A Python 2.7 programming tutorial

Version 2.0

John W. Shipman

2013-07-07 12:18

Abstract

A tutorial for the Python programming language, release 2.7.

This publication is available in Web form[1] and also as a PDF document[2]. Please forward any comments to **tcc-doc@nmt.edu**.

Table of Contents

[1] http://www.nmt.edu/tcc/help/pubs/lang/pytut27/
[2] http://www.nmt.edu/tcc/help/pubs/lang/pytut27/pytut27.pdf

1. Introduction

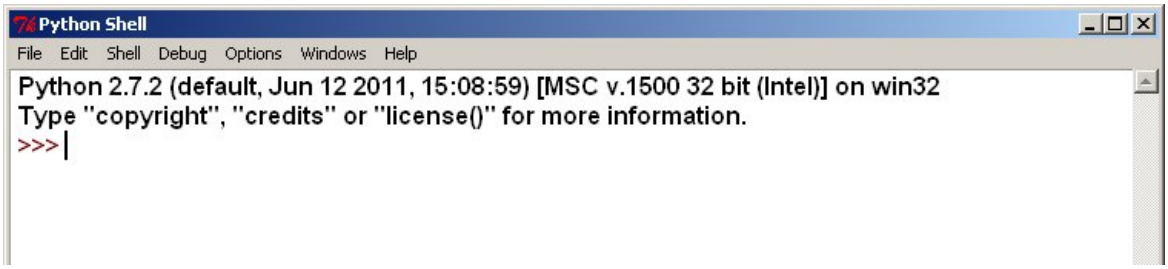
This document contains some tutorials for the Python programming language, as of Python version 2.7. These tutorials accompany the free Python classes taught by the New Mexico Tech Computer Center. Another good tutorial is at the Python website[3].

1.1. Starting Python in conversational mode

This tutorial makes heavy use of Python's conversational mode. When you start Python in this way, you will see an initial greeting message, followed by the prompt "`>>>`".

- On a TCC workstation in Windows, from the *Start* menu, select *All Programs → Python 2.7 → IDLE (Python GUI)*. You will see something like this:

[3] http://docs.python.org/tut/tut.html



- For Linux or MacOS, from a shell prompt (such as "`$`" for the `bash` shell), type:

```
python
```

You will see something like this:

```
$ python
Python 2.7.1 (r271:86832, Apr 12 2011, 16:15:16)
[GCC 4.6.0 20110331 (Red Hat 4.6.0-2)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

When you see the "`>>>`" prompt, you can type a Python expression, and Python will show you the result of that expression. This makes Python useful as a desk calculator. (That's why we sometimes refer to conversational mode as "calculator mode".) For example:

```
>>> 1+1
2
>>>
```

Each section of this tutorial introduces a group of related Python features.

2. Python's numeric types

Pretty much all programs need to do numeric calculations. Python has several ways of representing numbers, and an assortment of operators to operate on numbers.

2.1. Basic numeric operations

To do numeric calculations in Python, you can write expressions that look more or less like algebraic expressions in many other common languages. The "`+`" operator is addition; "`-`" is subtraction; use "`*`" to multiply; and use "`/`" to divide. Here are some examples:

```
>>> 99 + 1
100
>>> 1 - 99
-98
>>> 7 * 5
35
>>> 81 / 9
9
```

The examples in this document will often use a lot of extra space between the parts of the expression, just to make things easier to read. However, these spaces are not required:

```
>>> 99+1
100
>>> 1-99
-98
```

When an expression contains more than one operation, Python defines the usual order of operations, so that higher-precedence operations like multiplication and division are done before addition and subtraction. In this example, even though the multiplication comes after the addition, it is done first.

```
>>> 2 + 3 * 4
14
```

If you want to override the usual precedence of Python operators, use parentheses:

```
>>> (2+3)*4
20
```

Here's a result you may not expect:

```
>>> 1 / 5
0
```

You might expect a result of 0.2, not zero. However, Python has different kinds of numbers. Any number without a decimal point is considered an *integer*, a whole number. If any of the numbers involved contain a decimal point, the computation is done using *floating point* type:

```
>>> 1.0 / 4.0
0.25
>>> 1.0 / 3.0
0.33333333333333331
```

That second example above may also surprise you. Why is the last digit a one? In Python (and in pretty much all other contemporary programming languages), many real numbers cannot be represented exactly. The representation of 1.0/3.0 has a slight error in the seventeenth decimal place. This behavior may be slightly annoying, but in conversational mode, Python doesn't know how much precision you want, so you get a ridiculous amount of precision, and this shows up the fact that some values are approximations.

You can use Python's `print` statement to display values without quite so much precision:

```
>>> print 1.0/3.0
0.333333333333
```

It's okay to mix integer and floating point numbers in the same expression. Any integer values are coerced to their floating point equivalents.

```
>>> print 1.0/5
0.2
>>> print 1/5.0
0.2
```

Later we will learn about Python's string format method , which allows you to specify exactly how much precision to use when displaying numbers. For now, let's move on to some more of the operators on numbers.

The "%" operator between two numbers gives you the *modulo*. That is, the expression "*x % y*" returns the remainder when *x* is divided by *y*.

```
>>> 13 % 5
3
>>> 5.72 % 0.5
0.21999999999999975
>>> print 5.72 % 0.5
0.22
```

Exponentiation is expressed as "*x ** y*", meaning *x* to the *y* power.

```
>>> 2 ** 8
256
>>> 2 ** 30
1073741824
>>> 2.0 ** 0.5
1.4142135623730951
>>> 10.0 ** 5.2
158489.31924611141
>>> 2.0 ** 100
1.2676506002282294e+30
```

That last number, `1.2676506002282294e+30`, is an example of *exponential* or *scientific* notation. This number is read as "1.26765... times ten to the 30th power". Similarly, a number like `1.66e-24` is read as "1.66 times ten to the minus 24th power".

So far we have seen examples of the integer type, which is called `int` in Python, and the floating-point type, called the `float` type in Python. Python guarantees that `int` type supports values between -2,147,483,648 and 2,147,483,647 (inclusive).

There is another type called `long`, that can represent much larger integer values. Python automatically switches to this type whenever an expression has values outside the range of `int` values. You will see letter "`L`" appear at the end of such values, but they act just like regular integers.

```
>>> 2 ** 50
1125899906842624L
>>> 2 ** 100
1267650600228229401496703205376L
>>> 2 ** 1000
10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788156958581275946729175531468251871452856923140435984577574698574803934567774824230985421074605062371141877954182153046474983581941267398767559165543946077062914571196477686542167660429831652624386837205668069376L
```

2.2. The assignment statement

So far we have worked only with numeric constants and operators. You can attach a name to a value, and that value will stay around for the rest of your conversational Python session.

Python names must start with a letter or the underbar (_) character; the rest of the name may consist of letters, underbars, or digits. Names are case-sensitive: the name `Count` is a different name than `count`.

For example, suppose you wanted to answer the question, "how many days is a million seconds?" We can start by attaching the name `sec` to a value of a million:

```
>>> sec = 1e6
>>> sec
1000000.0
```

A statement of this type is called an *assignment statement*. To compute the number of minutes in a million seconds, we divide by 60. To convert minutes to hours, we divide by 60 again. To convert hours to days, divide by 24, and that is the final answer.

```
>>> minutes = sec / 60.0
>>> minutes
16666.666666666668
>>> hours=minutes/60
>>> hours
277.77777777777777
>>> days=hours/24.
>>> days
11.574074074074074
>>> print days, hours, minutes, sec
11.5740740741 277.777777778 16666.6666667 1000000.0
```

You can attach more than one name to a value. Use a series of names, separated by equal signs, like this.

```
>>> total = remaining = 50
>>> print total, remaining
50 50
```

The general form of an assignment statement looks like this:

```
name1 = name2 = ... = expression
```

Here are the rules for evaluating an assignment statement:

- Each $name_i$ is some Python variable name. Variable names must start with either a letter or the underbar (_) character, and the remaining characters must be letters, digits, or underbar characters. Examples: `skateKey`; `_x47`; `sum_of_all_fears`.
- The *expression* is any Python expression.
- When the statement is evaluated, first the *expression* is evaluated so that it is a single value. For example, if the *expression* is "`(2+3)*4`", the resulting single value is the integer `20`.

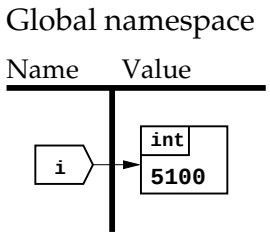
  Then all the names $name_i$ are *bound* to that value.

What does it mean for a name to be bound to a value? When you are using Python in conversational mode, the names and value you define are stored in an area called the *global namespace*. This area is like a two-column table, with names on the left and values on the right.

Here is an example. Suppose you start with a brand new Python session, and type this line:

```
>>> i = 5100
```

Here is what the global namespace looks like after the execution of this assignment statement.

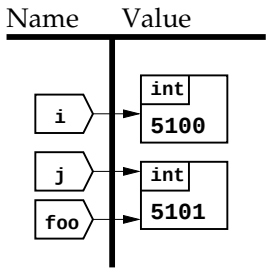


In this diagram, the value appearing on the right shows its type, `int` (integer), and the value, 5100.

In Python, values have types, but names are *not* associated with any type. A name can be bound to a value of any type at any time. So, a Python name is like a luggage tag: it identifies a value, and lets you retrieve it later.

Here is another assignment statement, and a diagram showing how the global namespace appears after the statement is executed.

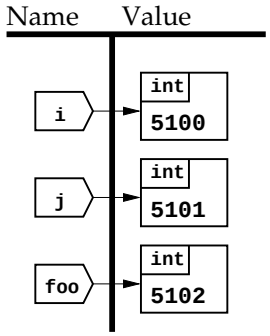
```
>>> j = foo = i + 1
```



The expression "`i + 1`" is equivalent to "`5100 + 1`", since variable `i` is bound to the integer 5100. This expression reduces to the integer value 5101, and then the names `j` and `foo` are both bound to that value. You might think of this situation as being like one piece of baggage with two tags tied to it.

Let's examine the global namespace after the execution of this assignment statement:

```
>>> foo = foo + 1
```



Because `foo` starts out bound to the integer value 5101, the expression "`foo + 1`" simplifies to the value 5102. Obviously, `foo = foo + 1` doesn't make sense in algebra! However, it is a common way for programmers to add one to a value.

Note that name `j` is still bound to its old value, 5101.

2.3. More mathematical operations

Python has a number of built-in functions. To call a function in Python, use this general form:

```
f(arg1, arg2, ... )
```

That is, use the function name, followed by an open parenthesis "`(`", followed by zero or more *arguments* separated by commas, followed by a closing parenthesis "`)`".

For example, the `round` function takes one numeric argument, and returns the nearest whole number (as a `float` number). Examples:

```
>>> round ( 4.1 )
4.0
>>> round(4.9)
5.0
>>> round(4.5)
5.0
```

The result of that last case is somewhat arbitrary, since 4.5 is equidistant from 4.0 and 5.0. However, as in most other modern programming languages, the value chosen is the one further from zero. More examples:

```
>>> round (-4.1)
-4.0
>>> round (-4.9)
-5.0
>>> round (-4.5)
-5.0
```

For historical reasons, trigonometric and transcendental functions are not built-in to Python. If you want to do calculations of those kinds, you will need to tell Python that you want to use the `math` package. Type this line:

```
>>> from math import *
```

Once you have done this, you will be able to use a number of mathematical functions. For example, `sqrt(`*x*`)` computes the square root of *x*:

```
>>> sqrt(4.0)
2.0
>>> sqrt(81)
9.0
>>> sqrt(100000)
316.22776601683796
```

Importing the `math` module also adds two predefined variables, `pi` (as in π) and `e`, the base of natural logarithms:

```
>>> print pi, e
3.14159265359 2.71828182846
```

Here's an example of a function that takes more than argument. The function `atan2(`*dy* , *dx*`)` returns the arctangent of a line whose slope is *dy/dx*.

```
>>> atan2 ( 1.0, 0.0 )
1.5707963267948966
```

```
>>> atan2(0.0, 1.0)
0.0
>>> atan2(1.0, 1.0)
0.78539816339744828
>>> print pi/4
0.785398163397
```

For a complete list of all the facilities in the `math` module, see the *Python quick reference*[4]. Here are some more examples; `log` is the natural logarithm, and `log10` is the common logarithm:

```
>>> log(e)
1.0
>>> log10(e)
0.43429448190325182
>>> exp ( 1.0 )
2.7182818284590451
>>> sin ( pi / 2 )
1.0
>>> cos(pi/2)
6.1230317691118863e-17
```

Mathematically, $\cos(\pi/2)$ should be zero. However, like pretty much all other modern programming languages, transcendental functions like this use approximations. 6.12×10^{-17} is, after all, pretty close to zero.

Two math functions that you may find useful in certain situations:

- `floor(`*x*`)` returns the largest whole number that is less than or equal to *x*.
- `ceil(`*x*`)` returns the smallest whole number that is greater than or equal to *x*.

```
>>> floor(4.9)
4.0
>>> floor(4.1)
4.0
>>> floor(-4.1)
-5.0
>>> floor(-4.9)
-5.0
>>> ceil(4.9)
5.0
>>> ceil(4.1)
5.0
>>> ceil(-4.1)
-4.0
>>> ceil(-4.9)
-4.0
```

Note that the `floor` function always moves toward -∞ (minus infinity), and `ceil` always moves toward +∞.

[4] http://www.nmt.edu/tcc/help/pubs/python/web/math-module.html

3. Character string basics

Python has extensive features for handling strings of characters. There are two types:

- A `str` value is a string of zero or more 8-bit characters. The common characters you see on North American keyboards all use 8-bit characters. The official name for this character set is ASCII[5], for American Standard Code for Information Interchange.

  This character set has one surprising property: all capital letters are considered less than all lowercase letters, so the string `"Z"` sorts before string `"a"`.

- A `unicode` value is a string of zero or more 32-bit Unicode characters. The Unicode character set covers just about every written language and almost every special character ever invented.

We'll mainly talk about working with `str` values, but most `unicode` operations are similar or identical, except that Unicode literals are preceded with the letter `u`: for example, `"abc"` is type `str`, but `u"abc"` is type `unicode`.

3.1. String literals

In Python, you can enclose string constants in either single-quote (`'...'`) or double-quote (`"..."`) characters.

```
>>> cloneName = 'Clem'
>>> cloneName
'Clem'
>>> print cloneName
Clem
>>> fairName = "Future Fair"
>>> print fairName
Future Fair
>>> fairName
'Future Fair'
```

When you display a string value in conversational mode, Python will usually use single-quote characters. Internally, the values are the same regardless of which kind of quotes you use. Note also that the `print` statement shows only the content of a string, without any quotes around it.

To convert an integer (`int` type) value *`i`* to its string equivalent, use the function "`str(i)`":

```
>>> str(-497)
'-497'
>>> str(000)
'0'
```

The inverse operation, converting a string *`s`* back into an integer, is written as "`int(s)`":

```
>>>
>>> int("-497")
-497
>>> int("-0")
0
>>> int ( "012this ain't no number" )
Traceback (most recent call last):
```

[5] http://en.wikipedia.org/wiki/ASCII

```
  File "<stdin>", line 1, in ?
ValueError: invalid literal for int(): 012this ain't no number
```

The last example above shows what happens when you try to convert a string that isn't a valid number.

To convert a string *s* containing a number in base *B*, use the form "`int(s, B)`":

```
>>> int ( '0F', 16 )
15
>>> int ( "10101", 2 )
21
>>> int ( "0177776", 8 )
65534
```

To obtain the 8-bit integer code contained in a one-character string *s*, use the function "`ord(s)`". The inverse function, to convert an integer *i* to the character that has code *i*, use "`chr(i)`". The numeric values of each character are defined by the ASCII[6] character set.

```
>>> chr( 97 )
'a'
>>> ord("a")
97
>>> chr(65)
'A'
>>> ord('A')
65
```

In addition to the printable characters with codes in the range from 32 to 127 inclusive, a Python string can contain any of the other unprintable, special characters as well. For example, the *null character*, whose official name is NUL, is the character whose code is zero. One way to write such a character is to use this form:

```
'\xNN'
```

where *NN* is the character's code in hexadecimal (base 16) notation.

```
>>> chr(0)
'\x00'
>>> ord('\x00')
0
```

Another special character you may need to deal with is the *newline* character, whose official name is LF (for "line feed"). Use the special *escape sequence* "`\n`" to produced this character.

```
>>> s = "Two-line\nstring."
>>> s
'Two-line\nstring.'
>>> print s
Two-line
string.
```

As you can see, when a newline character is displayed in conversational mode, it appears as "`\n`", but when you print it, the character that follows it will appear on the next line. The code for this character is 10:

[6] http://en.wikipedia.org/wiki/ASCII

```
>>> ord('\n')
10
>>> chr(10)
'\n'
```

Python has several other of these escape sequences. The term "escape sequence" refers to a convention where a special character, the "escape character", changes the meaning of the characters after it. Python's escape character is backslash (\).

Input	Code	Name	Meaning
\b	8	BS	*backspace*
\t	9	HT	*tab*
\"	34	"	Double quote
\'	39	'	Single quote
\\	92	\	Backslash

There is another handy way to get a string that contains newline characters: enclose the string within *three* pairs of quotes, either single or double quotes.

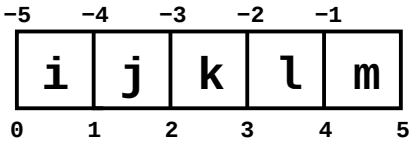
```
>>> multi = """This string
...   contains three
...   lines."""
>>> multi
'This string\n  contains three\n  lines.'
>>> print multi
This string
  contains three
  lines.
>>> s2 = '''
... xyz
... '''
>>> s2
'\nxyz\n'
>>> print s2

xyz

>>>
```

Notice that in Python's conversational mode, when you press *Enter* at the end of a line, and Python knows that your line is not finished, it displays a "`...`" prompt instead of the usual "`>>>`" prompt.

3.2. Indexing strings

To extract one or more characters from a string value, you have to know how positions in a string are numbered.

Here, for example, is a diagram showing all the positions of the string `'ijklm'`.



In the diagram above, the numbers show the positions *between* characters. Position 0 is the position before the first character; position 1 is the position between the first and characters; and so on.

You may also refer to positions relative to the *end* of a string. Position -1 refers to the position before the last character; -2 is the position before the next-to-last character; and so on.

To extract from a string *s* the character that occurs just *after* position *n*, use an expression of this form:

```
s[n]
```

Examples:

```
>>> stuff = 'ijklm'
>>> stuff[0]
'i'
>>> stuff[1]
'j'
>>> stuff[4]
'm'
>>> stuff [ -1 ]
'm'
>>> stuff [-3]
'k'
>>> stuff[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

The last example shows what happens when you specify a position after all the characters in the string.

You can also extract multiple characters from a string; see Section 4.3, “Slicing sequences” (p. 23).

3.3. String methods

Many of the operations on strings are expressed as *methods*. A method is sort of like a function that lives only inside values of a certain type. To call a method, use this syntax:

```
expr.method(arg1, arg2, ...)
```

where each arg_i is an argument to the method, just like an argument to a function.

For example, any string value has a method called `center` that produces a new string with the old value centered, using extra spaces to pad the value out to a given length. This method takes as an argument the desired new length. Here's an example:

```
>>> star = "*"
>>> star.center(5)
'  *  '
```

The following sections describe some of the more common and useful string methods.

3.3.1. `.center()`: Center some text

Given some string value *s*, to produce a new string containing *s* centered in a string of length *n*, use this method call:

```
s.center(n)
```

This method takes one argument *n*, the size of the result. Examples:

```
>>> k = "Ni"
>>> k.center(5)
'  Ni '
>>> "<*>".center(12)
'    <*>     '
```

Note that in the first example we are asking Python to center the string `"Ni"` in a field of length 5. Clearly we can't center a 2-character string in 5 characters, so Python arbitrarily adds the leftover space character before the old value.

3.3.2. `.ljust()` and `.rjust()`: Pad to length on the left or right

Another useful string method left-justifies a value in a field of a given length. The general form:

```
s.ljust(n)
```

For any string expression *s*, this method returns a new string containing the characters from *s* with enough spaces added after it to make a new string of length *n*.

```
>>> "Ni".ljust(4)
'Ni  '
>>> "Too long to fit".ljust(4)
'Too long to fit'
```

Note that the `.ljust()` method will never return a shorter string. If the length isn't enough, it just returns the original value.

There is a similar method that right-justifies a string value:

```
s.rjust(n)
```

This method returns a string with enough spaces added before the value to make a string of length *n*. As with the `.ljust()` method, it will never return a string shorter than the original.

```
>>> "Ni".rjust(4)
'  Ni'
>>> m = "floccinaucinihilipilification"
>>> m.rjust(4)
'floccinaucinihilipilification'
```

3.3.3. `.strip()`, `.lstrip()`, and `.rstrip()`: Remove leading and/or trailing whitespace

Sometimes you want to remove whitespace (spaces, tabs, and newlines) from a string. For a string *s*, use these methods to remove leading and trailing whitespace:

- *s*`.strip()` returns *s* with any leading or trailing whitespace characters removed.
- *s*`.lstrip()` removes only leading whitespace.

- `s.rstrip()` removes only trailing whitespace.

```
>>> spaceCase = ' \n \t Moon   \t\t '
>>> spaceCase
' \n \t Moon   \t\t '
>>> spaceCase.strip()
'Moon'
>>> spaceCase.lstrip()
'Moon   \t\t '
>>> spaceCase.rstrip()
' \n \t Moon'
```

3.3.4. `.count()`: How many occurrences?

The method `s.count(t)` searches string `s` to see how many times string `t` occurs in it.

```
>>> chiq = "banana"
>>> chiq
'banana'
>>> chiq.count("a")
3
>>> chiq.count("b")
1
>>> chiq.count("x")
0
>>> chiq.count("an")
2
>>> chiq.count("ana")
1
```

Note that this method does not count overlapping occurrences. Although the string `"ana"` occurs twice in string `"banana"`, the occurrences overlap, so `"banana".count("ana")` returns only 1.

3.3.5. `.find()` and `.rfind()`: Locate a string within a longer string

Use this method to search for a string `t` within a string `s`:

```
s.find(t)
```

If `t` matches any part of `s`, the method returns the position where the first match begins; otherwise, it returns -1.

```
>>> chiq
'banana'
>>> chiq.find ( "b" )
0
>>> chiq.find ( "a" )
1
>>> chiq.find ( "x" )
-1
>>> chiq.find ( "nan" )
2
```

If you need to find the *last* occurrence of a substring, use the similar method `s.rfind(t)`, which returns the position where the last match starts, or -1 if there is no match.

```
>>> chiq.rfind("a")
5
>>> chiq[5]
'a'
>>> chiq.rfind("n")
4
>>> chiq.rfind("b")
0
>>> chiq.rfind("Waldo")
-1
```

3.3.6. `.startswith()` and `.endswith()`

You can check to see if a string `s` starts with a string `t` using a method call like this:

```
s.startswith(t)
```

This method returns `True` if `s` starts with a string that matches `t`; otherwise it returns `False`.

```
>>> chiq
'banana'
>>> chiq.startswith("b")
True
>>> chiq.startswith("ban")
True
>>> chiq.startswith ( 'Waldo' )
False
```

There is a similar method `s.endswith(t)` that tests whether string `s` ends with `t`:

```
>>> chiq.endswith("Waldo")
False
>>> chiq.endswith("a")
True
>>> chiq.endswith("nana")
True
```

The special values `True` and `False` are discussed later in Section 6.1, "Conditions and the `bool` type" (p. 34).

3.3.7. `.lower()` and `.upper()`: Change the case of letters

The methods `s.lower()` and `s.upper()` are used to convert uppercase characters to lowercase, and vice versa, respectively.

```
>>> poet = 'E. E. Cummings'
>>> poet.upper()
'E. E. CUMMINGS'
>>> poet.lower()
'e. e. cummings'
```

3.3.8. Predicates for testing for kinds of characters

Use the string methods in this section to test whether a string contains certain kinds of characters. Each of these methods is a *predicate*, that is, it asks a question and returns a value of `True` or `False`.

- `s.isalpha()` tests whether all the characters of `s` are letters.
- `s.isupper()` tests whether all the letters of `s` are uppercase. (It ignores any non-letter characters.)
- `s.islower()` tests whether all the letters of `s` are lowercase letters. (This method also ignores non-letter characters.)
- `s.isdigit()` tests whether all the characters of `s` are digits.

```
>>> mixed = 'abcDEFghi'
>>> mixed.isalpha()
True
>>> mixed.isupper()
False
>>> mixed.islower()
False
>>> "ABCDGOLDFISH".isupper()
True
>>> "lmno goldfish".islower()
True
>>> "abc $%&*(".islower()
True
>>> "abC $%&*(".islower()
False
>>> paradise = "87801"
>>> paradise.isalpha()
False
>>> paradise.isdigit()
True
>>> "abc123".isdigit()
False
```

3.3.9. `.split()`: Break fields out of a string

The `.split()` method is used to break a string up into pieces wherever a certain string called the *delimiter* is found; it returns a list of strings containing the text between the delimiters. For example, suppose you have a string that contains a series of numbers separated by whitespace. A call to the `.split()` method on that string, with no arguments, returns a list of the parts of the string that are surrounded by whitespace.

```
>>> line = "  1.4   8.6  -23.49   "
>>> line.split()
['1.4', '8.6', '-23.49']
```

You can also specify a delimiter as the argument of the `.split()` method. Examples:

```
>>> s = 'farcical/aquatic/ceremony'
>>> s.split('/')
['farcical', 'aquatic', 'ceremony']
>>> "//a/b/".split('/')
['', '', 'a', 'b', '']
```

```
>>> "Stilton; Wensleydale; Cheddar;Edam".split("; ")
['Stilton', 'Wensleydale', 'Cheddar;Edam']
```

You may also provide a second argument that limits the number of pieces to be split from the string.

```
>>> t = 'a/b/c/d/e'
>>> t.split('/')
['a', 'b', 'c', 'd', 'e']
>>> t.split('/', 1)
['a', 'b/c/d/e']
>>> t.split('/', 3)
['a', 'b', 'c', 'd/e']
```

3.4. The string format method

One of the commonest string operations is to combine fixed text and variable values into a single string. For example, maybe you have a variable named `nBananas` that contains the number of bananas, and you want to format a string something like `"We have 27 bananas today"`. Here's how you do it:

```
>>> nBananas = 54
>>> "We have {0} bananas today".format(nBananas)
'We have 54 bananas today'
```

Here is the general form of the string format operation:

S`.format(`p_0`, `p_1`, ..., `k_0`=`e_0`, `k_1`=`e_1`, ...)`

In this form:

- S is a format string that specifies the fixed parts of the desired text and also tells where the variable parts are to go and how they are to look.
- The `.format()` method takes zero or more positional arguments p_i followed by and zero or more keyword arguments k_i=e_i, where each k_i is any Python name and each e_i is any Python expression.
- The format string contains a mixture of ordinary text and *format codes*. Each of the format codes is enclosed in braces `{...}`. A format code containing a number refers to the corresponding positional argument, and a format code containing a name refers to the corresponding keyword argument.

Examples:

```
>>> "We have {0} bananas.".format(27)
'We have 27 bananas.'
>>> "We have {0} cases of {1} today.".format(42, 'peaches')
'We have 42 cases of peaches today.'
>>> "You'll have {count} new {thing}s by {date}".format(
...     count=27, date="St. Swithin's Day", thing="cooker")
"You'll have 27 new cookers by St. Swithin's Day"
```

You can control the formatting of an item by using a format code of the form "`{`N`:`*`type`*`}`", where N is the number or name of the argument to the `.format()` method, and *`type`* specifies the details of the formatting.

The *`type`* may be a single type code like `s` for string, `d` for integer, or `f` for float.

```
>>> "{0:d}".format(27)
'27'
```

```
>>> "{0:f}".format(27)
'27.000000'
>>> "{animal:s}".format(animal="sheep")
'sheep'
```

You may also include a field size just before the type code. With `float` values, you can also specify a precision after the field size by using a "`.`" followed by the desired number of digits after the decimal.

```
>>> "({bat:8s})".format(bat='fruit')
'(fruit   )'
>>> "{0:8f}".format(1.0/7.0)
'0.142857'
>>> "{n:20.11f}".format(n=1.0/7.0)
'       0.14285714286'
>>> "{silly:50.40f}".format(silly=5.33333)
'        5.3333300000000001261923898709937930107117'
```

Notice in the last example above that it is possible for you to produce any number of spurious digits beyond the precision used to specify the number originally! Beware, because those extra digits are utter garbage.

When you specify a precision, the value is rounded to the nearest value with that precision.

```
>>> "{0:.1f}".format(0.999)
'1.0'
>>> "{0:.1f}".format(0.99)
'1.0'
>>> "{0:.1f}".format(0.9)
'0.9'
>>> "{0:.1f}".format(0.96)
'1.0'
>>> "{0:.1f}".format(0.9501)
'1.0'
>>> "{0:.1f}".format(0.9499999)
'0.9'
```

The "`e`" type code forces exponential notation. You may also wish to use the "`g`" (for general) type code, which selects either float or exponential notation depending on the value.

```
>>> avo = 6.022e23
>>> "{0:e}".format(avo)
'6.022000e+23'
>>> "{0:.3e}".format(avo)
'6.022e+23'
>>> "{num:g}".format(num=144)
'144'
>>> "{num:g}".format(num=avo)
'6.022e+23'
```

By default, strings are left-justified within the field size and numbers are right-justified. You can change this by placing an alignment code just after the "`:`": "`<`" to left-align the field, "`^`" to center it, and "`>`" to right-align it.

```
>>> "/{0:<6s}/".format('git')
'/git   /'
```

```
>>> "/{0:^6s}/".format('git')
'/ git  /'
>>> "/{0:>6s}/".format('git')
'/   git/'
>>> '*{count:<8d}*'.format(count=13)
'*13      *'
```

Normally, short values are padded to length with spaces. You can specify a different padding character by placing it just after the "`:`".

```
>>> "{0:08d}".format(17)
'00000017'
"{film:@>20s}".format(film='If')
'@@@@@@@@@@@@@@@@@@If'
>>> "{film:@^20s}".format(film='If')
'@@@@@@@@@If@@@@@@@@@'
```

If you need to produce any "`{`" or "`}`" characters in the result, you must double them within the format code.

```
>>> "Set {0}: contents {{red, green, blue}}".format('glory')
'Set glory: contents {red, green, blue}'
```

One thing we sometimes need to is to format something to a size that is not known until the program is running. For example, suppose we want to format a ticket number from a variable named `ticket_no`, with left zero fill, and the width is given by a variable named `how_wide`. This would do the job:

```
>>> how_wide = 8
>>> ticket_no = 147
>>> "Ticket {num:0{w}d}".format(num=ticket_no, w=how_wide)
'Ticket 00000147'
```

Here, where the width is expected, "`{w}`" appears. Because there is a keyword argument that is effectively `w=8`, the value `"8"` is used for the width.

Note

The string `.format()` method has been available only since Python 2.6. If you are looking at older code, you may see a different technique using the "`%`" operator. For example, `'Attila the %s' % 'Bun'` yields `'Attila the bun'`. For an explanation, see the Python library documentation[7]. However, the old format operator is deprecated.

4. Sequence types

Mathematically, a *sequence* in Python represents an ordered set.

Sequences are an example of *container classes*: values that contain other values inside them.

Type name	Contains	Examples	Mutable?
`str`	8-bit characters	`"abc" 'abc' "" '' '\n' '\x00'`	No

[7] http://docs.python.org/library/stdtypes.html#string-formatting-operations

Type name	Contains	Examples	Mutable?
`unicode`	32-bit characters	`u'abc' u'\u000c'`	No
`list`	Any values	`[23, "Ruth", 69.8] []`	Yes
`tuple`	Any values	`(23, "Ruth", 69.8) () (44,)`	No

- `str` and `unicode` are used to hold text, that is, strings of characters.
- `list` and `tuple` are used for sequences of zero or more values of any type. Use a `list` if the contents of the sequence may change; use a `tuple` if the contents will not change, and in certain places where tuples are required. For example, the right-hand argument of the string format operator (see Section 3.4, "The string format method" (p. 18)) must be a tuple if you are formatting more than one value.
- To create a list, use an expression of the form

  ```
  [ expr1, expr1, ... ]
  ```

  with a list of zero or more values between *square brackets*, "`[…]`".
- To create a tuple, use an expression of the form

  ```
  ( expr1, expr1, ... )
  ```

  with a list of zero or more values enclosed in *parentheses*, "`(…`").

  To create a tuple with only one element *v*, use the special syntax "`(v,)`". For example, `(43+1,)` is a one-element tuple containing the integer 44. The trailing comma is used to distinguish this case from the expression "`(43+1)`", which yields the integer 44, not a tuple.
- *Mutability*: You can't change *part* of an immutable value. For example, you can't change the first character of a string from `'a'` to `'b'`. It is, however, easy to build a new string out of pieces of other strings.

Here are some calculator-mode examples. First, we'll create a string named `s`, a list named `L`, and a tuple named `t`:

```
>>> s = "abcde"
>>> L = [0, 1, 2, 3, 4, 5]
>>> t = ('x', 'y')
>>> s
'abcde'
>>> L
[0, 1, 2, 3, 4, 5]
>>> t
('x', 'y')
```

4.1. Functions and operators for sequences

The built-in function `len(S)` returns the number of elements in a sequence *S*.

```
>>> print len(s), len(L), len(t)
5 6 2
```

Function `max(S)` returns the largest value in a sequence *S*, and function `min(S)` returns the smallest value in a sequence *S*.

```
>>> max(L)
5
>>> min(L)
0
>>> max(s)
'e'
>>> min(s)
'a'
```

To test for set membership, use the "`in`" operator. For a value *v* and a sequence *S*, the expression `v in S` returns the Boolean value `True` if there is at least one element of *S* that equals *v*; it returns `False` otherwise.

```
>>> 2 in L
True
>>> 77 in L
False
```

There is an inverse operator, `v not in S`, that returns `True` if *v* does not equal any element of *S*, `False` otherwise.

```

>>> 2 not in L
False
>>> 77 not in L
True
```

The "`+`" operator is used to concatenate two sequences of the same type.

```
>>> s + "xyz"
'abcdexyz'
>>> L + L
[0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 4, 5]
>>> t + ('z',)
('x', 'y', 'z')
```

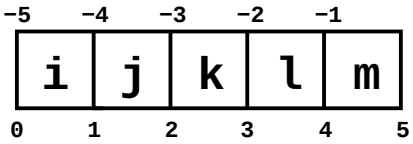
When the "`*`" operator occurs between a sequence *S* and an integer *n*, you get a new sequence containing *n* repetitions of the elements of *S*.

```
>>> "x" * 5
'xxxxx'
>>> "spam" * 8
'spamspamspamspamspamspamspamspam'
>>> [0, 1] * 3
[0, 1, 0, 1, 0, 1]
```

4.2. Indexing the positions in a sequence

Positions in a sequence refer to locations *between* the values. Positions are numbered from left to right starting at 0. You can also refer to positions in a sequence using negative numbers to count from right to left: position -1 is the position before the last element, position -2 is the position before the next-to-last element, and so on.

Here are all the positions of the string `"ijklm"`.



To extract a single element from a sequence, use an expression of the form `S[i]`, where `S` is a sequence, and `i` is an integer value that selects the element just *after* that position.

```
>>> s[0]
'a'
>>> s[4]
'e'
>>> s[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: string index out of range
```

The last line is an error; there is nothing after position 5 in string `s`.

```
>>> L[0]
0
>>> t[0]
'x'
```

4.3. Slicing sequences

For a sequence *S*, and two positions *B* and *E* within that sequence, the expression `S [ B : E ]` produces a new sequence containing the elements of S between those two positions.

```
>>> L
[0, 1, 2, 3, 4, 5]
>>> L[2]
2
>>> L[4]
4
>>> L[2:4]
[2, 3]
>>> s = 'abcde'
>>> s[2]
'c'
>>> s[4]
'e'
>>> s[2:4]
'cd'
```

Note in the example above that the elements are selected from position 2 to position 4, which does *not* include the element `L[4]`.

You may omit the starting position to slice elements from at the beginning of the sequence up to the specified position. You may omit the ending position to specify a slice that extends to the end of the sequence. You may even omit both in order to get a copy of the whole sequence.

```
>>> L[:4]
[0, 1, 2, 3]
```

```
>>> L[4:]
[4, 5]
>>> L[:]
[0, 1, 2, 3, 4, 5]
```

You can replace part of a list by using a slicing expression on the *left-hand* side of the "=" in an assignment statement, and providing a list of replacement elements on the right-hand side of the "=". The elements selected by the slice are deleted and replaced by the elements from the right-hand side.

In slice assignment, it is not necessary that the number of replacement elements is the same as the number of replaced elements. In this example, the second and third elements of `L` are replaced by the five elements from the list on the right-hand side.

```
>>> L
[0, 1, 2, 3, 4, 5]
>>> L[2:4]
[2, 3]
>>> L[2:4] = [93, 94, 95, 96, 97]
>>> L
[0, 1, 93, 94, 95, 96, 97, 4, 5]
```

You can even delete a slice from a sequence by assigning an an empty sequence to a slice.

```
>>> L
[0, 1, 93, 94, 95, 96, 97, 4, 5]
>>> L[3]
94
>>> L[7]
4
>>> L[3:7] = []
>>> L
[0, 1, 93, 4, 5]
```

Similarly, you can insert elements into a sequence by using an empty slice on the left-hand side.

```
>>> L
[0, 1, 93, 4, 5]
>>> L[2]
93
>>> L[2:2] = [41, 43, 47, 53]
>>> L
[0, 1, 41, 43, 47, 53, 93, 4, 5]
```

Another way to delete elements from a sequence is to use Python's `del` statement.

```
>>> L
[0, 1, 41, 43, 47, 53, 93, 4, 5]
>>> L[3:6]
[43, 47, 53]
>>> del L[3:6]
>>> L
[0, 1, 41, 93, 4, 5]
```

4.4. Sequence methods

To find the position of a value *V* in a sequence *S*, use this method:

```
S.index(V)
```

This method returns the position of the first element of *S* that equals *V*. If no elements of *S* are equal, Python raises a `ValueError` exception.

```
>>> menu1
['beans', 'kale', 'tofu', 'trifle', 'sardines']
>>> menu1.index("kale")
1
>>> menu1.index("spam")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
```

The method `S.count(V)` method returns the number of elements of *S* that are equal to *V*.

```
>>> menu1[2:2] = ['spam'] * 3
>>> menu1
['beans', 'kale', 'spam', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.count('gravy')
0
>>> menu1.count('spam')
3
>>> "abracadabra".count("a")
5
>>> "abracadabra".count("ab")
2
>>> (1, 6, 55, 6, 55, 55, 8).count(55)
3
```

4.5. List methods

All list instances have methods for changing the values in the list. These methods work only on lists. They do not work on the other sequence types that are not mutable, that is, the values they contain may not be changed, added, or deleted.

For example, for any list instance *L*, the `.append(v)` method appends a new value *v* to that list.

```
>>> menu1 = ['kale', 'tofu']
>>> menu1
['kale', 'tofu']
>>> menu1.append ( 'sardines' )
>>> menu1
['kale', 'tofu', 'sardines']
>>>
```

To insert a single new value *V* into a list *L* at an arbitrary position *P*, use this method:

```
L.insert(P, V)
```

To continue the example above:

```
>>> menu1
['kale', 'tofu', 'sardines']
>>> menu1.insert(0, 'beans')
>>> menu1
['beans', 'kale', 'tofu', 'sardines']
>>> menu1[3]
'sardines'
>>> menu1.insert(3, 'trifle')
>>> menu1
['beans', 'kale', 'tofu', 'trifle', 'sardines']
```

The method `L.remove(V)` removes the first element of *L* that equals *V*, if there is one. If no elements equal *V*, the method raises a `ValueError` exception.

```
>>> menu1
['beans', 'kale', 'spam', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('spam')
>>> menu1
['beans', 'kale', 'spam', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('spam')
>>> menu1
['beans', 'kale', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.remove('gravy')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.remove(x): x not in list
```

The `L.sort()` method sorts the elements of a list into ascending order.

```
>>> menu1
['beans', 'kale', 'spam', 'tofu', 'trifle', 'sardines']
>>> menu1.sort()
>>> menu1
['beans', 'kale', 'sardines', 'spam', 'tofu', 'trifle']
```

Note that the `.sort()` method itself does not return a value; it sorts the values of the list in place. A similar method is `.reverse()`, which reverses the elements in place:

```
>>> menu1
['beans', 'kale', 'sardines', 'spam', 'tofu', 'trifle']
>>> menu1.reverse()
>>> menu1
['trifle', 'tofu', 'spam', 'sardines', 'kale', 'beans']
```

4.6. The `range()` function: creating arithmetic progressions

The term *arithmetic progression* refers to a sequence of numbers such that the difference between any two successive elements is the same. Examples: [1, 2, 3, 4, 5]; [10, 20, 30, 40]; [88, 77, 66, 55, 44, 33].

Python's built-in `range()` function returns a list containing an arithmetic progression. There are three different ways to call this function.

To generate the sequence [0, 1, 2, ..., *n*-1], use the form `range(n)`.

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
>>> range(2)
[0, 1]
>>> range(0)
[]
```

Note that the sequence will never include the value of the argument *n*; it stops one value short.

To generate a sequence [*i*, *i*+1, *i*+2, ..., *n*-1], use the form `range(i, n)`:

```
>>> range(5,11)
[5, 6, 7, 8, 9, 10]
>>> range(1,5)
[1, 2, 3, 4]
```

To generate an arithmetic progression with a difference *d* between successive values, use the three-argument form `range(i, n, d)`. The resulting sequence will be [*i*, *i*+*d*, *i*+2**d*, ...], and will stop before it reaches a value equal to *n*.

```
>>> range ( 10, 100, 10 )
[10, 20, 30, 40, 50, 60, 70, 80, 90]
>>> range ( 100, 0, -10 )
[100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
>>> range ( 8, -1, -1 )
[8, 7, 6, 5, 4, 3, 2, 1, 0]
```

4.7. One value can have multiple names

It is necessary to be careful when modifying mutable values such as lists because there may be more than one name bound to that value. Here is a demonstration.

We start by creating a list of two strings and binding two names to that list.

```
>>> menu1 = menu2 = ['kale', 'tofu']
>>> menu1
['kale', 'tofu']
>>> menu2
['kale', 'tofu']
```

Then we make a new list using a slice that selects all the elements of `menu1`:

```
>>> menu3 = menu1 [ : ]
>>> menu3
['kale', 'tofu']
```

Now watch what happens when we modify `menu1`'s list:

```
>>> menu1.append ( 'sardines' )
>>> menu1
['kale', 'tofu', 'sardines']
>>> menu2
```

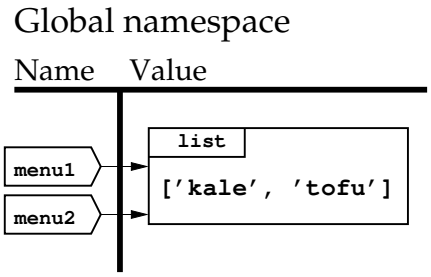
```
['kale', 'tofu', 'sardines']
>>> menu3
['kale', 'tofu']
```

If we appended a third string to `menu1`, why does that string also appear in list `menu2`? The answer lies in the definition of Python's assignment statement:

> To evaluate an assignment statement of the form
>
> $V_1 = V_2 = ... = expr$
>
> where each V_i is a variable, and *expr* is some expression, first reduce *expr* to a single value, then bind each of the names v_i to that value.

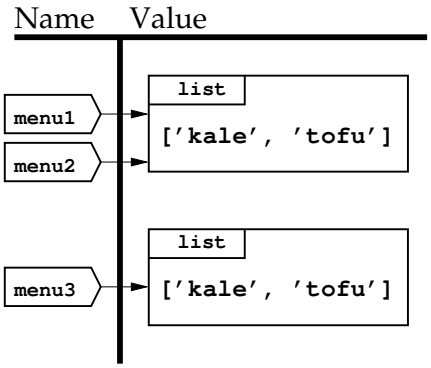
So let's follow the example one line at a time, and see what the global namespace looks like after each step. First we create a list instance and bind two names to it:

```
>>> menu1=menu2=['kale', 'tofu']
```



Two different names, `menu1` and `menu2`, point to the same list. Next, we create an element-by-element copy of that list and bind the name `menu3` to the copy.

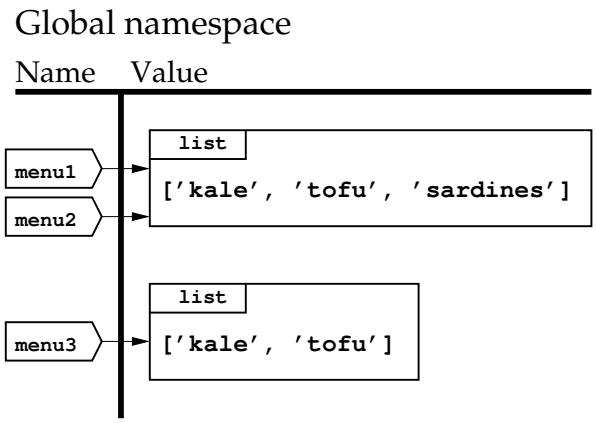
```
>>> menu3 = menu1[:]
>>> menu3
['kale', 'tofu']
```



So, when we add a third string to `menu1`'s list, the name `menu2` is still bound to that same list.

```
>>> menu1.append ( 'sardines' )
>>> menu1
['kale', 'tofu', 'sardines']
```

```
>>> menu2
['kale', 'tofu', 'sardines']
```



This behavior is seldom a problem in practice, but it is important to keep in mind that two or more names can be bound to the same value.

If you are concerned about modifying a list when other names may be bound to the same list, you can always make a copy using the slicing expression "*L*`[:]`".

```
>>> L1 = ['bat', 'cat']
>>> L2 = L1
>>> L3 = L1[:]
>>> L1.append('hat')
>>> L2
['bat', 'cat', 'hat']
>>> L3
['bat', 'cat']
```

5. Dictionaries

Python's dictionary type is useful for many applications involving table lookups. In mathematical terms:

> A Python dictionary is a set of zero or more ordered pairs (*key*, *value*) such that:
>
> - The *value* can be any type.
> - Each *key* may occur only once in the dictionary.
> - No *key* may be mutable. In particular, a key may not be a list or dictionary, or a tuple containing a list or dictionary, and so on.

The idea is that you store values in a dictionary associated with some key, so that later you can use that key to retrieve the associated value.

5.1. Operations on dictionaries

The general form used to create a new dictionary in Python looks like this:

```
{k1: v1,  k2: v2,  ...}
```

To retrieve the value associated with key *k* from dictionary *d*, use an expression of this form:

```
d[k]
```

Here are some conversational examples:

```
>>> numberNames = {0:'zero', 1:'one', 10:'ten', 5:'five'}
>>> numberNames[10]
'ten'
>>> numberNames[0]
'zero'
>>> numberNames[999]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 999
```

Note that when you try to retrieve the value for which no key exists in the dictionary, Python raises a `KeyError` exception.

To add or replace the value for a key *k* in dictionary *d*, use an assignment statement of this form:

```
d[k] = v
```

For example:

```
>>> numberNames[2] = "two"
>>> numberNames[2]
'two'
>>> numberNames
{0: 'zero', 1: 'one', 10: 'ten', 2: 'two', 5: 'five'}
```

Note

The ordering of the pairs within a dictionary is undefined. Note that in the example above, the pairs do not appear in the order they were added.

You can use strings, as well as many other values, as keys:

```
>>> nameNo={"one":1, "two":2, "forty-leven":4011}
>>> nameNo["forty-leven"]
4011
```

You can test to see whether a key *k* exists in a dictionary *d* with the "`in`" operator, like this:

```
k in d
```

This operation returns `True` if *k* is a key in dictionary *d*, `False` otherwise.

The construct "*k* `not in` *d*" is the inverse test: it returns `True` if *k* is *not* a key in *d*, `False` if it *is* a key.

```
>>> 1 in numberNames
True
>>> 99 in numberNames
False
>>> "forty-leven" in nameNo
```

```
True
>>> "eleventeen" in nameNo
False
>>> "forty-leven" not in nameNo
False
>>> "eleventeen" not in nameNo
True
```

Python's `del` (delete) statement can be used to remove a key-value pair from a dictionary.

```
>>> numberNames
{0: 'zero', 1: 'one', 10: 'ten', 2: 'two', 5: 'five'}
>>> del numberNames[10]
>>> numberNames
{0: 'zero', 1: 'one', 2: 'two', 5: 'five'}
>>> numberNames[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 10
```

5.2. Dictionary methods

A number of useful methods are defined on any Python dictionary. To test whether a key *k* exists in a dictionary *d*, use this method:

```
d.has_key(k)
```

This is the equivalent of the expression "`k in d`": it returns `True` if the key is in the dictionary, `False` otherwise.

```
>>> numberNames
{0: 'zero', 1: 'one', 2: 'two', 5: 'five'}
>>> numberNames.has_key(2)
True
>>> numberNames.has_key(10)
False
```

To get a list of all the keys in a dictionary *d*, use this expression:

```
d.keys()
```

To get a list of the values in a dictionary *d* , use this expression:

```
d.values()
```

You can get all the keys and all the values at the same time with this expression, which returns a list of 2-element tuples, in which each tuple has one key and one value as `(k, v)`.

```
d.items()
```

Examples:

```
>>> numberNames
{0: 'zero', 1: 'one', 2: 'two', 5: 'five'}
>>> numberNames.keys()
```

```
[0, 1, 2, 5]
>>> numberNames.values()
['zero', 'one', 'two', 'five']
>>> numberNames.items()
[(0, 'zero'), (1, 'one'), (2, 'two'), (5, 'five')]
>>> nameNo
{'forty-leven': 4011, 'two': 2, 'one': 1}
>>> nameNo.keys()
['forty-leven', 'two', 'one']
>>> nameNo.values()
[4011, 2, 1]
>>> nameNo.items()
[('forty-leven', 4011), ('two', 2), ('one', 1)]
```

Here is another useful method:

```
d.get(k)
```

If *k* is a key in *d*, this method returns `d[k]`. However, if *k* is not a key, the method returns the special value `None`. The advantage of this method is that if the *k* is not a key in *d*, it is not considered an error.

```
>>> nameNo.get("two")
2
>>> nameNo.get("eleventeen")
>>> huh = nameNo.get("eleventeen")
>>> print huh
None
```

Note that when you are in conversational mode, and you type an expression that results in the value `None`, nothing is printed. However, the `print` statement will display the special value `None` visually as the example above shows.

There is another way to call the `.get()` method, with two arguments:

```
d.get(k, default)
```

In this form, if key *k* exists, the corresponding value is returned. However, if *k* is not a key in *d*, it returns the *default* value.

```
>>> nameNo.get("two", "I have no idea.")
2
>>> nameNo.get("eleventeen", "I have no idea.")
'I have no idea.'
```

Here is another useful dictionary method. This is similar to the two-argument form of the `.get()` method, but it goes even further: if the key is not found, it stores a default value in the dictionary.

```
d.setdefault(k, default)
```

If key *k* exists in dictionary *d*, this expression returns the value `d[k]`. If *k* is not a key, it creates a new dictionary entry as if you had said "`d[k] = default`".

```
>>> nameNo.setdefault("two", "Unknown")
2
>>> nameNo["two"]
2
```

```
>>> nameNo.setdefault("three", "Unknown")
'Unknown'
>>> nameNo["three"]
'Unknown'
```

To merge two dictionaries *d1* and *d2*, use this method:

```
d1.update(d2)
```

This method adds all the key-value pairs from *d2* to *d1*. For any keys that exist in both dictionaries, the value after this operation will be the value from *d2*.

```
>>> colors = { 1: "red", 2: "green", 3: "blue" }
>>> moreColors = { 3: "puce", 4: "taupe", 5: "puce" }
>>> colors.update ( moreColors )
>>> colors
{1: 'red', 2: 'green', 3: 'puce', 4: 'taupe', 5: 'puce'}
```

Note in the example above that key 3 was in both dictionaries, but after the `.update()` method call, key 3 is related to the value from `moreColors`.

5.3. A namespace is like a dictionary

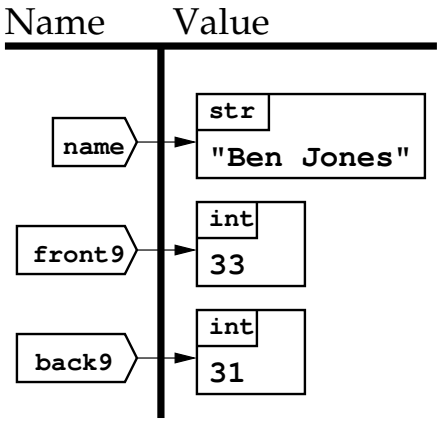
Back in Section 2.2, "The assignment statement" (p. 5), we first encountered the idea of a *namespace*. When you start up Python in conversational mode, the variables and functions you define live in the "global namespace".

We will see later on that Python has a number of different namespaces in addition to the global namespace. Keep in mind that namespaces are very similar to dictionaries:

- The names are like the keys of a dictionary: each one is unique.
- The values bound to those names are like the values in a dictionary. They can be any value of any type.

We can even use the same picture for a dictionary that we use to illustrate a namespace. Here is a small dictionary and a picture of it:

```
d = { 'name': 'Ben Jones', 'front9': 33, 'back9': 31 }
```



6. Branching

By default, statements in Python are executed sequentially. *Branching* statements are used to break this sequential pattern.

- Sometimes you want to perform certain operations only in some cases. This is called a *conditional branch*.
- Sometimes you need to perform some operations repeatedly. This is called *looping*.

Before we look at how Python does conditional branching, we need to look at Python's Boolean type.

6.1. Conditions and the `bool` type

Boolean algebra is the mathematics of true/false decisions. Python's `bool` type has only two values: `True` and `False`.

A typical use of Boolean algebra is in comparing two values. In Python, the expression *x* `<` *y* is `True` if *x* is less than *y*, `False` otherwise.

```
>>> 2 < 5
True
>>> 2 < 2
False
>>> 2 < 0
False
```

Here are the six comparison operators:

Math symbol	Python	Meaning
<	<	Less than
≤	<=	Less than or equal to
>	>	Greater than
≥	>=	Greater than or equal to
≡	==	Equal to
≠	!=	Not equal to

The operator that compares for equality is "==". (The "=" symbol is not an operator: it is used only in the assignment statement.)

Here are some more examples:

```
>>> 2 <= 5
True
>>> 2 <= 2
True
>>> 2 <= 0
False
>>> 4.9 > 5
False
>>> 4.9 > 4.8
True
>>> (2-1)==1
```

```
True
>>> 4*3 != 12
False
```

Python has a function `cmp(x, y)` that compares two values and returns:

- Zero, if *x* and *y* are equal.
- A negative number if *x* < *y*.
- A positive number if *x* > *y*.

```
>>> cmp(2,5)
-1
>>> cmp(2,2)
0
>>> cmp(2,0)
1
```

The function `bool(x)` converts any value *x* to a Boolean value. The values in this list are considered `False`; any other value is considered `True`:

- Any numeric zero: `0`, `0L`, or `0.0`.
- Any empty sequence: `""` (an empty string), `[]` (an empty list), `()` (an empty tuple).
- `{}` (an empty dictionary).
- The special unique value `None`.

```
>>> print bool(0), bool(0L), bool(0.0), bool(''), bool([]), bool(())
False False False False False False
>>> print bool({}), bool(None)
False False
>>> print bool(1), bool(98.6), bool('Ni!'), bool([43, "hike"])
True True True True
```

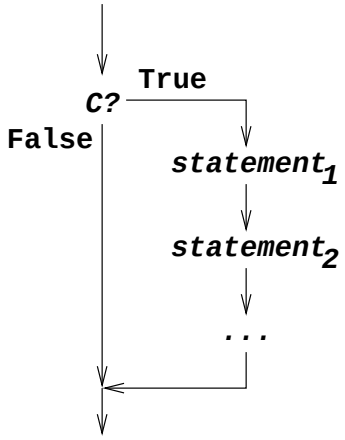
6.2. The `if` statement

The purpose of an `if` statement is to perform certain actions only in certain cases.

Here is the general form of a simple "one-branch" `if` statement. In this case, if some condition *C* is true, we want to execute some sequence of statements, but if *C* is not true, we don't want to execute those statements.

```
if C:
    statement_1
    statement_2
    ...
```

Here is a picture showing the flow of control through a simple `if` statement. Old-timers will recognize this as a *flowchart*.



There can be any number of statements after the `if`, but they must all be indented, and all indented the same amount. This group of statements is called a *block*.

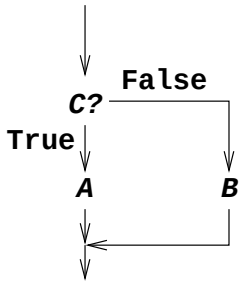
When the `if` statement is executed, the condition *C* is evaluated, and converted to a `bool` value (if it isn't already Boolean). If that value is `True`, the block is executed; if the value is `False`, the block is skipped.

Here's an example:

```
>>> half = 0.5
>>> if half > 0:
...     print "Half is better than none."
...     print "Burma!"
...
Half is better than none.
Burma!
```

Sometimes you want to do some action *A* when `C` is true, but perform some different action *B* when `C` is false. The general form of this construct is:

```
if C:
    block A
    ...
else:
    block B
    ...
```



As with the single-branch `if`, the condition *C* is evaluated and converted to Boolean. If the result is `True`, *block A* is executed; if `False`, *block B* is executed instead.

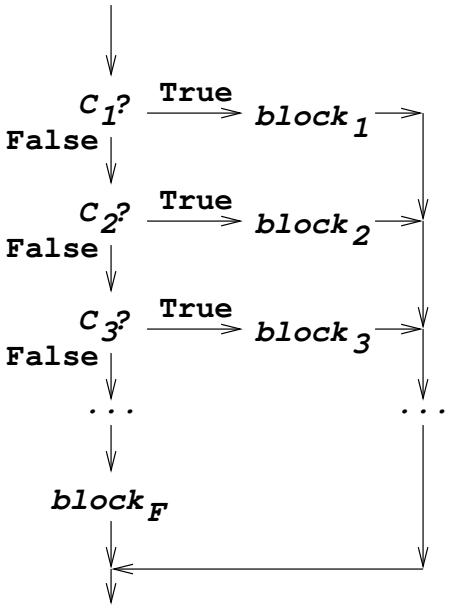
```
>>> half = 0.5
>>> if half > 0:
...     print "Half is more than none."
... else:
...     print "Half is not much."
...     print "Ni!"
...
Half is more than none.
```

Some people prefer a more "horizontal" style of coding, where more items are put on the same line, so as to take up less vertical space. If you prefer, you can put one or more statements on the same line as the `if` or `else`, instead of placing them in an indented block. Use a semicolon "`;`" to separate multiple statements. For example, the above example could be expressed on only two lines:

```
>>> if half > 0: print "Half is more than none."
... else: print "Half is not much."; print "Ni!"
...
Half is more than none.
```

Sometimes you want to execute only one out of three or four or more blocks, depending on several conditions. For this situation, Python allows you to have any number of "`elif` clauses" after an `if`, and before the `else` clause if there is one. Here is the most general form of a Python `if` statement:

```
if C1:
    block1
elif C2:
    block2
elif C3:
    block3
...
else:
    blockF
    ...
```



So, in general, an `if` statement can have zero or more `elif` clauses, optionally followed by an `else` clause. Example:

```
>>> i = 2
>>> if i==1: print "One"
... elif i==2: print "Two"
... elif i==3: print "Three"
... else: print "Many"
...
Two
```

You can have blocks within blocks. Here is an example:

```
>>> x = 3
>>> if  x >= 0:
...     if (x%2) == 0:
...         print "x is even"
...     else:
...         print "x is odd"
... else:
...     print "x is negative"
...
x is odd
```

6.3. A word about indenting your code

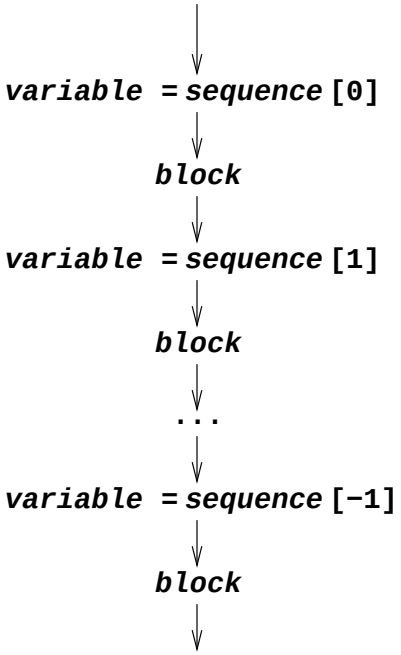
One of the most striking innovations of Python is the use of indentation to show the structure of the blocks of code, as in the `if` statement. Not everyone is thrilled by this feature. However, it is generally good practice to indent subsidiary clauses; it makes the code more readable. Those who argue that they should be allowed to violate this indenting practice are, in the author's opinion, arguing against what is generally regarded as a good practice.

The amount by which you indent each level is a matter of personal preference. You can use a *tab* character for each level of indention; tab stops are assumed to be every 8th character. Beware mixing tabs with spaces, however; the resulting errors can be difficult to diagnose.

6.4. The `for` statement: Looping

Use Python's "`for`" construct to do some repetitive operation for each member of a sequence. Here is the general form:

```
for variable in sequence:
    block
    ...
```



- The *sequence* can be any expression that evaluates to a sequence value, such as a list or tuple. The `range()` function is often used here to generate a sequence of integers.
- For each value in the *sequence* in turn, the *variable* is set to that value, and the *block* is executed.

  As with the `if` statement, the block consists of one or more statements, indented the same amount relative to the `if` keyword.

This example prints the cubes of all numbers from 1 through 5.

```
>>> for n in range(1,6):
...     print "The cube of {0} is {1}".format(n, n**3)
...
The cube of 1 is 1
The cube of 2 is 8
The cube of 3 is 27
The cube of 4 is 64
The cube of 5 is 125
```

You may put the body of the loop—that is, the statements that will be executed once for each item in the sequence—on the same line as the "`for`" if you like. If there are multiple statements in the body, separate them with semicolons.

```
>>> for n in range(1,6): print "{0}**3={1}".format(n, n**3),
...
1**3=1 2**3=8 3**3=27 4**3=64 5**3=125
>>> if 1 > 0: print "Yes";print "One is still greater than zero"
...
Yes
One is still greater than zero
```

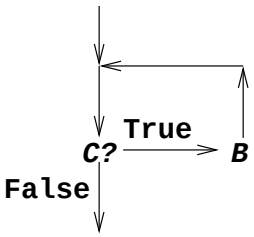
Here is an another example of iteration over a list of specific values.

```
>>> for  s in ('a', 'e', 'i', 'o', 'u'):
...     word  =  "st" + s + "ck"
...     print  "Pick up the", word
...
Pick up the stack
Pick up the steck
Pick up the stick
Pick up the stock
Pick up the stuck
```

6.5. The `while` statement

Use this statement when you want to perform a block *B* as long as a condition *C* is true:

```
while C:
    B
    ...
```



Here is how a `while` statement is executed.

1. Evaluate *C*. If the result is true, go to step 2. If it is false, the loop is done, and control passes to the statement after the end of *B*.
2. Execute block *B*.
3. Go back to step 1.

Here is an example of a simple `while` loop.

```
>>> i = 1
>>> while i < 100:
...     print i,
...     i = i * 2
...
1 2 4 8 16 32 64
```

This construct has the potential to turn into an *infinite loop*, that is, one that never terminates. Be sure that the body of the loop does something that will eventually make the loop terminate.

6.6. Special branch statements: `break` and `continue`

Sometimes you need to exit a `for` or `while` loop without waiting for the normal termination. There are two special Python branch statements that do this:

- If you execute a `break` statement anywhere inside a `for` or `while` loop, control passes out of the loop and on to the statement after the end of the loop.

- A `continue` statement inside a `for` loop transfers control back to the top of the loop, and the variable is set to the next value from the sequence if there is one. (If the loop was already using the last value of the sequence, the effect of `continue` is the same as `break`.)

Here are examples of those statements.

```
>>> i = 0
>>> while  i < 100:
...     i  =  i + 3
...     if  ( i % 5 ) == 0:
...         break
...     print i,
...
3 6 9 12
```

In the example above, when the value of `i` reaches 15, which has a remainder of 0 when divided by 5, the `break` statement exits the loop.

```
>>> for  i in range(500, -1, -1):
...     if (i % 100) != 0:
...         continue
...     print i,
...
500 400 300 200 100 0
```

7. How to write a self-executing Python script

So far we have used Python's conversational mode to demonstrate all the features. Now it's time to learn how to write a complete program.

Your program will live in a file called a *script*. To create your script, use your favorite text editor (*emacs*, *vi*, *Notepad*, whatever), and just type your Python statements into it.

How you make it executable depends on your operating system.

- On Windows platforms, be sure to give your script file a name that ends in "`.py`". If Python is installed, double-clicking on any script with this ending will use Python to run the script.
- Under Linux and MacOS X, the first line of your script must look like this:

  ```
  #!pythonpath
  ```

  The *`pythonpath`* tells the operating system where to find Python. This path will usually be "`/usr/local/bin/python`", but you can use the "`which`" shell command to find the path on your computer:

  ```
  $ which python
  /usr/local/bin/python
  ```

  Once you have created your script, you must also use this command to make it executable:

  ```
  chmod +x your-script-name
  ```

Here is a complete script, set up for a typical Linux installation. This script, `powersof2`, prints a table showing the values of 2^n and 2^{-n} for n in the range 1, 2, ..., 12.

```
#!/usr/local/bin/python
print "Table of powers of two"
print
print "{0:>10s} {1:>2s} {2:s}".format("2**n", "n", "2**(-n)")
for n in range(13):
    print "{0:10d} {1:2d} {2:17.15f}".format(2**n, n, 2.0**(-n))
```

Here we see the invocation of this script under the `bash` shell, and the output:

```
$ ./powersof2
Table of powers of two

      2**n  n 2**(-n)
         1  0 1.000000000000000
         2  1 0.500000000000000
         4  2 0.250000000000000
         8  3 0.125000000000000
        16  4 0.062500000000000
        32  5 0.031250000000000
        64  6 0.015625000000000
       128  7 0.007812500000000
       256  8 0.003906250000000
       512  9 0.001953125000000
      1024 10 0.000976562500000
      2048 11 0.000488281250000
      4096 12 0.000244140625000
```

8. `def`: Defining functions

You can define your own functions in Python with the `def` statement.

- Python functions can act like mathematical functions such as `len(s)`, which computes the length of `s`. In this example, values like `s` that are passed to the function are called *parameters* to the function.
- However, more generally, a Python function is just a container for some Python statements that do some task. A function can take any number of parameters, even zero.

Here is the general form of a Python function definition. It consists of a `def` statement, followed by an indented block called the *body* of the function.

```
def name ( arg0, arg1, ... ):
    block
```

The parameters that a function expects are called *arguments* inside the body of the function.

Here's an example of a function that takes no arguments at all, and does nothing but print some text.

```
>>> def pirateNoises():
...     for arrCount in range(7):
...         print "Arr!",
...
>>>
```

To call this function:

```
>>> pirateNoises()
Arr! Arr! Arr! Arr! Arr! Arr! Arr!
>>>
```

To call a function in general, use an expression of this form:

```
name ( param0, param1, ... )
```

- The name of the function is followed by a left parenthesis "`(`", a list of zero or more *parameter* values separated by commas, then a right parenthesis "`)`".
- The parameter values are substituted for the corresponding arguments to the function. The value of parameter $param_0$ is substituted for argument arg_0; $param_1$ is substituted for arg_1 ; and so forth.

Here's a simple example showing argument substitution.

```
>>> def grocer(nFruits, fruitKind):
...     print "Stock: {0} cases of {1}".format(nFruits, fruitKind)
...
>>> grocer ( 37, 'kale' )
Stock: 37 cases of kale
>>> grocer(0,"bananas")
Stock: 0 cases of bananas
```

8.1. `return`: Returning values from a function

So far we have seen some simple functions that take arguments or don't take arguments. How do we define functions like `len()` that return a value?

Anywhere in the body of your function, you can write a `return` statement that terminates execution of the function and returns to the statement where it was called.

Here is the general form of this statement:

```
return expression
```

The *`expression`* is evaluated, and its value is returned to the caller.

Here is an example of a function that returns a value:

```
>>> def square(x):
...     return x**2
...
>>> square(9)
81
>>> square(2.5)
6.25
>>>
```

- You can omit the *`expression`*, and just use a statement of this form:

  ```
  return
  ```

  In this case, the special placeholder value `None` is returned.
- If Python executes your function body and never encounters a `return` statement, the effect is the same as a `return` with no value: the special value `None` is returned.

Here is another example of a function that returns a value. This function computes the factorial of a positive integer:

> The factorial of *n*, denoted *n*!, is defined as the product of all the integers from 1 to *n* inclusive.

For example, 4! = 1×2×3×4 = 24.

We can define the factorial function *recursively* like this:

- If *n* is 0 or 1, *n*! is 1.
- If *n* is greater than 1, *n*! = n × (n-1)!.

And here is a recursive Python function that computes the factorial, and a few examples of its use.

```
>>> def fact(n):
...     if n <= 1:
...         return 1
...     else:
...         return n * fact(n-1)
...
>>> for i in range(5):
...     print i, fact(i)
...
0 1
1 1
2 2
3 6
4 24
>>> fact(44)
2658271574788448768043625811014615890319638528000000000L
>>>
```

8.2. Function argument list features

The general form of a `def` shown in Section 8, "`def`: Defining functions" (p. 42) is over-simplified. In general, the argument list of a function is a sequence of four kinds of arguments:

1. If the argument is just a name, it is called a *positional* argument. There can be any number of positional arguments, including zero.
2. You can supply a default value for the argument by using the form "*`name=value`*". Such arguments are called *keyword* arguments. See Section 8.3, "Keyword arguments" (p. 45).

   A function can have any number of keyword arguments, including zero.

   All keyword arguments must follow any positional arguments in the argument list.
3. Sometimes it is convenient to write a function that can accept any number of positional arguments. To do this, use an argument of this form:

   ```
   * name
   ```

   A function may have only one such argument, and it must follow any positional or keyword arguments. For more information about this feature, see Section 8.4, "Extra positional arguments" (p. 46).
4. Sometimes it is also convenient to write a function that can accept any number of keyword arguments, not just the specific keyword arguments. To do this, use an argument of this form:

```
** name
```

If a function has an argument of this form, it must be the last item in the argument list. For more information about this feature, see Section 8.5, "Extra keyword arguments" (p. 46).

8.3. Keyword arguments

If you want to make some of the arguments to your function optional, you must supply a default value. In the argument list, this looks like "*name=value*".

Here's an example of a function with one argument that has a default value. If you call it with no arguments, the name `mood` has the string value `'bleah'` inside the function. If you call it with an argument, the name `mood` has the value you supply.

```
>>> def report(mood='bleah'):
...     print "My mood today is", mood
...
>>> report()
My mood today is bleah
>>> report('hyper')
My mood today is hyper
>>>
```

If your function has multiple arguments, and the caller supplies multiple parameters, here is how they are matched up:

- The function call must supply at least as many parameters as the function has positional arguments.
- If the caller supplies more positional parameters than the function has positional arguments, parameters are matched with keyword arguments according to their position.

Here are some examples showing how this works.

```
>>> def f(a, b="green", c=3.5):
...     print a, b, c
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() takes at least 1 argument (0 given)
>>> f(47)
47 green 3.5
>>> f(47, 48)
47 48 3.5
>>> f(47, 48, 49)
47 48 49
>>> f(47, 48, 49, 50)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() takes at most 3 arguments (4 given)
>>>
```

Here is another feature: the caller of a function can supply what are called *keyword parameters* of the form "*name=value*". If the function has an argument with a matching keyword, that argument will be set to *value*.

- If a function's caller supplies both positional and keyword parameters, all positional parameters must precede all keyword parameters.
- Keyword parameters may occur in any order.

Here are some examples of calling a function with keyword parameters.

```
>>> def g(p0, p1, k0="K-0", k1="K-1"):
...     print p0, p1, k0, k1
...
>>> g(33,44)
33 44 K-0 K-1
>>> g(33,44,"K-9","beep")
33 44 K-9 beep
>>> g(55,66,k1="whirr")
55 66 K-0 whirr
>>> g(7,8,k0="click",k1="clank")
7 8 click clank
>>>
```

8.4. Extra positional arguments

You can declare your function in such a way that it will accept any number of positional parameters. To do this, use an argument of the form "*`*name`*" in your argument list.

- If you use this special argument, it must follow all the positional and keyword arguments in the list.
- When the function is called, this name will be bound to a tuple containing any positional parameters that the caller supplied, over and above parameters that corresponded to other parameters.

Here is an example of such a function.

```
>>> def h(i, j=99, *extras):
...     print i, j, extras
...
>>> h(0)
0 99 ()
>>> h(1,2)
1 2 ()
>>> h(3,4,5,6,7,8,9)
3 4 (5, 6, 7, 8, 9)
>>>
```

8.5. Extra keyword arguments

You can declare your function in such a way that it can accept any number of keyword parameters, in addition to any keyword arguments you declare.

To do this, place an argument of the form "**`*name`*" last in your argument list.

When the function is called, that name is bound to a dictionary that contains any keyword-type parameters that are passed in that have names that don't match your function's keyword-type arguments. In that dictionary, the keys are the names used by the caller, and the values are the values that the caller passed.

Here's an example.

```
>>> def k(p0, p1, nickname='Noman', *extras, **extraKeys):
...     print p0, p1, nickname, extras, extraKeys
...
>>> k(1,2,3)
1 2 3 () {}
>>> k(4,5)
4 5 Noman () {}
>>> k(6, 7, hobby='sleeping', nickname='Sleepy', hatColor='green')
6 7 Sleepy () {'hatColor': 'green', 'hobby': 'sleeping'}
>>> k(33, 44, 55, 66, 77, hometown='McDonald', eyes='purple')
33 44 55 (66, 77) {'hometown': 'McDonald', 'eyes': 'purple'}
>>>
```

8.6. Documenting function interfaces

Python has a preferred way to document the purpose and usage of your functions. If the first line of a function body is a string constant, that string constant is saved along with the function as the *documentation string*. This string can be retrieved by using an expression of the form `f.__doc__`, where `f` is the function name.

Here's an example of a function with a documentation string.

```
>>> def pythag(a, b):
...     """Returns the hypotenuse of a right triangle with sides a and b.
...     """
...     return (a*a + b*b)**0.5
...
>>> pythag(3,4)
5.0
>>> pythag(1,1)
1.4142135623730951
>>> print pythag.__doc__
Returns the hypotenuse of a right triangle with sides a and b.

>>>
```

9. Using Python modules

Once you start building programs that are more than a few lines long, it's critical to apply this overarching principle to programming design:

> **Important**
>
> Divide and conquer.

In other words, rather than build your program as one large blob of Python statements, divide it into logical pieces, and divide the pieces into smaller pieces, until the pieces are each small enough to understand.

Python has many tools to help you divide and conquer. In Section 8, "`def`: Defining functions" (p. 42), we learned how to package up a group of statements into a function, and how to call that function and retrieve the result.

Way back in Section 2.3, "More mathematical operations" (p. 8), we got our first look at another important tool, Python's *module* system. Python does not have a built-in function to compute square roots, but there is a built-in module called `math` that includes a function `sqrt()` that computes square roots.

In general, a module is a package of functions and variables that you can import and use in your programs. Python comes with a large variety of modules, and you can also create your own. Let's look at Python's module system in detail.

- In Section 9.1, "Importing items from modules" (p. 48), we learn to import items from existing modules.
- Section 9.2, "Import entire modules" (p. 49) shows another way to use items from modules.
- Section 9.4, "Build your own modules" (p. 51).

9.1. Importing items from modules

Back in Section 2.2, "The assignment statement" (p. 5), we learned that there is an area called the "global namespace," where Python keeps the names and values of the variables you define.

The Python `dir()` function returns a list of all the names that are currently defined in the global namespace. Here is a conversational example; suppose you have just started up Python in conversational mode.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> frankness = 0.79
>>> dir()
['__builtins__', '__doc__', '__name__', 'frankness']
>>> def oi():
...     print "Oi!"
...
>>> dir()
['__builtins__', '__doc__', '__name__', 'frankness', 'oi']
>>> type(frankness)
<type 'float'>
>>> type(oi)
<type 'function'>
>>>
```

When Python starts up, three variables are always defined: `__builtins__`, `__doc__`, and `__name__`. These variables are for advanced work and needn't concern us now.

Note that when we define a variable (`frankness`), next time we call `dir()`, that name is in the resulting list. When we define a function (`oi`), its name is also added. Note also that you can use the `type()` function to find the type of any currently defined name: `frankness` has type `float`, and `oi` has type `function`.

Now let's see what happens when we import the contents of the `math` module into the global namespace:

```
>>> from math import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2',
'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod',
'frankness', 'frexp', 'hypot', 'ldexp', 'log', 'log10', 'modf', 'oi', 'pi',
 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> sqrt(64)
8.0
```

```
>>> pi*10.0
31.415926535897931
>>> cos(0.0)
1.0
>>>
```

As you can see, the names we have defined (`oi` and `frankness`) are still there, but all of the variables and functions from the `math` module are now in the namespace, and we can use its functions and variables like `sqrt()` and `pi`.

In general, an `import` statement of this form copies all the functions and variables from the module into the current namespace:

```
from someModule import *
```

However, you can also be selective about which items you want to import. Use a statement of this form:

```
from someModule import item1, item2, ...
```

where the keyword `import` is followed by a list of names, separated by commas.

Here's another example. Assume that you have just started a brand new Python session, and you want to import only the `sqrt()` function and the constant `pi`:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> from math import sqrt, pi
>>> dir()
['__builtins__', '__doc__', '__name__', 'pi', 'sqrt']
>>> sqrt(25.0)
5.0
>>> cos(0.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'cos' is not defined
>>>
```

We didn't ask for the `cos()` function to be imported, so it is not part of the global namespace.

9.2. Import entire modules

Some modules have hundreds of different items in them. In cases like that, you might not want to clutter up your global namespace with all those items. There is another way to import a module. Here is the general form:

```
import moduleName
```

This statement adds only one name to the current namespace—the name of the module itself. You can then refer to any item inside that module using an expression of this form:

```
moduleName.itemName
```

Here is an example, again using the built-in `math` module. Assume that you have just started up a new Python session and you have added nothing to the namespace yet.

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math']
>>> type(math)
<type 'module'>
>>> math.sqrt(121.0)
11.0
>>> math.pi
3.1415926535897931
>>> math.cos(0.0)
1.0
>>>
```

As you can see, using this form of `import` adds only one name to the namespace, and that name has type `module`.

There is one more additional feature of `import` we should mention. If you want to import an entire module M_1, but you want to refer to its contents using a different name M_2, use a statement of this form:

```
import M1 as M2
```

An example:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> import math as crunch
>>> dir()
['__builtins__', '__doc__', '__name__', 'crunch']
>>> type(crunch)
<type 'module'>
>>> crunch.pi
3.1415926535897931
>>> crunch.sqrt(888.888)
29.81422479287362
>>>
```

You can apply Python's built-in `dir()` function to a module object to find out what names are defined inside it:

```
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math']
>>> dir(math)
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp',
'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh']
>>>
```

9.3. A module is a namespace

Modules are yet another example of a Python namespace, just as we've discussed in Section 2.2, "The assignment statement" (p. 5) and Section 5.3, "A namespace is like a dictionary" (p. 33).

When you import a module using the form "`import moduleName`", you can refer to some name *N* inside that module using the period operator: "*moduleName*`.`*N*".

So, like any other namespace, a module is a container for a unique set of names, and the values to which each name is connected.

9.4. Build your own modules

If you have a common problem to solve, chances are very good that there are modules already written that will reduce the amount of code you have to write.

- Python comes with a large collection of built-in modules. See the *Python Library Reference*[8].
- The `python.org` site also hosts a collection of thousands of third-party modules: see the *Python package index*[9].

You can also build your own modules. A module is similar to a script (see Section 7, "How to write a self-executing Python script" (p. 41)): it is basically a text file containing the definitions of Python functions and variables.

To build your own module, use a common text editor to create a file with a name of the form "*moduleName*`.py`". The *moduleName* you choose must be a valid Python name—it must start with a letter or underbar, and consist entirely of letters, underbars, and digits.

Inside that file, place Python function definitions and ordinary assignment statements.

Here is a very simple module containing one function and one variable. It lives in a file named `cuber.py`.

```
def cube(x):
    return x**3

cubeVersion = "1.9.33"
```

Here is an example interactive session that uses that module:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> from cuber import *
>>> dir()
['__builtins__', '__doc__', '__name__', 'cube', 'cubeVersion']
>>> cube(3)
27
>>> cubeVersion
'1.9.33'
>>>
```

There is one more refinement we suggest for documenting the contents of a module. If the first line of the module's file is a string constant, it is saved as the module's "documentation string." If you later import such a module using the form "`import moduleName`", you can retrieve the contents of the documentation string using the expression "*moduleName*`.__doc__`".

[8] http://docs.python.org/lib/lib.html
[9] http://pypi.python.org/pypi

Here is an expanded version of our `cuber.py` with a documentation string:

```
"""cuber.py:  Simple homemade Python module

  Contents:
    cube(x):  Returns the cube of x
    cubeVersion:  Current version number of this module
"""
def cube(x):
    return x**3

cubeVersion = "1.9.33"
```

Finally, an example of how to retrieve the documentation string:

```
>>> import cuber
>>> print cuber.__doc__
cuber.py:  Simple homemade Python module

  Contents:
    cube(x):  Returns the cube of x
    cubeVersion:  Current version number of this module

>>> cuber.cube(10)
1000
>>>
```

10. Input and output

Python makes it easy to read and write files. To work with a file, you must first open it using the built-in `open()` function. If you are going to read the file, use the form "`open(`*`filename`*`)`", which returns a *file object*. Once you have a file object, you can use a variety of methods to perform operations on the file.

10.1. Reading files

For example, for a file object *`F`*, the method *`F`*`.readline()` attempts to read and return the next line from that file. If there are no lines remaining, it returns an empty string.

Let's start with a small text file named `trees` containing just three lines:

```
yew
oak
alligator juniper
```

Suppose this file lives in your current directory. Here is how you might read it one line at a time:

```
>>> treeFile = open ( 'trees' )
>>> treeFile.readline()
'yew\n'
>>> treeFile.readline()
'oak\n'
```

```
>>> treeFile.readline()
'alligator juniper\n'
>>> treeFile.readline()
''
```

Note that the newline characters (`'\n'`) are included in the return value. You can use the string `.rstrip()` method to remove trailing newlines, but beware: it also removes any other trailing whitespace.

```
>>> 'alligator juniper\n'.rstrip()
'alligator juniper'
>>> 'eat all my trailing spaces         \n'.rstrip()
'eat all my trailing spaces'
```

To read all the lines in a file at once, use the `.readlines()` method. This returns a list whose elements are strings, one per line.

```
>>> treeFile=open("trees")
>>> treeFile.readlines()
['yew\n', 'oak\n', 'alligator juniper\n']
```

A more general method for reading files is the `.read()` method. Used without any arguments, it reads the entire file and returns it to you as one string.

```
>>> treeFile = open ("trees")
>>> treeFile.read()
'yew\noak\nalligator juniper\n'
```

To read exactly *N* characters from a file *F*, use the method `F.read(N)`. If *N* characters remain in the file, you will get them back as an *N*-character string. If fewer than *N* characters remain, you will get the remaining characters in the file (if any).

```
>>> treeFile = open ( "trees" )
>>> treeFile.read(1)
'y'
>>> treeFile.read(5)
'ew\noa'
>>> treeFile.read(50)
'k\nalligator juniper\n'
>>> treeFile.read(80)
''
```

One of the easiest ways to read the lines from a file is to use a `for` statement. Here is an example:

```
>>> >>> treeFile=open('trees')
>>> for treeLine in treeFile:
...     print treeLine.rstrip()
...
yew
oak
alligator juniper
```

As with the `.readline()` method, when you iterate over the lines of a file in this way, the lines will contain the newline characters. If the above example did not trim these lines with `.rstrip()`, each line of output would be followed by a blank line, because the `print` statement adds a newline.

10.2. File positioning for random-access devices

For random-access devices such as disk files, there are methods that let you find your current position within a file, and move to a different position.

- `F.tell()` returns your current position in file *F*.
- `F.seek(N)` moves your current position to *N*, where a position of zero is the beginning of the file.
- `F.seek(N, 1)` moves your current position by a distance of *N* characters, where positive values of *N* move toward the end of the file and negative values move toward the beginning.

  For example, `F.seek(80, 1)` would move the file position 80 characters further from the start of the file.
- `F.seek(N, 2)` moves to a position *N* characters relative to the end of the file. For example, `F.seek(0, 2)` would move to the end of the file; `F.seek(-200, 2)` would move your position to 200 bytes before the end of the file.

```
>>> treeFile = open ( "trees" )
>>> treeFile.tell()
0L
>>> treeFile.read(6)
'yew\noa'
>>> treeFile.tell()
6L
>>> treeFile.seek(1)
>>> treeFile.tell()
1L
>>> treeFile.read(5)
'ew\noa'
>>> treeFile.tell()
6L
>>> treeFile.seek(1, 1)
>>> treeFile.tell()
7L
>>> treeFile.seek(-3, 1)
>>> treeFile.tell()
4L
>>> treeFile.seek(0, 2)
>>> treeFile.tell()
26L
>>> treeFile.seek(-3, 2)
>>> treeFile.tell()
23L
>>> treeFile.read()
'er\n'
```

10.3. Writing files

To create a disk file, open the file using a statement of this general form:

```
F = open ( filename, "w" )
```

The second argument, `"w"`, specifies write access. If possible, Python will create a new, empty file by that name. If there is an existing file by that name, and if you have write permission to it, the existing file will be deleted.

To write some content to the file you are creating, use this method:

```
F.write(s)
```

where `s` is any string expression.

Warning

The data you have sent to a file with the `.write()` method may not actually appear in the disk file until you close it by calling the `.close()` method on the file.

This is due to a mechanism called *buffering*. Python accumulates the data you have sent to the file, until a certain amount is present, and then it "flushes" that data to the physical file by writing it. Python also flushes the data to the file when you close it.

If you would like to make sure that the data you have written to the file is actually physically present in the file without closing it, call the `.flush()` method on the file object.

```
>>> sports = open ( "sportfile", "w" )
>>> sports.write ( "tennis\nrugby\nquoits\n" )
>>> sports.close()
>>> sportFile = open ( "sportfile" )
>>> sportFile.readline()
'tennis\n'
>>> sportFile.readline()
'rugby\n'
>>> sportFile.readline()
'quoits\n'
>>> sportFile.readline()
''
```

Here is a lengthy example demonstrating the action of the `.flush()` method.

```
>>> sporting = open('sports', 'w')
>>> sporting.write('golf\n')
>>> echo = open('sports')
>>> echo.read()
''
>>> echo.close()
>>> sporting.flush()
>>> echo = open('sports')
>>> echo.read()
'golf\n'
>>> echo.close()
>>> sporting.write('soccer')
>>> sporting.close()
>>> open('sports').read()
'golf\nsoccer'
```

Note that you must explicitly provide newline characters in the arguments to `.write()`.

11. Introduction to object-oriented programming

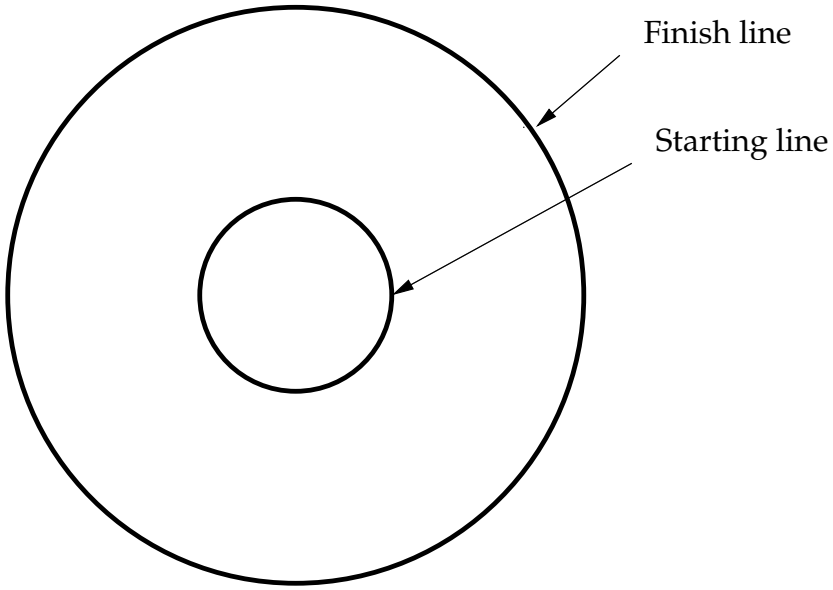
So far we have used a number of Python's built-in types such as `int`, `float`, `list`, and `file`.

Now it is time to begin exploring some of the more serious power of Python: the ability to create your own types.

This is a big step, so let's start by reviewing some of the historical development of computer language features.

11.1. A brief history of snail racing technology

An entrepreneur name Malek Ology would like to develop a service to run snail races to help non-profit organizations raise funds. Here is the proposed design for Malek's snail-racing track:



At the start of the race, the snails, with their names written on their backs in organic, biodegradable ink, are placed inside the starting line, and Malek starts a timer. As each snail crosses the finish line, Malek records their times.

Malek wants to write a Python program to print the race results. We'll look at the evolution of such a program through the history of programming. Let's start around 1960.

11.2. Scalar variables

Back around 1960, the hot language was FORTRAN. A lot of the work in this language was done using *scalar variables*, that is, a set of variable names, each of which held one number.

Suppose we've just had a snail race, and Judy finished in 87.3 minutes, while Kyle finished in 96.6 minutes. We can create Python variables with those values like this:

```
>>> judy = 87.3
>>> kyle = 96.6
```

To find the winner, we can use some `if` statements like this:

```
>>> if judy < kyle:
...     print "Judy wins with a time of", judy
... elif judy > kyle:
...     print "Kyle wins with a time of", kyle
```

```
... else:
...     print "Judy and Kyle are tied with a time of", judy
...
Judy wins with a time of 87.3
>>>
```

If Judy and Kyle are the only two snails, this program will work fine. Malek puts this all into a script. After each race, he changes the first two lines that give the finish times, and then runs the script.

This will work, but there are a number of objections:

- The person who prepares the race results has to know Python so they can edit the script.
- It doesn't really save any time. Any second-grader can look at the times and figure out who won.
- The names of the snails are part of the program, so if different snails are used, we have to write a new program.
- What if there are *three* snails? There are a lot more cases: three cases where a snail clearly wins; three more possible two-way ties; and a three-way tie. What if Malek wants to race ten snails at once? Too complicated!

11.3. Snail-oriented data structures: Lists

Let's consider the general problem of a race involving any number of snails. Malek is considering diversifying into amoeba racing, so there might be thousands of competitors in a race. So let's not limit the number of competitors in the program.

Also, to make it possible to use cheaper labor for production runs, let's write a general-purpose script that will read a file with the results for each race, so a relatively less skilled person can prepare that file, and then run a script that will review the results.

We'll use a very simple text file format to encode the race results. Here's an example file for that first race between Judy and Kyle:

```
87.3 Judy
96.6 Kyle
```

And here is a script that will process that file and report on the winning time. The script is called `snailist.py`. First, reads a race results file named `results` and stores the times into a list. The `.split()` method is used to break each line into parts, with the first part containing the elapsed time.

```
#!/usr/local/bin/python
#=================================================================
# snailist.py:  First snail racing results script.
#-----------------------------------------------------------------

#--
# Create an empty list to hold the finish times.
#--
timeList = []

#--
# Open the file containing the results.
#--
resultsFile = open ( 'results' )

#--
```

```
# Go through the lines of that file, storing each finish time.
#--
for  resultsLine in resultsFile:
    #--
    # Create a list of the fields in the line, e.g., ['87.3', 'Judy\n'].
    #--
    fieldList  =  resultsLine.split()

    #--
    # Convert the finish time into a float and append it to timeList.
    #--
    timeList.append ( float ( fieldList[0] ) )
```

At this point, `timeList` is a list of `float` values. We use the `.sort()` method to sort the list into ascending order, so that the winning time will be in the first element.

```
#--
# Sort timeList into ascending order, then set 'winningTime' to
# the best time.
#--
timeList.sort()
print "The winning time is", timeList[0]
```

Try building the `results` file and the script yourself to verify that they work. Try some cases where there are ties.

This script is fine as far as it goes. However, there is one major drawback: it doesn't tell you who won!

11.4. Snail-oriented data structures: A list of tuples

To improve on the script above, let's modify the script so that it keeps each snail's time and name together in a two-element tuple such as `(87.3, 'Judy')`.

In the improved script, the `timeList` list is a list of these tuples, and not just a list of times. We can then sort this list, using an interesting property of tuples. If you compare two tuples, and their first elements are not equal, the result is the same as if you compared their first elements. However, if the first elements are equal, Python then compares the second elements of each tuple, and so on until it either finds two unequal values, or finds that all the elements are equal.

Here's an example. Recall that the function `cmp(a, b)`, function compares two arbitrary values and returns a negative number if *a* comes before *b*, or a positive number if *a* comes after *b*, or zero if they are considered equal:

```
>>> cmp(50,30)
1
>>> cmp(30,50)
-1
>>> cmp(50,50)
0
>>>
```

If you compare two tuples and the first elements are unequal, the result is the same as if you compared the first two elements. For example:

```
>>> cmp ( (50,30,30), (80,10,10) )
-1
>>>
```

If, however, the first elements are equal, Python then compares the second elements, or the third elements, until it either finds two unequal elements, or finds that all the elements are equal:

```
>>> cmp ( (50,30,30), (80,10,10) )
-1
>>> cmp ( (50,30,30), (50,10,10) )
1
>>> cmp ( (50,30,30), (50,30,80) )
-1
>>> cmp ( (50,30,30), (50,30,30) )
0
>>>
```

So, watch what happens when we sort a list of two-tuples containing snail times and names:

```
>>> timeList = [ (87.3, 'Judy'), (96.6, 'Kyle'), (63.0, 'Lois') ]
>>> timeList.sort()
>>> timeList
[(63.0, 'Lois'), (87.299999999999997, 'Judy'), (96.599999999999994, 'Kyle')]
>>>
```

Now we have a list that is ordered the way the snails finished. Here is our modified script:

```
#!/usr/local/bin/python
#=====================================================================
# snailtuples.py:  Second snail racing results script.
#---------------------------------------------------------------------

#--
# Create an empty list to hold the result tuples.
#--
timeList = []

#--
# Open the file containing the results.
#--
resultsFile = open ( 'results' )

#--
# Go through the lines of that file, storing each finish time.
# Note that 'resultsLine' is set to each line of the file in
# turn, including the terminating newline ('\n').
#--
for  resultsLine in resultsFile:
    #--
    # Create a list of the fields in the line, e.g., ['87.3', 'Judy\n'].
    # We use the second argument to .split() to limit the number
    # of fields to two maximum; the first argument (None) means
    # split the line wherever there is any whitespace.
    #--
```

```
    fieldList  =  resultsLine.split(None, 1)

    #--
    # Now create a tuple (time,name) and append it to fieldList.
    # Use .rstrip to remove the newline from the second field.
    #--
    snailTuple = (float(fieldList[0]), fieldList[1].rstrip())
    timeList.append ( snailTuple )

#--
# Sort timeList into ascending order.
#--
timeList.sort()

#--
# Print the results.
#--
print "Finish  Time  Name"
print "------ ------ ----"
for  position in range(len(timeList)):
    snailTuple  =  timeList[position]
    print "{0:4d}   {1:6.1f} {2}".format(position+1, snailTuple[0],
                                        snailTuple[1])
```

Here is a sample run with our original two-snail `results` file:

```
Finish  Time  Name
------ ------ ----
   1     87.3 Judy
   2     96.6 Kyle
```

Let's try a larger `results` file with some names that have spaces in them, just to exercise the script. Here's the input file:

```
93.3 Queen Elizabeth I
138.4 Erasmus
88.2 Jim Ryun
```

And the output for this run:

```
Finish  Time  Name
------ ------ ----
   1     88.2 Jim Ryun
   2     93.3 Queen Elizabeth I
   3    138.4 Erasmus
```

11.5. Abstract data types

The preceding section shows how you can use a Python tuple to combine two simple values into a compound value. In this case, we use a 2-element tuple whose first element is the snail's time and the second element is its name.

We might say that this tuple is an *abstract data type*, that is, a way of combining Python's basic types (such as floats and strings) into new combinations.

The next step is to combine values *and functions* into an abstract data type. Historically, this is how object-oriented programming arose. The "objects" are packages containing simpler values inside them. However, in general, these packages can also contain *functions*.

Before we start looking at how we build abstract data types in Python, let's define some import terms and look at some real-world examples.

`class`

When we try to represent in our program some items out in the real world, we first look to see which items are similar, and group them into *classes*. A class is defined by one or more things that share the same qualities.

For example, we could define the class of *fountains* by saying that they are all permanent man-made structures, that they hold water, that they are outdoors in a public place, and that they keep the water in a decorative way.

It should be easy to determine whether any item is a member of the class or not, by applying these defining rules. For example, Trevi Fountain in Rome fits all the rules: it is man-made, holds water, is outdoors, and is decorative. Lake Geneva has water spraying out of it, but it's not man-made, so it's not a fountain.

`instance`

One of the members of a class. For example, the class of *airplanes* includes the Wright Biplane of 1903, and the Spirit of St. Louis that Charles Lindbergh flew across the Atlantic.

An instance is always a single item. "Boeing 747" is not an instance, it is a class. However, a specific Boeing 747, with a unique tail number like N1701, is an instance.

`attribute`

Since the purpose of most computer applications is in record-keeping, within a program, we must often track specific qualities of an instance, which we call *attributes*.

For example, attributes of an airplane include its wingspan, its manufacturer, and its current location, direction, and airspeed.

We can classify attributes into *static* and *dynamic* attributes, depending on whether they change or not. For example, the wingspan and model number of an airplane do not change, but its location and velocity can.

`operations`

Each class has characteristic operations that can be performed on instances of the class. For example, operations on airplanes include: manufacture; paint; take off; change course; land.

Here is a chart showing some classes, instances, attributes, and operations.

Class	**Instance**	**Attribute**	**Operation**
Airplane	Wright Flyer	Wingspan	Take off
Mountain	Socorro Peak	Altitude	Erupt
Clock	Skeen Library Clock	Amount slow per day	Decorate

Important

You have now seen definitions for most of the important terms in object-oriented programming. Python classes and instances are very similar to these real-world classes and instances. Python instances have attributes too.

For historical reasons, the term *method* is the object-oriented programming equivalent of "operation."

The term *constructor method* is the Python name for the operation that creates a new instance.

So what is an *object?* This term is used in two different ways:

- An *object* is just an instance.
- *Object-oriented programming* means programming with classes.

11.6. Abstract data types in Python

We saw how you can use a two-element tuple to group a snail's time and name together. However, in the real world, we might need to track more than two attributes of an instance.

Suppose Malek wants to keep track of more attributes of a snail, such as its age in days, its weight in grams, its length in millimeters, and its color. We could use a six-element tuple like this:

```
(87.3, 'Judy', 34, 1.66, 39, 'tan')
```

The problem with this approach is that we have to remember that for a tuple `T`, the time is in `T[0]`, the name in `T[1]`, the age in `T[2]`, and so on.

A cleaner, more natural way to keep track of attributes is to give them names. We might encode those six attributes in a Python dictionary like this:

```
T = { 'time':87.3, 'name':'Judy', 'age':34, 'mass':1.66,
      'length':39, 'color':'tan'}
```

With this approach, we can retrieve the name as `T['name]` or the weight as `T['mass']`. However, now we have lost the ability to put several of these dictionaries into a list and sort the list—how is Python supposed to know which dictionary comes first? What we need is something like a dictionary, but with more features. What we need is Python's object-oriented features.

Now we're to look at actual Python classes and instances in action.

11.7. `class SnailRun`: A very small example class

Let's start building a snail-racing application for Malek the object-oriented Python way. Let's assume that all we're tracking about a particular snail is its name and its finishing time. We need to define a class named `SnailRun`, whose instances track just these two attributes.

Here is the general form of a class declaration in Python:

```
class ClassName:
    def method_1(self, ...):
        block_1
    def method_2(self, ...):
        block_2
    ... etc.
```

A class declaration starts out with the keyword `class`, followed by the name of the class you are defining, then a colon (`:`). The methods of the class follow; each method starts with "`def`", just as you use to define a function.

Before we look at the construction of the class, let's see how it works in practice. To create an instance in Python, you use the name of the class as if it were a function call, followed by a list of arguments in

parentheses. Our `SnailRun` constructor method will need two arguments: the snail's name and its finish time. Once we have defined the class, we can build a new instance like this:

```
judyRace9 = SnailRun ( 'judy', 87.3 )
```

To get the snail's name and time attributes from an instance, we use the instance name, followed by a dot (`.`), followed by the attribute name:

```
>>> judyRace9.name
'judy'
>>> print judyRace9.time
87.3
```

Our example class, `SnailRun`, will have just two methods:

- All classes have a *constructor* method named "`__init__`". This method is used to create a new instance.
- We'll write a `.show()` method to format the contents of the instance for display.

Continuing our example from above, here's an example of the use of the `.show()` method:

```
>>> print judyRace9.show()
Snail 'judy' finished in 87.3 minutes.
```

Here is the entire class definition:

```
class SnailRun:
    def __init__ ( self, snailName, finishTime ):
        self.name  =  snailName
        self.time  =  finishTime

    def show ( self ):
        return ( "Snail '{0}' finished in {1:.1d} minutes.".format(
                 self.name, self.time )
```

Instantiation means the construction of a new instance. Here is how instantiation works.

1. Somewhere in a Python program, the programmer starts the construction of a new instance by using the class's name followed by parentheses and a list of arguments. Let's call the arguments `(`a_1`, `a_2`, ...)`.
2. Python creates a new namespace that will hold the instance's attributes. Inside the constructor, this namespace is referred to as `self`.

Important

The instance *is* basically a namespace, that is, a container for attribute names and their definitions. For other examples of Python namespaces, see Section 2.2, "The assignment statement" (p. 5), Section 5.3, "A namespace is like a dictionary" (p. 33), and Section 9.3, "A module is a namespace" (p. 51).

3. The `__init__()` (constructor) method of the class is executed with the argument list `(self, `a_1`, `a_2`, ...)`.

   Note that if the constructor takes N arguments, the caller passes only the last $N-1$ arguments to it.
4. When the constructor method finishes, the instance is returned to the caller. From then one, the caller can refer to some attribute A of the instance I as "$A.I$".

Let's look again in more detail at the constructor:

```
    def __init__ ( self, snailName, finishTime ):
        self.name  =  snailName
        self.time  =  finishTime
```

All the constructor does is to take the snail's name and finish time and store these values in the instance's namespace under the names `.name` and `.time`, respectively.

Note that the constructor method does not (and cannot) include a `return` statement. The value of `self` is implicitly returned to the statement that called the constructor.

As for the other methods of a class, their definitions also start with the special argument `self` that contains the instance namespace. For any method that takes *`N`* arguments, the caller passes only the last *`N-1`* arguments to it.

In our example class, the `def` for the `.show()` method has one argument named `self`, but the caller invokes it with no arguments at all:

```
>>> kyleRace3=SnailRun('Kyle', 96.6)
>>> kyleRace3.show()
"Snail 'Kyle' finished in 96.6 minutes."
```

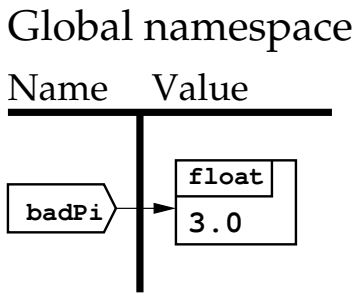
11.8. Life cycle of an instance

To really understand what is going on inside a running Python program, let's follow the creation of an instance of the `SnailRun` class from the preceding section.

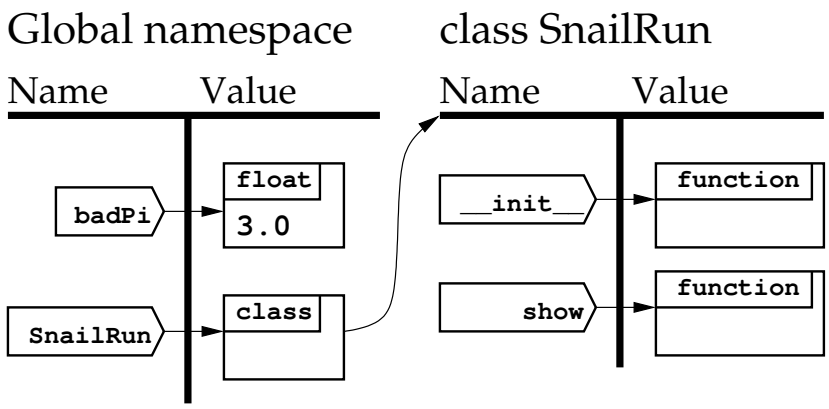
Just for review, let's assume you are using conversational mode, and you create a variable like this:

```
>>> badPi = 3.00
```

Whenever you start up Python, it creates the "global namespace" to hold the names and values you define. After the statement above, here's how it looks.



Next, suppose you type in the class definition as above. As it happens, a class is a namespace too—it is a container for methods. So the global namespace now has two names in it: the variable `badPi` and the class `SnailRun`. Here is a picture of the world after you define the class:



Next, create an instance of class `SnailRun` like this:

```
>>> j1 = SnailRun ( 'Judy', 87.3 )
```

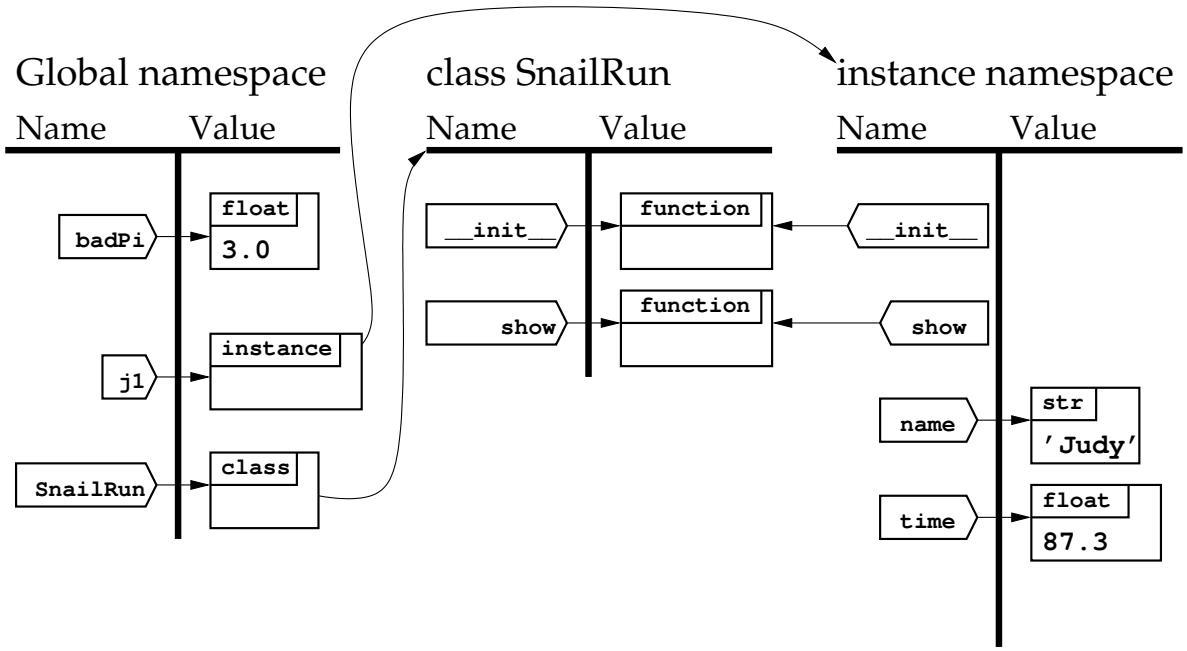
Here is the sequence of operations:

1. Python creates a new instance namespace. This namespace is initially a *copy of the class's namespace*: it contains the two methods `.__init__()` and `.show()`.
2. The constructor method starts execution with these arguments:
   - The name `self` is bound to the instance namespace.
   - The name `snailName` is bound to the string value `'Judy'`.
   - The name `finishTime` is bound to the float value 87.3.
3. This statement in the constructor

   ```
          self.name  =  snailName
   ```

   creates a new attribute `.name` in the instance namespace, and assigns it the value `'Judy'`.
4. The next statement in the constructor creates an attribute named `.time` in the instance namespace, and binds it to the value 87.3.
5. The constructor completes, and back in conversational mode, in the global namespace, variable `j1` is bound to the instance namespace.

Here's a picture of the world after all this:



11.9. Special methods: Sorting snail race data

Certain method names have special meaning to Python; each of these *special method names* starts with two underbars, "`__`".

A class's constructor method, `__init__()`, is an example of a special method. Whenever you use the class's name as if it were a function, in an expression like "`SnailRun('Judy', 67.3)`", Python executes the constructor method to build the new instance.

There is a full list of all the Python special method names in the *Python quick reference*[10]. Next we will look at another special method, `__cmp__`, that Python calls whenever you compare two instances of that class.

Going back to our snail-racing application, an instance of the `SnailRun` class contains everything we need to know about one snail's performance: its name in the `.name` attribute and its finish time in the `.time` attribute.

However, using the tuple representation back in Section 11.4, "Snail-oriented data structures: A list of tuples" (p. 58), we were able to put a collection of these tuples into a list, and sort the list so that they were ordered by finish time, with the winner first. Let's see what we need to add to `class SnailRun` so that we can sort a list of them into finish order by calling the `.sort()` method on the list.

First, a bit of review. Back in Section 6.1, "Conditions and the `bool` type" (p. 34), we learned about the built-in Python function `cmp(x, y)`, which returns:

- a negative number if *x* is less than *y*;
- a positive number if *x* is greater than *y*; or
- zero if *x* equals *y*.

In a Python class, if you define a method named "`__cmp__`", that method is called whenever Python compares two instances of the class. It must return a result using the same conventions as the built-in `cmp()` function: negative for "<", zero for "==", positive for ">".

[10] http://www.nmt.edu/tcc/help/pubs/python/web/special-methods.html

In the case of "`class SnailRun`", we want the snail with the better finishing time to be considered less than the slower snail. So here is one way to define the `__cmp__` method for our class:

```
    def __cmp__ ( self, other ):
        """Define how to compare two SnailRun instances.
        """
        if self.time < other.time:
            return -1
        elif self.time > other.time:
            return 1
        else:
            return 0
```

When this method is called, `self` is an instance of class `SnailRun`, and `other` should also be an instance of `SnailRun`.

However, this logic exactly duplicates what the built-in `cmp()` function does to compare two `float` values, so we can simplify it like this:

```
    def __cmp__ ( self, other ):
        """Define how to compare two SnailRun instances.
        """
        return cmp(self.time, other.time)
```

Let's look at another special method, `__str__()`. This one defines how Python converts an instance of a class into a string. It is called, for example, when you name an instance in a `print` statement, or when you pass an instance to Python's built-in `str()` function.

The `__str__()` method of a class returns a string value. It is up to the writer of the class what string value gets returned. As usual for Python methods, the `self` argument contains the instance. In the case of `class SnailRun`, we'll want to display the snail's name (`.name` attribute) and finishing time (`.time` attribute). Here's one possible version:

```
    def __str__ ( self ):
        """Return a string representation of self.
        """
        return "{0:8.1f} {1}".format(self.time, self.name)
```

This method will format the finishing time into an 8-character string, with one digit after the decimal point, followed by one space, then the snail's name.

Let's assume that the `__cmp__()` and `__str__()` methods have been added to our `snails.py` module, and show their use in some conversational examples.

```
>>> from snails import *
>>> sally4 = SnailRun('Sally', 88.8)
>>> jim4=SnailRun('Jim', 76.5)
>>>
```

Now that we have two `SnailRun` instances, we can show how the `__str__()` method formats them for printing:

```
>>> print sally4
    88.8 Sally
>>> print jim4
    76.5 Jim
>>>
```

We can also show the various ways that Python compares two instances using our new `__cmp__()` method.

```
>>> cmp(sally4,jim4)
1
>>> sally4 > jim4
True
>>> sally4 <= jim4
False
>>> sally4 < jim4
False
>>>
```

Now that we have defined how instances are to be ordered, we can sort a list of them in order by finish time. First we throw them into the list in any old order:

```
>>> judy4 = SnailRun ( 'Judy', 67.3 )
>>> blake4 = SnailRun ( 'Blake', 181.4 )
>>> race4 = [sally4, jim4, judy4, blake4]
>>> for run in race4:
...     print run
...
    88.8 Sally
    76.5 Jim
    67.3 Judy
   181.4 Blake
>>>
```

The `.sort()` method on a list uses Python's `cmp()` function to compare items when sorting them, and this in turn will call our class's `__cmp__()` method to sort them by finishing time.

```
>>> race4.sort()
>>> for run in race4:
...     print run
...
    67.3 Judy
    76.5 Jim
    88.8 Sally
   181.4 Blake
>>>
```

For an extended example of a class that implements a number of special methods, see *`rational.py`: An example Python class*[11]. This example shows how to define a new kind of numbers, and specify how operators such as "`+`" and "`*`" operate on instances.

[11] http://www.nmt.edu/tcc/help/pubs/lang/python/examples/rational/