence:

>>> cars = 52   
>>> "In July we sold %d cars." % cars   
'In July we sold 52 cars.'

* The format sequence "%f" formats the next item in the tuple as a floating-point number, and "%s" formats the next item as a string:

>>> "In %d days we made %f million %s." % (34,6.1,'dollars')   
'In 34 days we made 6.100000 million dollars.'

* By default, the floating-point format prints six decimal places.
* The number of expressions in the tuple has to match the number of format sequences in the string. Also, the types of the expressions have to match the format sequences:

>>> "%d %d %d" % (1,2)   
TypeError: not enough arguments for format string   
>>> "%d" % 'dollars'   
TypeError: illegal argument type for built-in operation

* In the first example, there aren't enough expressions; in the second, the expression is the wrong type.
* For more control over the format of numbers, we can specify the number of digits as part of the format sequence:

>>> "%6d" % 62   
'    62'   
>>> "%12f" % 6.1   
'    6.100000'

* The number after the percent sign is the minimum number of spaces the number will take up. If the value provided takes fewer digits, leading spaces are added. If the number of spaces is negative, trailing spaces are added:

>>> "%-6d" % 62   
'62    '

* For floating-point numbers, we can also specify the number of digits after the decimal point:

>>> "%12.2f" % 6.1   
'        6.10'

* In this example, the result takes up twelve spaces and includes two digits after the decimal. This format is useful for printing dollar amounts with the decimal points aligned.
* For example, imagine a dictionary that contains student names as keys and hourly wages as values. Here is a function that prints the contents of the dictionary as a formatted report:

def report (wages) :   
  students = wages.keys()   
  students.sort()   
  for student in students :   
    print "%-20s %12.2f" % (student, wages[student])

* To test this function, we'll create a small dictionary and print the contents:

>>> wages = {'mary': 6.23, 'joe': 5.45, 'joshua': 4.25}   
>>> report (wages)   
joe                          5.45   
joshua                       4.25   
mary                         6.23

* By controlling the width of each value, we guarantee that the columns will line up, as long as the names contain fewer than twenty-one characters and the wages are less than one billion dollars an hour.

### Directories

* When you create a new file by opening it and writing, the new file goes in the current directory (wherever you were when you ran the program). Similarly, when you open a file for reading, Python looks for it in the current directory.
* If you want to open a file somewhere else, you have to specify the **path** to the file, which is the name of the directory (or folder) where the file is located:

>>>   f = open("/usr/share/dict/words","r")   
>>>   print f.readline()   
Aarhus

* This example opens a file named words that resides in a directory named dict, which resides in share, which resides in usr, which resides in the top-level directory of the system, called /.
* You cannot use / as part of a filename; it is reserved as a delimiter between directory and filenames.
* The file /usr/share/dict/words contains a list of words in alphabetical order, of which the first is the name of a Danish university.

### Pickling

* In order to put values into a file, you have to convert them to strings. You have already seen how to do that with str:

>>> f.write (str(12.3))   
>>> f.write (str([1,2,3]))

* The problem is that when you read the value back, you get a string. The original type information has been lost. In fact, you can't even tell where one value ends and the next begins:

>>>   f.readline()   
'12.3[1, 2, 3]'

* The solution is **pickling**, so called because it "preserves" data structures. The pickle module contains the necessary commands. To use it, import pickle and then open the file in the usual way:

>>> import pickle   
>>> f = open("test.pck","w")

* To store a data structure, use the dump method and then close the file in the usual way:

>>> pickle.dump(12.3, f)   
>>> pickle.dump([1,2,3], f)   
>>> f.close()

* Then we can open the file for reading and load the data structures we dumped:

>>> f = open("test.pck","r")   
>>> x = pickle.load(f)   
>>> x   
12.3   
>>> type(x)   
<type 'float'>   
>>> y = pickle.load(f)   
>>> y   
[1, 2, 3]   
>>> type(y)   
<type 'list'>

* Each time we invoke load, we get a single value from the file, complete with its original type.

### Exceptions

* Whenever a runtime error occurs, it creates an **exception**. Usually, the program stops and Python prints an error message.
* For example, dividing by zero creates an exception:

>>> print 55/0   
ZeroDivisionError: integer division or modulo

* So does accessing a nonexistent list item:

>>> a = []   
>>> print a[5]   
IndexError: list index out of range

* Or accessing a key that isn't in the dictionary:

>>> b = {}   
>>> print b['what']   
KeyError: what

* Or trying to open a nonexistent file:

>>> f = open("Idontexist", "r")   
IOError: [Errno 2] No such file or directory: 'Idontexist'

* In each case, the error message has two parts: the type of error before the colon, and specifics about the error after the colon.
* Normally Python also prints a traceback of where the program was, but we have omitted that from the examples.
* Sometimes we want to execute an operation that could cause an exception, but we don't want the program to stop. We can **handle** the exception using the try and except statements.
* For example, we might prompt the user for the name of a file and then try to open it. If the file doesn't exist, we don't want the program to crash; we want to handle the exception:

filename = raw\_input('Enter a file name: ')   
try:   
  f = open (filename, "r")   
except IOError:   
  print 'There is no file named', filename

* The try statement executes the statements in the first block. If no exceptions occur, it ignores the except statement. If an exception of type IOError occurs, it executes the statements in the except branch and then continues.
* We can encapsulate this capability in a function: exists takes a filename and returns true if the file exists, false if it doesn't:

def exists(filename):   
  try:   
    f = open(filename)   
    f.close()   
    return True   
  except IOError:   
    return False

* You can use multiple except blocks to handle different kinds of exceptions.
* If your program detects an error condition, you can make it **raise** an exception. Here is an example that gets input from the user and checks for the value 17. Assuming that 17 is not valid input for some reason, we raise an exception.

def inputNumber () :   
  x = input ('Pick a number: ')   
  if x == 17 :   
    raise ValueError, '17 is a bad number'   
  return x

* The raise statement takes two arguments: the exception type and specific information about the error. ValueError is one of the exception types Python provides for a variety of occasions. Other examples include TypeError, KeyError, and my favorite, NotImplementedError.
* If the function that called inputNumber handles the error, then the program can continue; otherwise, Python prints the error message and exits:

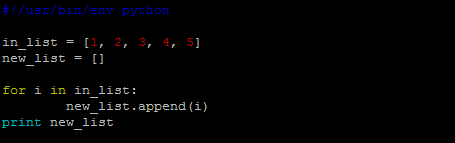
>>> inputNumber ()   
Pick a number: 17   
ValueError: 17 is a bad number

* The error message includes the exception type and the additional information you provided.

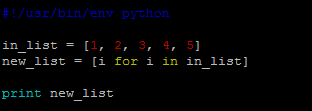
1. **List Comprehensions**

* List comprehensions provides compact way of mapping a list to another list by applying a function to each of elements of the list
* It consists of brackets containing an expression followed by a for clause then zero or more for or if clauses
* The result will be a new list resulting from evaluating the expression in context of for and if clauses which follow it
* It doesn’t solve a new problem but provide a new syntax to solve an existing problem.
* For example lets write following program to copy items from one list to another

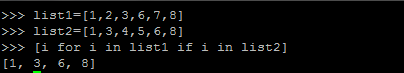
General:



Using List Comprehension:



* Let’s walk through following examples to understand more about list comprehensions (lc).
  1. Write a lc to compare two lists and print common elements in both list



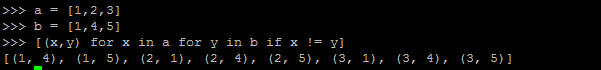
* 1. Write a lc to read a list of strings and print the length of each element



* 1. Write a lc to print even squares from a range



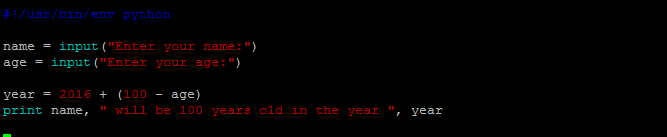
* 1. Write a lc to create tuple from pair of elements from two lists such that the elements in given pair are not equal.



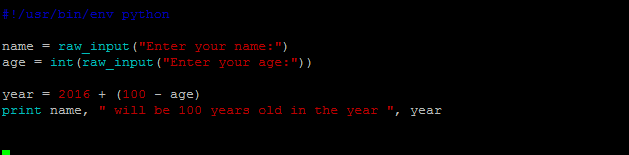
**Python Practice Problems**

1. Write a program that asks the user to enter their name and their age. Print out a message addressed to them that tells them the year that they will turn 100 years old. (Note: Write two solutions for this in which use input() in one solution and raw\_input() in another solution)

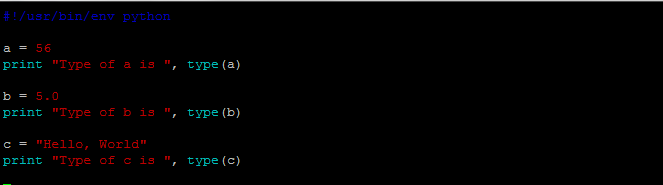
Using input():



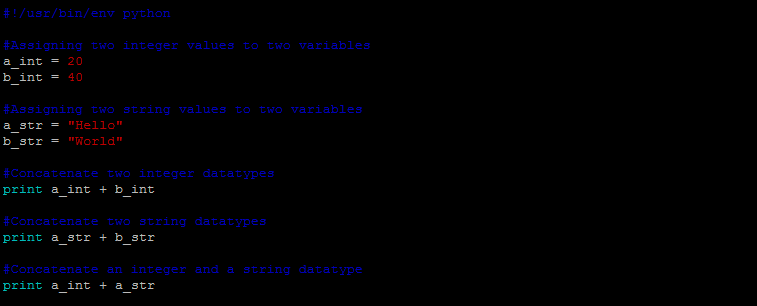
Using raw\_input():



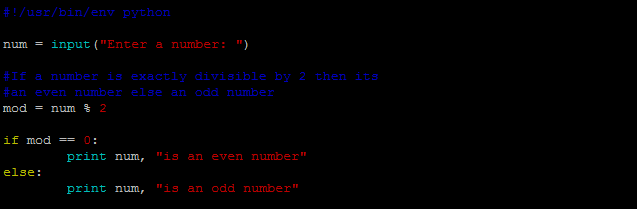
1. Write a program to assign value 56 to a, 5.0 to b, “Hello, World” to c and print the datatype of the values assigned.



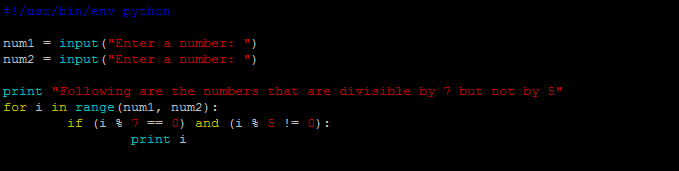
1. Assign two int datatype, two str datatype and wirte a program to concatenate the possible combinations and try to analyze the output



1. Write a program which accepts a user number and print if it’s odd or even

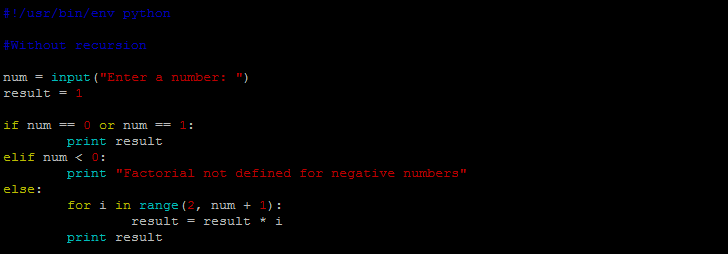


1. Write a program to print all the numbers that are divisible by 7 but not by 5

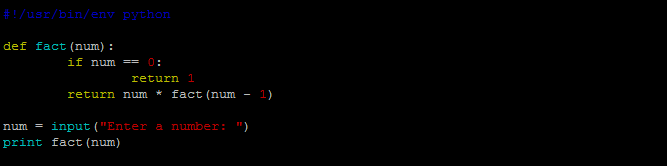


1. Write a program to find the factorial of a given number both using recursion and without recursion

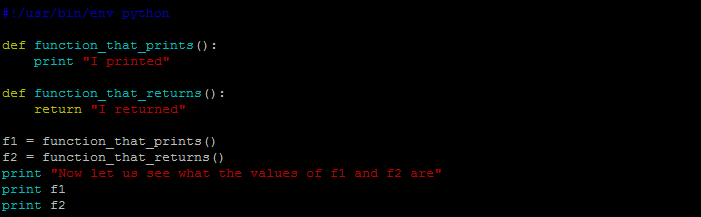
Without Recursion:



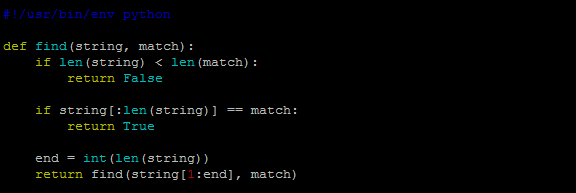
With Recursion:



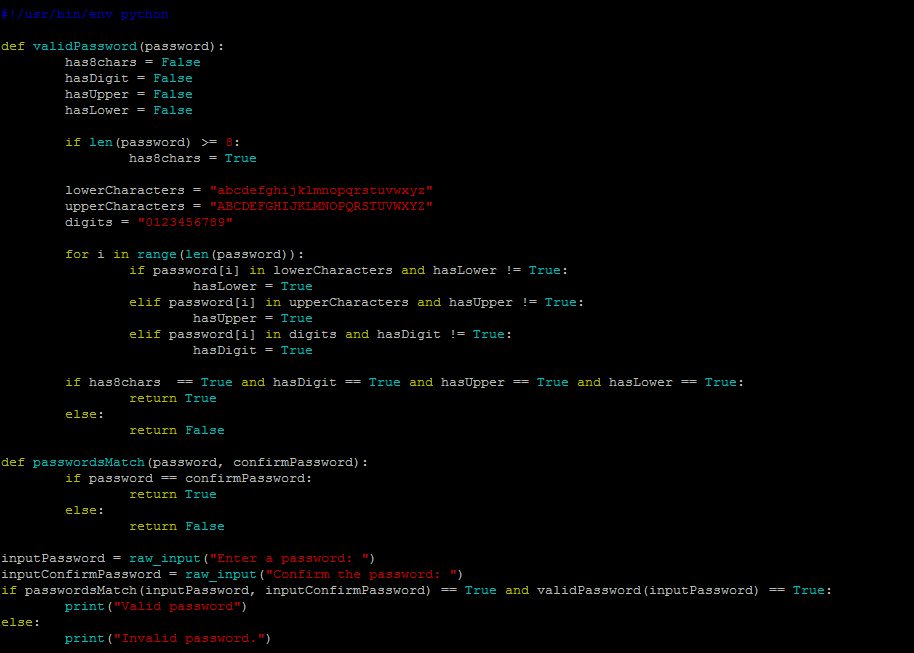
1. Run the following program and analyze the difference between using a print and return statements in a function in python



1. Write a function to search for a substring in a string using recursion. If found then print True else False

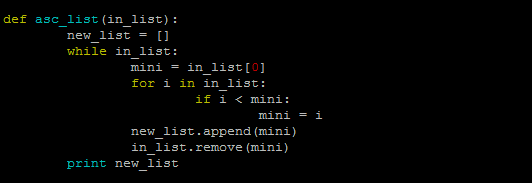


1. Write a program which takes a password twice and then check if it has 8 characters, atleast one lower and one upper case and one number. If yes then print Valid else Invalid

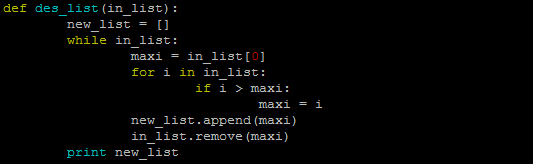


1. Write function to sort user input list of integers in both ascending and descending order without using sort function.

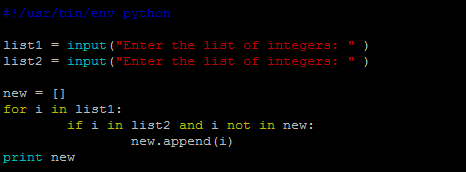
Ascending order:



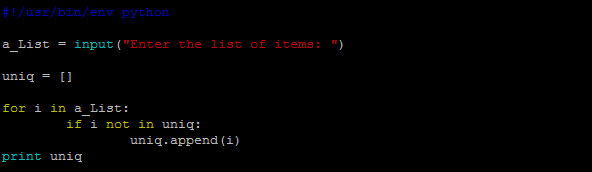
Descending Order:



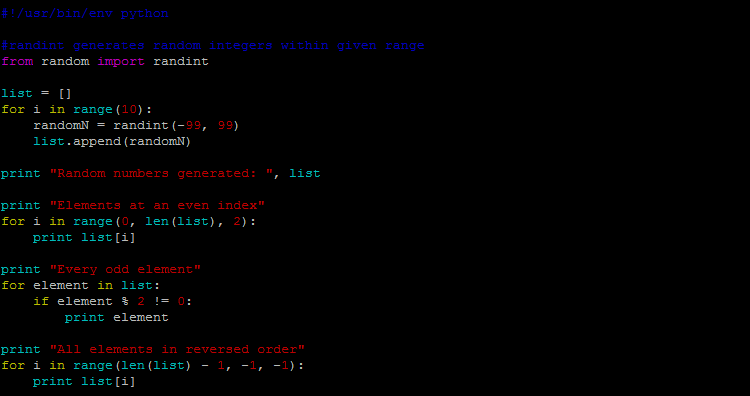
1. Write a program to compare two lists and print the common items onto the terminal



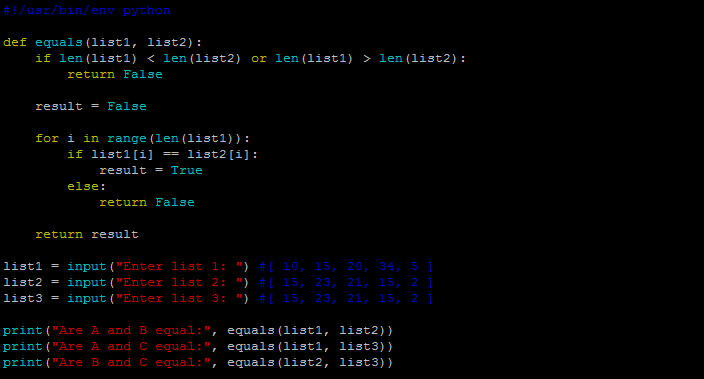
1. Write a program to remove the duplicates from given list and print unique items



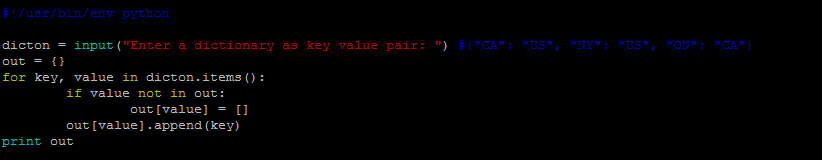
1. Write a program to initialize a list with ten random integers and then print a) every element at even index b) every odd element c) All elements in reverse order



1. Write a function def equals(a, b) that checks whether two lists have the same elements in same order



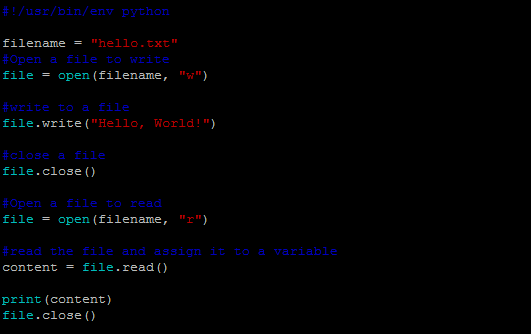
1. Write a program to reverse key & values of a dictionary – each key should map to set of values that mapped to it



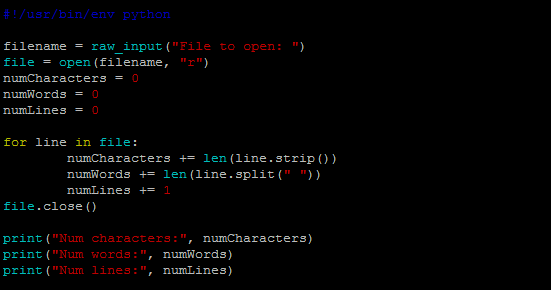
1. Write a program to count occurrence of each word in a text file



1. Write a program to open a file -> add some text -> close the file -> open file again -> read the message to a variable and print it



1. Write a program to read a file and print number of words, characters and lines in the file



1. Write a program to read a file and add line numbers to each line and write it to another file

