Programas y lenguajes formales

Programas

Secuencia de instrucciones para ejecutar una tarea

Elementos basicos del programa:

input
Get data from the keyboard, a file, or some other device.

output
Display data on the screen or send data to a file or other device.

math
Perform basic mathematical operations like addition and multiplication.

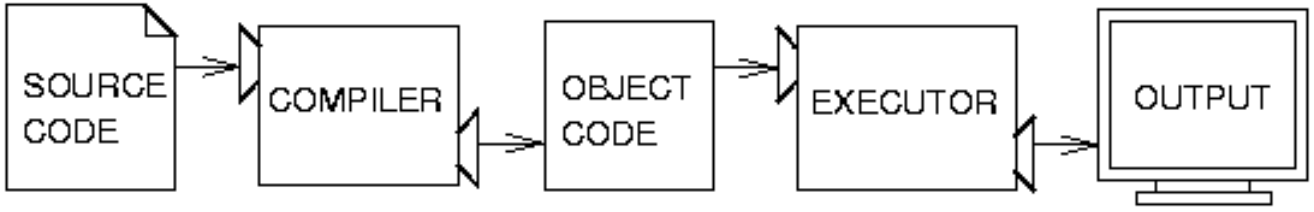
conditional execution
Check for certain conditions and execute the appropriate sequence of statements.

repetition
Perform some action repeatedly, usually with some variation.

Inerpretes y compiladores

Two kinds of programs process high-level languages into low-level languages: **interpreters** and **compilers**. An interpreter reads a high-level program and executes it, meaning that it does what the program says. It processes the program a little at a time, alternately reading lines and performing computations.

A compiler reads the program and translates it completely before the program starts running. In this case, the high-level program is called the **source code**, and the translated program is called the **object code** or the **executable**. Once a program is compiled, you can execute it repeatedly without further translation.



Debbuging

Programming is a complex process, and because it is done by human beings, it often leads to errors. For whimsical reasons, programming errors are called **bugs** and the process of tracking them down and correcting them is called **debugging**.

Three kinds of errors can occur in a program: **syntax errors**, **runtime errors**, and **semantic errors**. It is useful to distinguish between them in order to track them down more quickly.

Syntax errors

A program can be executed only if it is syntactically correct; otherwise, the process fails and returns an error message.

Syntax refers to the structure of a program and the rules about that structure.

For example, in English, a sentence must begin with a capital letter and end with a period. this sentence contains a **syntax error**. So does this one

Runtime errors

The second type of error is a runtime error, so called because the error does not appear until you run the program. These errors are also called **exceptions** because they usually indicate that something exceptional (and bad) has happened.

Semantic errors

The third type of error is the **semantic error**. If there is a semantic error in your program, it will run successfully, in the sense that the computer will not generate any error messages, but it will not do the right thing. It will do something else.

Specifically, it will do what you told it to do.

The problem is that the program you wrote is not the program you wanted to write. The meaning of the program (its semantics) is wrong. Identifying semantic errors can be tricky because it requires you to work backward by looking at the output of the program and trying to figure out what it is doing.

Experimental debugging

One of the most important skills you will acquire is debugging. Although it can be frustrating, debugging is one of the most intellectually rich, challenging, and interesting parts of programming.

In some ways, debugging is like detective work. You are confronted with clues, and you have to infer the processes and events that led to the results you see.

Debugging is also like an experimental science. Once you have an idea what is going wrong, you modify your program and try again. If your hypothesis was correct, then you can predict the result of the modification, and you take a step closer to a working program. If your hypothesis was wrong, you have to come up with a new one

For some people, programming and debugging are the same thing. That is, programming is the process of gradually debugging a program until it does what you want. The idea is that you should start with a program that does *something* and make small modifications, debugging them as you go, so that you always have a working program.

Formal and natural languages

Natural languages are the languages that people speak, such as English, Spanish, and French. They were not designed by people (although people try to impose some order on them); they evolved naturally.

Formal languages are languages that are designed by people for specific applications. For example, the notation that mathematicians use is a formal language that is particularly good at denoting relationships among numbers and symbols. Chemists use a formal language to represent the chemical structure of molecules. And most importantly:

Programming languages are formal languages that have been designed to express computations.

Formal languages tend to have strict rules about syntax.

For example,

$$3+3=6$$

is a syntactically correct mathematical statement,
But

$$3=+6\$$$

is not.

$$H_2O$$

is a syntactically correct chemical name, but

$$_2Zz$$

is not.

Syntax rules come in two flavors, pertaining to **tokens** and **structure**.

Tokens are the basic elements of the language, such as words, numbers, and chemical elements.

One of the problems with 3=+6$ is that $ is not a legal token in mathematics

Similarly, $_2$Zz is not legal because there is no element with the abbreviation Zz.

The second type of syntax error pertains to the structure of a statement — that is, the way the tokens are arranged. T

he statement

3=+6$ I

s structurally illegal because you can't place a plus sign immediately after an equal sign.

Similarly, molecular formulas have to have subscripts after the element name, not before.

Parsing

When you read a sentence in English or a statement in a formal language, you have to figure out what the structure of the sentence is.

This process is called **parsing**.

For example, when you hear the sentence,
"The other shoe fell,"
you understand that "the other shoe" is the subject and "fell" is the predicate.

Once you have parsed a sentence, you can figure out what it means, or the semantics of the sentence. Assuming that you know what a shoe is and what it means to fall, you will understand the general implication of this sentence.

Although formal and natural languages have many features in common — tokens, structure, syntax, and semantics — there are many differences:

ambiguity
Natural languages are full of ambiguity, which people deal with by using contextual clues and other information. Formal languages are designed to be nearly or completely unambiguous, which means that any statement has exactly one meaning, regardless of context.

redundancy
In order to make up for ambiguity and reduce misunderstandings, natural languages employ lots of redundancy. As a result, they are often verbose. Formal languages are less redundant and more concise.

literalness

Natural languages are full of idiom and metaphor. If I say, "The other shoe fell," there is probably no shoe and nothing falling. Formal languages mean exactly what they say.

Reading a program

First, remember that formal languages are much more dense than natural languages, so it takes longer to read them.

Also, the structure is very important, so it is usually not a good idea to read from top to bottom, left to right. Instead, learn to parse the program in your head, identifying the tokens and interpreting the structure

Finally, the details matter. Little things like spelling errors and bad punctuation, which you can get away with in natural languages, can make a big difference in a formal language.

Variables, expresiones y enunciados

Values and types

A **value** is one of the fundamental things — like a letter or a number — that a program manipulates.

Values belong to different **types**: 2 is an integer, and 'Hello, World!' is a **string**, so-called because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

The print statement also works for integers.

```
>>> print 4
4
```

If you are not sure what type a value has, the interpreter can tell you.

```
>>> type('Hello, World!')
<type 'str'>
>>> type(17)

<type 'int'>
```

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called **floating-point**.

```
>>> type( 3.2 )
<type 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

```
>>> type( '17' )
<type 'str'>
>>> type( '3.2' )
<type 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is a legal expression:

```
>>> print 1,000,000
1 0 0
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated list of three integers, which it prints consecutively. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

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.

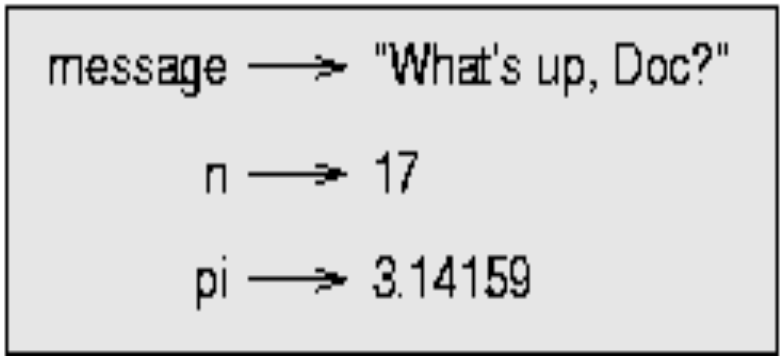
The **assignment statement** creates new variables and gives them values:

```
>>> message = "What's up, Doc?"
>>> n = 17
>>> pi = 3.14159
```

This example makes three assignments. The first assigns the string "What's up, Doc?" to a new variable named message. The second gives the integer 17 to n, and 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.

A common way to represent variables on paper is to write the name with an arrow pointing to the variable's value. This kind of figure is called a **state diagram** because it shows what state each of the variables is in (think of it as the variable's state of mind). This diagram shows the result of the assignment statements:



The print statement also works with variables.

```
>>> print message
What's up, Doc?
>>> 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

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.

In a script, an expression all by itself is a legal statement, but it doesn't do anything. The script

```
17
3.2
'Hello, World!'
1 + 1
```

produces no output at all

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    (python 2)
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.

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:

Parentheses 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.

Exponentiation has the next highest precedence, so 2**1+1 is 3 and not 4, and 3*1**3 is 3 and not 27.

Multiplication and **D**ivision have the same precedence, which is higher than

Addition 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
>>banana nut bread
```

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.

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.

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. In this case, it reminds the reader about the ever-surprising behavior of integer division.

Funciones

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'>
```

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. Returning to the example from the previous chapter, 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
0.98333333333
```

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

In mathematics, you have probably seen functions like sin and log, and you have learned to evaluate expressions like sin(pi/2) and log(1/x). First, you evaluate the expression in parentheses (the argument). For example, pi/2 is approximately 1.571, and 1/x is 0.1 (if x happens to be 10.0).

Then, you evaluate the function itself, either by looking it up in a table or by performing various computations.

The sin of 1.571 is 1, and the log of 0.1 is -1 (assuming that log indicates the logarithm base 10).

This process can be applied repeatedly to evaluate more complicated expressions like log(1/sin(pi/2))..

First, you evaluate the argument of the innermost function, then evaluate the function, and so on.

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)
```

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)
```

Adding new functions

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.

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.

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."
```

You should notice a few things about this program:
You can call the same procedure repeatedly. In fact, it is quite common and useful to do so.
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.

Definitions and use

Pulling together the code fragments, the whole program looks like this:

```
def newLine():
  print

def threeLines():
  newLine()
  newLine()
  newLine()

print "First Line."
threeLines()
print "Second Line."
```

This program contains two function definitions: newLine and threeLines. Function definitions get executed just like other statements, but the effect is to create the new function. The statements inside the function do not get executed until the function is called, and the function definition generates no output.

As you might expect, you have to create a function before you can execute it. In other words, the function definition has to be executed before the first time it is called.

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
```

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 = “An apple a day"
>>> chant2 = ” keeps the doctor away "
>>> catTwice(chant1, chant2)
An apple a day keeps the doctor away An apple a day keeps the doctor away
```

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.

Conditionals and 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.

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.

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.”
```

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
```

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. It 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).

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)
```