Python 3.2 quick reference

John W. Shipman

2012-07-05 14:40

Abstract

A reference guide to most of the common features of the Python programming language, version 3.2.

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/python32/
[2] http://www.nmt.edu/tcc/help/pubs/lang/python32/python32.pdf

1. Python 3.2: A fine general-purpose programming language

The Python programming language is a recent, general-purpose, higher-level programming language. It is available for free and runs on pretty much every current platform.

This document is a reference guide, not a tutorial. If you are new to Python programming, see the tutorial written by Guido van Rossum[3], the inventor of Python.

Not every feature of Python is covered here. If you are interested in exploring some of the more remote corners of the language, refer to the official standard library[4] and language reference[5] documents. Bookmark the standard library right away if you haven't already; you will find it most useful. The language reference is formal and technical and will be of interest mainly to specialists.

1.1. Python 2.x and 3.x

Since the language's inception in the early 1990s, the maintainers have been careful to add new features in a way that minimized the number of cases where old code would not function under the new release.

The 3.0 version was the first to violate this rule. A number of language features that were compatible through the 2.7 release have been taken out in the 3.x versions. However, this is unlikely to happen again; it is mainly a one-time cleanup and simplification of the language.

Since 3.0, the Python language is actually smaller and more elegant. Furthermore, the upgrade path from 2.7 (the last major 2.x release) to the 3.x versions is straightforward and to a large extent automated.

Important

There is no hurry to convert old 2.x programs to 3.x! The 2.7 release will certainly be maintained for many years. Your decision about when to convert may depend on the porting of any third-party library modules you use.

In any case, the conversion process is discussed in Section 26, “The conversion path from 2.x to 3.x” (p. 50).

2. Starting Python

You can use Python in two different ways:

- In “calculator” or “conversational mode”, Python will prompt you for input with three greater-than signs (**>>>**). Type a line and Python will print the result. Here's an example:

```
>>> 2+2
4
>>> 1.0 / 7.0
0.14285714285714285
```

- You can also use Python to write a program, sometimes called a *script*.

How you start Python depends on your platform.

[3] http://docs.python.org/py3k/tutorial/
[4] http://docs.python.org/py3k/library/
[5] http://docs.python.org/py3k/reference/

- To install Python on your own workstation, refer to the Python downloads page[6].
- On a Tech Computer Center Windows workstation, see Section 2.1, "Using Python in Windows" (p. 4).
- On a TCC Linux workstation, see Section 2.2, "Using Python in Linux" (p. 4).

2.1. Using Python in Windows

Under Windows, we recommend the IDLE integrated development environment for Python work.

If you are using Python at the NM Tech Computer Center (TCC), you can get conversational mode from *Start* → *All Programs* → *Python 3.2* → *IDLE (Python GUI)*.

You will see the usual "`>>>`" interactive prompt. You can use conversational mode in this window.

To construct a Python script, use *File* → *New Window*. Write your script in this window, then save it with *File* → *Save As...*; make sure your file name ends with `.py`. Then use *Run* → *Run Module* (or *F5*) to run your script. The output will appear in the conversational-mode window.

You may also run a Python script by double-clicking on it, provided that its name ends with "`.py`".

2.2. Using Python in Linux

To enter conversational mode on a Linux system, type this command:

```
python3
```

Type *Control-D* to terminate the session.

If you write a Python script named *`filename`*`.py`, you can execute it using the command

```
python3 filename.py arguments...
```

Under Unix, you can also make a script self-executing by placing this line at the top:

```
#!/usr/bin/env python3
```

You must also tell Linux that the file is executable by using the command "`chmod +x` *`filename`*". For example, if your script is called `hello.py`, you would type this command:

```
chmod +x hello.py
```

3. Line syntax

The comment character is "`#`"; comments are terminated by end of line.

Long lines may be continued by ending the line with a backslash (\), but this is not necessary if there is at least one open "`(`", "`[`", or "`{`".

4. Names and keywords

Python names (also called identifiers) can be any length and follow these rules:

- The first or only character must be a letter (uppercase or lowercase) or the underbar character, "`_`".
- Any additional characters may be letters, underbars, or digits.

[6] http://python.org/download/

Examples: `coconuts`, `sirRobin`, `blanche_hickey_869`, `__secretWord`.

Case is significant in Python. The name `Robin` is not the same name as `robin`.

You can use non-ASCII Unicode characters as Python names. For details, see the the reference documentation[7].

The names below are *keywords*, also known as reserved words. They have special meaning in Python and cannot be used as names or identifiers.

```
False  assert    del      for     in        or      while
None   break     elif     from    is        pass    with
True   class     else     global  lambda    raise   yield
and    continue  except   if      nonlocal  return
as     def       finally  import  not       try
```

Certain names starting with the underbar character ("_") are reserved.

- The name "_" is available during conversational mode to refer to the result of the previous computation. In a script, it has no special meaning and is not defined.
- Many names starting and ending with two underbars ("__...__") have special meaning in Python. It is best not to use such names for your own code.
- Names starting with one underbar are not imported by the `import` statement.
- Names starting with two underbars are private to the class in which they are declared. This is useful for distinguishing names in a class from names in its parent classes.

  Such names are not actually hidden; they are mangled in a prescribed way. The mangling prevents conflicts with names in parent classes, while still making them visible to introspective applications such as the Python debugger.

5. Python types

All the data you manipulate in Python consists of *objects* and *names*.

- An object represents some value. It may be a simple value like the number 17, or a very complex value, like an object that describes the results of every cricket game ever played between England and Australia.

  Every object has a *type*. Python has a rich set of built-in types. You can also invent your own types using Python's object-oriented programming features.

  The type of an object determines what operations can be performed on it. For example, an object of Python's `int` (integer) type holds a whole number, and you can use operators like "`+`" and "`-`" to add and subtract objects of that type.
- A Python name is like a luggage tag: it is a handle you use to manipulate the values contained in an object.

  For the rules Python uses for names, see Section 4, "Names and keywords" (p. 4).

[7] http://docs.python.org/py3k/reference/lexical_analysis.html#identifiers

Important

Unlike many current languages such as C and C++, a Python name is not associated with a type. A name is associated with an object, and that object has a type.

The association between a name and an object is called a *binding*. For example, after executing this statement,

```
x = 17
```

we say that the name `x` is *bound* to the object 17, which has type `int`.

There is no reason that name `x` can't be associated with an integer at one point and a completely different type later in the execution of a script. (It is, however, bad practice. Programs are more clear when each name is used for only one purpose.)

Here is a list of Python's built-in primitive types. To learn how to invent your own types, see Section 25, “Classes: invent your own types” (p. 50). For a discussion of the distinction between mutable and immutable types, see Section 8.2, “Mutability and immutability” (p. 12).

Table 1. Python's built-in types

Type name	Values	Examples
`bool`	The two Boolean values `True` and `False`. See Section 6, "The `bool` type: Truth values" (p. 8).	`True`, `False`
Section 7, "Numeric types" (p. 8)		
`int`	Integers of any size, limited only by the available memory. See Section 7.1, "Type `int`: Whole numbers" (p. 8).	`42`, `-3`, `123456789012345678901234567890`
`float`	Floating-point numbers; see Section 7.2, "Type `float`: Approximated real numbers" (p. 9).	`3.14159`, `-1.0`, `6.0235e23`
`complex`	Complex numbers. If the idea of computing with the square root of -1 bothers you, just ignore this type, otherwise see Section 7.3, "Type `complex`: Imaginary numbers" (p. 10).	`(3.2+4.9j)`, `(0+3.42e-3j)`
Section 8, "Sequence types (ordered sets)" (p. 10)		
`str`	Strings (sequences of zero or more Unicode characters); see Section 9, "Type `str`: Strings of text characters" (p. 15). Strings can be empty: write these as `""` or `''`.	`'Sir Robin'`, `"xyz"`, `"I'd've"`, `"\u262e\u262f"`
`bytes`	Immutable sequences of zero or more positive integers in the range [0, 255]. See Section 10, "Type `bytes`: Immutable sequences of 8-bit integers" (p. 30).	`b'git'`, `bytes([14, 202, 6])`
`bytearray`	Mutable sequences of zero or more positive integers in the range [0, 255]. See Section 11, "Type `bytearray`: Mutable sequences of 8-bit integers" (p. 30).	`bytearray()`, `bytearray(b'Bletchley')`
`list`	A mutable sequence of values; see Section 12, "Type `list`: Mutable sequences of arbitrary objects" (p. 30).	`['dot', 'dash']`; `[]`
`tuple`	An immutable sequence of values; see Section 13, "Type `tuple`: Immutable sequences of arbitrary objects" (p. 30).	`('dot', 'dash')`; `()`; `("singleton",)`
Section 15, "The set types: `set` and `frozenset`" (p. 30)		
`set`	An unordered, immutable set of zero or more distinct values.	`set()`, `set('red', 'yellow', 'green')`
`frozenset`	An unordered, mutable set of zero or more distinct values.	`frozenset([3, 5, 7])`, `frozenset()`
`dict`	Use `dict` values (dictionaries) to structure data as look-up tables; see Section 16, "Type `dict`: Mappings" (p. 30).	`{'go':1, 'stop':0}`, `{}`
`None`	A special, unique value that may be used where a value is required but there is no obvious value. See Section 17, "Type `None`: The special placeholder value" (p. 30).	`None`

6. The `bool` type: Truth values

A value of `bool` type represents a Boolean (true or false) value. There are only two values, written in Python as "`True`" and "`False`".

Internally, `True` is represented as 1 and `False` as 0, and they can be used in numeric expressions as those values.

Here's an example. In Python, the expression "*a* `<` *b*" compares two values *a* and *b*, and returns `True` if *a* is less than *b*, `False` is *a* is greater than or equal to *b*.

```
>>> 2 < 3
True
>>> 3 < 2
False
>>> True+4
5
>>> False * False
0
```

These values are considered `False` wherever true/false values are expected, such as in an `if` statement:

- The `bool` value `False`.
- Any numeric zero: the `int` value `0`, the `float` value `0.0`, the `long` value `0L`, or the `complex` value `0.0j`.
- Any empty sequence: the `str` value `''`, the `unicode` value `u''`, the empty `list` value `[]`, or the empty `tuple` value `()`.
- Any empty mapping, such as the empty `dict` (dictionary) value `{}`.
- The special value `None`.

All other values are considered `True`. To convert any value to a Boolean, see Section 19.3, "`bool()`: Convert to `bool` type" (p. 34).

7. Numeric types

Python has three built-in types for representing numbers.

7.1. Type `int`: Whole numbers

A Python object of type `int` represents an integer, that is, a signed whole number. The range of `int` values is limited only by the available memory.

To write an `int` constant, you may use several different formats.

- A zero is written as just `0`.
- To write an integer in decimal (base 10), the first digit must not be zero. Examples: `17`, `100000000000000`.
- To write an integer in octal (base 8), precede it with "`0o`". Examples: `0o177`, `0o37`.

Note

In Python 2.x versions, any number starting with a `0` followed by other digits was considered octal. This convention is no longer allowed in Python 3.x.

```
>>> 0o37
31
>>> 0o177
127
>>> 00004
  File "<stdin>", line 1
    00004
        ^
SyntaxError: invalid token
```

- To write an integer in hexadecimal (base 16), precede it with "`0x`" or "`0X`". Examples: `0x7f`, `0X1000`.
- To write an integer in binary (base 12), precede it with "`0b`" or "`0B`". Examples: `0b1001`, `0B111110011101`.

To produce a negative number, use the unary "`-`" operator before the number. Note that this is an operator and not part of the constant.

```
>>> 100 - -5
105
>>> 100 - (-5)
105
```

To convert a string value to an `int`, see Section 19.6, "`int()`: Convert to `int` type" (p. 34).

7.2. Type `float`: Approximated real numbers

Values of this type represent real numbers, with the usual limitations of IEEE-754 floating point type: it cannot represent very large or very small numbers, and the precision is limited to only about 15 digits. For complete details on the IEEE-754 standard and its limitations, see the Wikipedia article[8].

To write a `float` constant, use at least one digit, plus either a decimal point or an exponent or both.

- The decimal point, if present, must be preceded or followed by one or more digits. Examples:

  ```
  3.14
  0.0
  1.
  .1
  ```

  Just for the sake of legibility, we recommend that you use at least one digit on *both* sides of the decimal point. Depending on the context, a reader might wonder whether a number such as "`.1`" is one-tenth, or the number one preceded by an errant particle of pepper on the paper or monitor.
- The exponent, if present, consists of either "`e`" or "`E`", optionally followed by a "`+`" or "`-`" sign, followed by at least one digit. The resulting value is equal to the part before the exponent, times ten to the power of the exponent, that is, scientific notation[9].

[8] http://en.wikipedia.org/wiki/IEEE_754-2008
[9] http://en.wikipedia.org/wiki/Scientific_notation

For example, Avogadro's Number gives the number of atoms of carbon in 12 grams of carbon[12], and is written as 6.0221418×10^{23}. In Python that would be "`6.0221418e23`".

7.3. Type `complex`: Imaginary numbers

Mathematically, a complex number is a number of the form *`A+Bi`* where *`i`* is the imaginary number, equal to the square root of -1.

Complex numbers are quite commonly used in electrical engineering. In that field, however, because the symbol *`i`* is used to represent current, they use the symbol *`j`* for the square root of -1. Python adheres to this convention: a number followed by "`j`" is treated as an imaginary number. Python displays complex numbers in parentheses when they have a nonzero real part.

```
>>> 5j
5j
>>> 1+2.56j
(1+2.56j)
>>> (1+2.56j)*(-1-3.44j)
(7.8064-6j)
```

Unlike Python's other numeric types, complex numbers are a composite quantity made of two parts: the real part and the imaginary part, both of which are represented internally as `float` values. You can retrieve the two components using attribute references. For a complex number *`C`*:

- *`C`*`.real` is the real part.
- *`C`*`.imag` is the imaginary part as a `float`, not as a `complex` value.

```
>>> a=(1+2.56j)*(-1-3.44j)
>>> a
(7.8064-6j)
>>> a.real
7.8064
>>> a.imag
-6.0
```

To construct a `complex` value from two `float` values, see Section 19.4, "`complex()`: Convert to `complex` type" (p. 34).

8. Sequence types (ordered sets)

Python's sequence types – `str`, `bytes`, `bytearray`, `list`, `tuple` and `range` – share a number of common operations (see Section 8.3, "Common operations on sequence types" (p. 13)).

Objects of each of these types share a common algebraic abstraction: the *ordered set*. Among the properties of an ordered set are:

- An ordered set may be empty, or it may contain one or more objects.
- The contained objects don't all have to be different. For example, the list `[1, 2, 2, 2]` contains three copies of the `int` value `2`.
- If an ordered set has more than object, they can be numbered: there is a first one, a second one, and so on.

Here are the sections describing Python's sequence types.

- Section 8.1, "It's a Unicode world now" (p. 11).

- Section 9, “Type `str`: Strings of text characters” (p. 15).
- Section 10, “Type `bytes`: Immutable sequences of 8-bit integers” (p. 30).
- Section 11, “Type `bytearray`: Mutable sequences of 8-bit integers” (p. 30).
- Section 12, “Type `list`: Mutable sequences of arbitrary objects” (p. 30).
- Section 13, “Type `tuple`: Immutable sequences of arbitrary objects” (p. 30).
- Section 14, “Type `range`: A range of values” (p. 30).
- Section 8.3, “Common operations on sequence types” (p. 13).

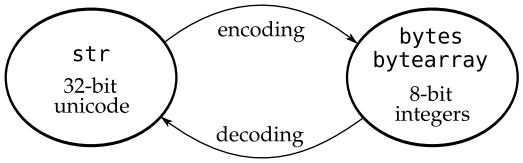
8.1. It's a Unicode world now

For the first several decades of software history, the processing of text in the USA almost everywhere used the 7-bit ASCII[10] code or various 8-bit codes. To this day, input and output to storage devices uses streams of 8-bit bytes.

However, for so many reasons, but especially for internationalization of interfaces, it is time to abandon this tiny character set and join the wider world. The 32-bit characters of the Unicode standard[11] provide enough characters to last us for many years into the future.

In all the Python 3.x versions, there is a strict separation of 32-bit character data and sequences of 8-bit bytes.

- All character data is represented as type `str`. An object of this type represents an ordered set of 32-bit Unicode characters.
- Ordered sets of 8-bit types may use either type `bytes` (for immutable values) or type `bytearray` (for mutable values; see Section 8.2, “Mutability and immutability” (p. 12)).
- To convert from one to the other, you must specify an *encoding*, that is, a system for converting between 8-bit and 32-bit representations. There are many encoding systems, but in most cases the UTF-8[12] encoding is a reasonable default, compatible with most current Web practice.



To encode data:

- The `str.encode()` method allows you to specify the encoding; see Section 9.1, “Methods on class `str`” (p. 16).
- The built-in `bytes()` and `bytearray()` functions allow you to specify an encoding; see Section 10, “Type `bytes`: Immutable sequences of 8-bit integers” (p. 30) and Section 11, “Type `bytearray`: Mutable sequences of 8-bit integers” (p. 30).
- When you open a file, you can specify an encoding, and this encoding will be used to translate any 32-bit data that gets written to that file. See Section 19.11, “`open()`: Open a file” (p. 35).

To decode data:

- The `str()` built-in function allows you to specify an encoding that will be used to decode 8-bit data as Unicode. See Section 9, “Type `str`: Strings of text characters” (p. 15).

[10] http://en.wikipedia.org/wiki/ASCII
[11] http://en.wikipedia.org/wiki/Unicode
[12] http://en.wikipedia.org/wiki/UTF-8

- The `bytes.decode()` and `bytearray.decode()` functions allow you to specify an encoding; see Section 10, "Type `bytes`: Immutable sequences of 8-bit integers" (p. 30) and Section 11, "Type `bytearray`: Mutable sequences of 8-bit integers" (p. 30).
- What happens when you read data from a file depends on how the file was opened. If it is character data, you may specify an encoding in the call to the `open()` function, or use the local site default encoding.

  If you are handling byte data, you may include `"b"` in the `mode` argument to the `open()` function, and you will get back data of type `bytes`.

8.2. Mutability and immutability

One important quality of a Python type is whether it is considered mutable or immutable. An object is considered mutable if *part* of its contained value can be changed.

For example, of Python's two universal container types, `list` objects are mutable, but `tuple` objects are not.

```
>>> some_list = [2, 3, 5, 7, 11]
>>> some_list[2] = 888
>>> some_list
[2, 3, 888, 7, 11]
>>> some_tuple = (2, 3, 5, 7, 11)
>>> some_tuple[2] = 888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

For another example, `str` values are immutable. You cannot change one character within a string of three characters:

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

Of Python's two types that represent strings of 8-bit bytes, objects of the `bytes` type are immutable, but `bytearray` objects are mutable.

```
>>> v1
b'aeiou'
>>> v2=bytearray(v1)
>>> print(v1, v2)
b'aeiou' bytearray(b'aeiou')
>>> v2.append(ord('y'))
>>> v2
bytearray(b'aeiouy')
>>> v1.append(ord('y'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'bytes' object has no attribute 'append'
```

8.3. Common operations on sequence types

A number of Python operators, functions, and methods operate on all of the sequence types: `str`, `bytes`, `bytearray`, `list`, `tuple`, and `range`.

Functions include:

- Section 19.7, "`iter()`: Return an iterator for a given sequence" (p. 35).
- Section 19.8, "`len()`: How many elements in a sequence?" (p. 35).
- Section 19.10, "`min()`: What is the smallest element of a sequence?" (p. 35).
- Section 19.9, "`max()`: What is the largest element of a sequence?" (p. 35).

Additionally, these methods are supported on any value *s* that is one of the sequence types.

***s*.count(*k*)**
Returns the number of elements of *s* that are equal to *k*.

```
>>> [0, 1, 0, 0, 0, 5].count(0)
[0, 1, 0, 0, 0, 5].count(0)
4
>>> b'abracadabra'.count(b'a')
b'abracadabra'.count(b'a')
5
```

***s*.index(*k*)**
Returns the position of the first element of *s* that is equal to *k*. If no element matches, this function raises a `ValueError` exception.

```
>>> "abcde".index('d')
3
>>> "abcde".index('x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
>>> range(18).index(14)
14
```

These operators apply to sequences.

S_1*+*S_2
Concatenation—for two sequences S_1 and S_2 of the same type, a new sequence containing all the elements from S_1 followed by all the elements of S_2.

```
>>> "vi" + "car"
'vicar'
>>> [1,2,3]+[5,7,11,13]+[15]
[1, 2, 3, 5, 7, 11, 13, 15]
>>> ('roy', 'g')+('biv',)
('roy', 'g', 'biv')
```

***S**n**
For a sequence *S* and a positive integer *n*, the result is a new sequence containing all the elements of *S* repeated *n* times.

```
>>> 'worra'*8
'worraworraworraworraworraworraworra'
```

```
>>> [0]*4
[0, 0, 0, 0]
>>> (True, False)*5
(True, False, True, False, True, False, True, False, True, False)
```

x* in *S

Is any element of a sequence *S* equal to *x*?

For convenience in searching for substrings, if the sequence to be searched is a string, the *x* operand can be a multi-character string. In that case, the operation returns `True` if *x* is found anywhere in *S*.

```
>>> 1 in [2,4,6,0,8,0]
False
>>> 0 in [2,4,6,0,8,0]
True
>>> 'a' in 'banana'
True
>>> 3.0 in (2.5, 3.0, 3.5)
True
>>> "baz" in "rowrbazzle"
True
```

x* not in *S

Are all the elements of a sequence *S* not equal to *x*?

```
>>> 'a' not in 'banana'
False
>>> 'x' not in 'banana'
True
```

***S*[*i*]**

Subscripting: retrieve the *i*th element of *s*, *counting from zero*. If *i* is greater than or equal to the number of elements of *S*, an `IndexError` exception is raised.

```
>>> 'Perth'[0]
'P'
>>> 'Perth'[1]
'e'
>>> 'Perth'[4]
'h'
>>> 'Perth'[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> ('red', 'yellow', 'green')[2]
'green'
```

***S*[*i*:*j*]**

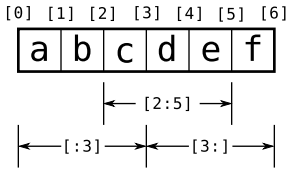
Slicing: For a sequence *S* and two integers *i* and *j*, return a new sequence with copies of the elements of *S* between *positions* *i* and *j*.

The values used in slicing refer to the positions *between* elements, where position zero is the position *before* the first element; position 1 is between the first and second element; and so on.

You can also specify positions relative to the end of a sequence. Position -1 is the position before the last element; -2 is the position before the second-to-last element; and so on.

You can omit the starting position to obtain a slice starting at the beginning. You can omit the ending position to get all the elements through the last.

For example, here is a diagram showing three slices of the string `'abcdef'`.




```
>>> 'abcdef'[2:5]
'cde'
>>> 'abcdef'[:3]
'abc'
>>> 'abcdef'[3:]
'def'
>>> (90, 91, 92, 93, 94, 95)[2:5]
(92, 93, 94)
```

***S*[*i*:*j*:*k*]**

You can use a slice expression like this to select every *k*th element. Examples:

```
>>> teens = range(13,20)
>>> teens
[13, 14, 15, 16, 17, 18, 19]
>>> teens[::2]
[13, 15, 17, 19]
>>> teens[1::2]
[14, 16, 18]
>>> teens[1:5]
[14, 15, 16, 17]
>>> teens[1:5:2]
[14, 16]
```

9. Type `str`: Strings of text characters

A value of Python's `str` type is a sequence of zero or more 32-bit Unicode characters (see Section 8.1, “It's a Unicode world now” (p. 11)).

In addition to the functions described in Section 8.3, “Common operations on sequence types” (p. 13), these built-in functions apply to strings:

- Section 19.5, “`input()`: Read a string from standard input” (p. 34).
- Section 19.12, “`ord()`: What is the code point of this character?” (p. 35).
- Section 19.14, “`str()`: Convert to `str` type” (p. 36).

9.1. Methods on class `str`

All string values *S* (instances of type `str`) support the methods enumerated below.

9.1.1. *S*`.capitalize()`

Return *S* with its first character capitalized (if a letter).

```
>>> 'e e cummings'.capitalize()
'E e cummings'
>>> '---abc---'.capitalize()
'---abc---'
```

9.1.2. *S*`.center(`*w*`[, `*fill*`])`

Return *S* centered in a string of width *w*, padded with spaces. If *w*`<=len(`*S*`)`, the result is a copy of *S*. If the number of spaces of padding is odd, the extra space will placed after the centered value. Example:

```
>>> 'x'.center(4)
' x  '
```

If you would like to use some specific character to pad the result, pass that character as the optional second argument:

```
>>> "x".center(11, '/')
'/////x/////'
```

9.1.3. *S*`.count(`*t*`[,`*start*`[,`*end*`]])`

Return the number of times string *t* occurs in *S*. To search only a slice *S*`[`*start*`:`*end*`]` of S, supply *start* and *end* arguments.

```
>>> 'banana'.count('a')
3
>>> 'bananana'.count('na')
3
>>> 'banana'.count('a', 3)
2
>>> 'banana'.count('a', 3, 5)
1
>>> 'banana'.count('ana')
1
```

Note in the last example above how this function counts only the number of *non-overlapping* occurrences of the string.

9.1.4. *S*`.encode([`*encoding*`[, `*errors*`]])`

Encodes a Unicode string and returns the encoded form as a value of type `bytes`. See Section 10, “Type `bytes`: Immutable sequences of 8-bit integers” (p. 30), and also be sure you familiarize yourself with Section 8.1, “It's a Unicode world now” (p. 11), especially if you are familiar with the very different character handling in Python 2.x versions.

The default *encoding* is "utf-8". Many encodings are supported; for a full list, see the standard library documentation[13] for a current list. Here are some popular values:

"utf-8"	The standard UTF-8 encoding. This is the default encoding.
"utf-16"	The standard UTF-16 encoding.
"utf-32"	The standard UTF-32 encoding.
"ascii"	ASCII[14], for the American Standard Code for Information Interchange.

In the examples below, '\xa0' is the Unicode character , non-breaking space, which has no ASCII equivalent.

```
>>> s='(\xa0)'
s='(\xa0)'
>>> s.encode()
s.encode()
b'(\xc2\xa0)'
>>> s.encode('utf-8')
s.encode('utf-8')
b'(\xc2\xa0)'
>>> s.encode('utf-16')
s.encode('utf-16')
b'\xff\xfe(\x00\xa0\x00)\x00'
>>> s.encode('utf-32')
s.encode('utf-32')
b'\xff\xfe\x00\x00(\x00\x00\x00\xa0\x00\x00\x00)\x00\x00\x00'
```

The optional *errors* second argument specifies what to do when a character cannot be encoded.

"strict"
Raise a UnicodeEncodeError exception if any of the characters can't be encoded using the specified encoding. This is the default value of the *errors* argument.

```
>>> s = "(\xa0)"
>>> s.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\xa0' in
position 1: ordinal not in range(128)
```

"ignore"
Omit from the result any characters that can't be encoded.

```
>>> s = "(\xa0)"
>>> s.encode('ascii', 'ignore')
b'()'
```

"replace"
In the result, display as "?" any character that can't be encoded.

```
>>> "(\xa0)".encode('ascii', 'replace')
b'(?)'
```

[13] http://docs.python.org/py3k/library/codecs.html#standard-encodings
[14] http://en.wikipedia.org/wiki/ASCII

"xmlcharrefreplace"
For each character that can't be encoded, substitute an XML character entity reference of the form "&#*N*;", where *N* is the decimal value of the Unicode code point of the character. This rendering is a good choice for displaying Unicode characters in HTML.

```
>>> "(\xa0)".encode('ascii', 'xmlcharrefreplace')
b'( )'
```

"backslashreplace"
In the result, render characters that cannot be encoded using Python's backslash escape convention: `'\xNN'` for 8-bit code points, `'\uNNNN'` for 16-bit code points, and `'UNNNNNNNN'` for 32-bit code points.

```
>>> zoot='\xa0\u1234\U00012345';print(len(zoot))
3
>>> zoot.encode('ascii', 'backslashreplace')
b'\\xa0\\u1234\\U00012345'
```

9.1.5. *S*.endswith(*t*[,*start*[,*end*]])

Predicate to test whether *S* ends with string *t*. If you supply the optional *start* and *end* arguments, it tests whether the slice *S*[*start*:*end*] ends with *t*.

```
>>> 'bishop'.endswith('shop')
True
>>> 'bishop'.endswith('bath and wells')
False
>>> 'bishop'[3:5]
'ho'
>>> 'bishop'.endswith('o', 3, 5)
True
```

9.1.6. *S*.expandtabs([*tabsize*])

Returns a copy of *S* with all tabs replaced by one or more spaces. Each tab is interpreted as a request to move to the next "tab stop". The optional *tabsize* argument specifies the number of spaces between tab stops; the default is 8.

Here is how the function actually works. The characters of *S* are copied to a new string *T* one at a time. If the character is a tab, it is replaced by enough tabs so the new length of *T* is a multiple of the tab size (but always at least one space).

```
>>> 'X\tY\tZ'.expandtabs()
'X       Y       Z'
>>> 'X\tY\tZ'.expandtabs(4)
'X   Y   Z'
>>> print('+...'*8, '\n', 'a\tbb\tccc\tdddd\teeeee\tf'.expandtabs(4),
...       sep='')
+...+...+...+...+...+...+...+...
a   bb  ccc dddd    eeeee   f
```

9.1.7. *S*.find(*t*[,*start*[,*end*]])

If string *t* is not found in *S*, return -1; otherwise return the index of the first position in *S* that matches *t*.

The optional *start* and *end* arguments restrict the search to slice *S*[*start*:*end*].

```
>>> 'banana'.find('an')
1
>>> 'banana'.find('ape')
-1
>>> 'banana'.find('n', 3)
4
>>> 'council'.find('c', 1, 4)
-1
```

If you are testing whether a certain substring is found within a larger string, but you don't care exactly where it starts, see the "`in`" and "`not in`" operators in Section 8.3, "Common operations on sequence types" (p. 13).

9.1.8. *S*.format(**p*, ***kw*)

See Section 9.2, "The string `.format()` method" (p. 21).

9.1.9. *S*.index(*t*[,*start*[,*end*]])

Works like `.find()`, but if *t* is not found, it raises a `ValueError` exception.

```
>>> 'council'.index('un')
2
>>> 'council'.index('phd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

9.1.10. *S*.isalnum()

This method is a predicate that tests whether *S* is nonempty and all its characters are alphanumeric.

```
>>> ''.isalnum()
False
>>> 'abc123'.isalnum()
True
>>> '&*$#&*()abc123'.isalnum()
False
```

9.1.11. *S*.isalpha()

Predicate that tests whether *S* is nonempty and all its characters are letters.

```
>>> 'abc123'.isalpha()
False
>>> 'MaryRecruiting'.isalpha()
```

```
True
>>> ''.isalpha()
False
```

9.1.12. *S*.isdecimal()

Similar to Section 9.1.13, "*S*.isdigit()" (p. 20), but considers characters in the Unicode category Nd to be digits as well as the usual 0 through 9.

9.1.13. *S*.isdigit()

Predicate that tests whether *S* is nonempty and all its characters are digits.

```
>>> 'abc123'.isdigit()
False
>>> ''.isdigit()
False
>>> '2415'.isdigit()
True
```

9.1.14. *S*.isidentifier()

A predicate that tests whether a *S* is a valid Python identifier.

```
>>> "lionTamer".isidentifier()
True
>>> "spiny_norman".isidentifier()
True
>>> "_42".isidentifier()
True
>>> "9_a".isidentifier()
False
```

9.1.15. *S*.islower()

Predicate that tests whether *S* is nonempty and all its letters are lowercase (non-letter characters are ignored).

```
>>> ''.islower()
False
>>> 'abc123'.islower()
True
>>> 'ABC123'.islower()
False
```

9.1.16. *S*.isnumeric()

This method is a predicate that tests whether *S* is nonempty and contains only characters that are considered numeric characters in Unicode.

```
>>> '123'.isnumeric()
True
>>> ''.isnumeric()
False
```

9.1.17. *S*`.isprintable()`

A predicate that tests whether *S* is either empty or contains only printable characters. In the Unicode specification, a character is considered printable if it is not in classes `Other` or `Separator`, except for the space character which is a `Separator` but is considered printable.

```
>>> '\xa0'.isprintable()
False
>>> 'abc def $*^'.isprintable()
True
>>> ''.isprintable()
''.isprintable()
True
```

9.1.18. *S*`.isspace()`

Predicate that tests whether *S* is nonempty and all its characters are whitespace characters. In the example below, `'\xa0'` is a Unicode non-breaking space, which is considered a space character.

```
>>> ''.isspace()
False
>>> ' \t\r\n\xa0'.isspace()
True
>>> 'killer \t \n rabbit'.isspace()
False
```

9.1.19. *S*`.istitle()`

A predicate that tests whether *S* is nonempty and contains only words in "title case." In a title-cased string, uppercase characters may appear only at the beginning of the string or after some character that is not a letter. Lowercase characters may appear only after an uppercase letter.

```
>>> 'abc def GHI'.istitle()
False
>>> "Abc Def G Hij".istitle()
True
>>> ''.istitle()
False
```

9.2. The string `.format()` method

The `.format()` method of the `str` type is a convenient way to format text the way you want it.

Quite often, we want to embed data values in some explanatory text. For example, if we are displaying the number of nematodes in a hectare, it is a lot more meaningful to display it as `"There were 37.9`

nematodes per hectare" than just "37.9". So what we need is a way to mix constant text like "nematodes per hectare" with values from elsewhere in your program.

Here is the general form:

```
template.format(p0, p1, ..., k0=v0, k1=v1, ...)
```

The *template* is a string with one or more *replacement fields* embedded in constant text. Each replacement field is enclosed in single braces {...}, and specifies that a value is to be substituted at that position in the format string. The values to be substituted are passed as arguments to the .format() method.

The arguments to the .format() method are of two types. The list starts with zero or more positional arguments p_i, followed by zero or more keyword arguments of the form $k_i=v_i$, where each k_i is a name with an associated value v_i.

Just to give you the general flavor of how this works, here's a simple conversational example. In this example, the replacement field "{0}" is replaced by the first positional argument (49), and "{1}" is replaced by the second positional argument, the string "okra".

```
>>> "We have {0} hectares planted to {1}!".format(49, "okra")
'We have 49 hectares planted to okra!'
>>>
```

In the next example, we supply the values using keyword arguments. The arguments may be supplied in any order. The keyword names must be valid Python names (see Section 4, "Names and keywords" (p. 4)).

```
>>> "{monster} has now eaten {city}".format(
...     city='Tokyo', monster='Mothra')
'Mothra has now eaten Tokyo'
```

You may mix references to positional and keyword arguments:

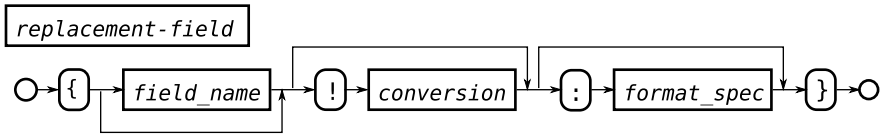
```
>>> "The {structure} sank {0} times in {1} years.".format(
...   3, 2, structure='castle')
'The castle sank 3 times in 2 years.'
```

If you need to include actual "{" and "}" characters in the result, double them, like this:

```
>>> "There are {0} members in set {{a}}.".format(15)
'There are 15 members in set {a}.'
```

9.2.1. General form of a replacement field

Here is the general form of a replacement field.



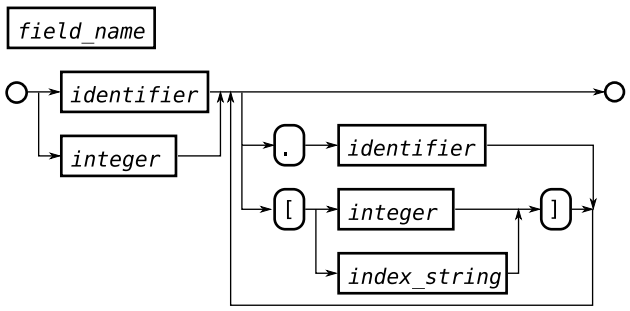
A *replacement_field* starts with an optional field name or number, optionally followed by a "!" and conversion specification, optionally followed by ":" and a format_spec that specifies the format.

- Section 9.2.2, "The *field_name* part" (p. 23).

- Section 9.2.3, "The *conversion* part" (p. 24).
- Section 9.2.4, "The *spec* part" (p. 24).
- Section 9.2.5, "Formatting a field of variable length" (p. 29): This is the trick necessary to format a field whose width is computed during program execution.

9.2.2. The *field_name* part

Here is the syntax for the *field_name* part of a replacement field, which specifies the source of the value to be formatted.



A *field_name* must start with either a number or a name.

- Numbers (*integer* in the diagram) refer to positional arguments passed to the `.format()` method, starting at 0 for the first argument.
- Names (*identifier* in the diagram) refer to keyword arguments to `.format()`.

Following this you can append any number of expressions that retrieve parts of the referenced value.

- If the associated argument has attributes, you can refer to them by using a period ("`.`") followed by the attribute name. For example:

```
>>> import string
 >>> string.digits
 '0123456789'
 >>> "Our digits are '{s.digits}'.".format(s=string)
 "Our digits are '0123456789'."
```

- If the associated argument is an iterable, you may extract an element from it by using an integer in square brackets `[...]`.

  For example:

```
>>> signal=['red', 'yellow', 'green']
 >>> signal[2]
 'green'
 >>> "The light is {0[2]}!".format(signal)
 'The light is green!'
```

- To extract an element from a mapping, use an expression of the form "`[k]`" where *k* is a key value, which may contain any character except "`]`".

```
>>> paintMap = {'M': 'blue', 'F': 'pink'}
>>> "Paint it {map[M]}.".format(map=paintMap)
'Paint it blue.'
```

In general, you can use any combination of these features. For example:

```
>>> "The sixth digit is '{s.digits[5]}'".format(s=string)
 "The sixth digit is '5'"
```

You may omit all of the numbers that refer to positional arguments, and they will be used in the sequence they occur. For example:

```
>>> "The date is {}-{}-{}.".format(2012, 5, 1)
 'The date is 2012-5-1.'
```

If you use this convention, you must omit all those numbers. You can, however, omit all the numbers and still use the keyword names feature:

```
>>> "Can I have {} pounds to {excuse}?".format(
...   50, excuse='mend the shed')
'Can I have 50 pounds to mend the shed?'
```

9.2.3. The *conversion* part

Following the *name* part of a replacement field, you can use one of these forms to force the value to be converted by a standard function:

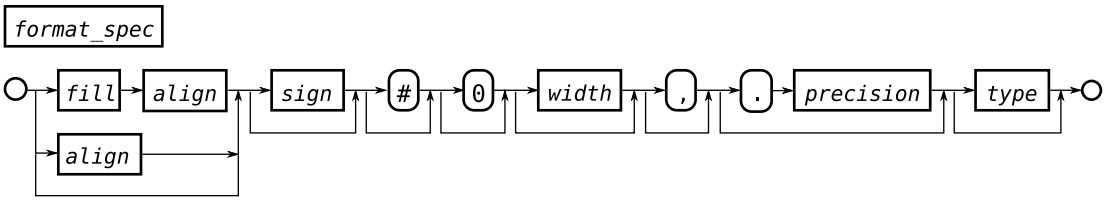
`!a`	Section 19.2, "`ascii()`: Convert to 8-bit ASCII" (p. 34).
`!r`	Section 19.13, "`repr()`: Printable representation" (p. 36).
`!s`	Section 19.14, "`str()`: Convert to `str` type" (p. 36). This is the default for values of types other than `str`.

Here's an example:

```
>>> "{}".format('Don\'t')
"Don't"
>>> "{!r}".format('Don\'t')
'"Don\'t"'
```

9.2.4. The *spec* part

After the *name* and *conversion* parts of a replacement field, you may use a colon ("`:`") and a format specifier to supply more details about how to format the related value. Here is the general form.



fill

You may specify any fill character except "`}`". This character is used to pad a short value to the specified length. It may be specified only in combination with an ***align*** character.

align

Specifies how to align values that are not long enough to occupy the specified length. There are four values:

<	Left-justify the value. This is the default alignment for string values.
>	Right-justify the value. This is the default alignment for numbers.
^	Center the value.
=	For numbers using a ***sign*** specifier, add the padding between the sign and the rest of the value.

Here are some examples of the use of ***fill*** and ***align***.

```
>>> "{:>8}".format(13)
'      13'
>>> "{:>8}".format('abc')
'     abc'
>>> "{:*>8}".format('abc')
'*****abc'
>>> "{:*<8}".format('abc')
'abc*****'
>>> "{:>5d}".format(14)
'   14'
>>> "{:#>5d}".format(14)
'###14'
>>> "{:<6}".format('Git')
'Git   '
>>> "{:*<6}".format('Git')
'Git***'
>>> "{:=^8}".format('Git')
'==Git==='
>>> "{:*=-9d}".format(-3)
'-*******3'
```

sign

This option controls whether an arithmetic sign is displayed. There are three possible values:

+	Always display a sign: + for positive, - for negative.
-	Display - only for negative values.
(one space)	Display one space for positive values, - for negative.

Here are some examples of use of the sign options.

```
>>> '{} {}'.format(17, -17)
'17 -17'
>>> '{:5} {:5}'.format(17, -17)
'   17   -17'
>>> '{:<5} {:<5}'.format(17, -17)
'17    -17  '
>>> '{:@<5} {:@<5}'.format(17, -17)
'17@@@ -17@@'
>>> '{:@>5} {:@>5}'.format(17, -17)
'@@@17 @@-17'
>>> '{:@^5} {:@^5}'.format(17, -17)
'@17@@ @-17@'
>>> '{:@^+5} {:@^+5}'.format(17, -17)
'@+17@ @-17@'
>>> '{:@^-5} {:@^-5}'.format(17, -17)
'@17@@ @-17@'
>>> '{:@^ 5} {:@^ 5}'.format(17, -17)
'@ 17@ @-17@'
```

#

This option selects the "alternate form" of output for some types.

- When formatting integers as binary, octal, or hexadecimal, the alternate form adds "`0b`", "`0o`", or "`0x`" before the value, to show the radix explicitly.

```
>>> "{:4x}".format(255)
'  ff'
>>> "{:#4x}".format(255)
'0xff'
>>>  "{:9b}".format(62)
'   111110'
>>> "{:#9b}".format(62)
' 0b111110'
>>> "{:<#9b}".format(62)
'0b111110 '
```

- When formatting `float`, `complex`, or `Decimal` values, the "`#`" option forces the result to contain a decimal point, even if it is a whole number.

```
>>>
"{:5.0f}".format(36)
'   36'
>>> "{:#5.0f}".format(36)
'  36.'
>>> from decimal import Decimal
>>> w=Decimal(36)
>>> "{:g}".format(w)
'36'
>>> "{:#g}".format(w)
'36.'
```

0

To fill the field with left zeroes, place a "`0`" at this position in your replacement field.

```
>>> "{:5d}".format(36)
'   36'
>>> "{:05d}".format(36)
'00036'
>>> "{:021.15}".format(1.0/7.0)
'00000.142857142857143'
```

width

Place a number at this position to specify the total width of the displayed value.

```
>>> "Beware the {}!".format('Penguin')
'Beware the Penguin!'
>>> "Beware the {:11}!".format('Penguin')
'Beware the Penguin    !'
>>> "Beware the {:>11}!".format('Penguin')
'Beware the     Penguin!'
```

,

Place a comma at this position in your replacement field to display commas between groups of three digits in whole numbers.

```
>>>  "{:,d}".format(12345678901234)
'12,345,678,901,234'
>>> "{:,f}".format(1234567890123.456789)
'1,234,567,890,123.456787'
>>> "{:25,f}".format(98765432.10987)
'        98,765,432.109870'
```

.precision

Use this part to specify the number of digits after the decimal point.

```
>>> from math import pi
>>> "{}".format(pi)
'3.141592653589793'
>>> "{:.3}".format(pi)
'3.14'
>>> "{:25,.3f}".format(1234567890123.456789)
'    1,234,567,890,123.457'
```

type

This code specifies the general type of format used. The default is to convert the value of a string as if using the `str()` function. Refer to the table below for allowed values.

b	Format an integer in binary.
c	Given a number, display the character that has that code.
d	Display a number in decimal (base 10).
e	Display a `float` value using the exponential format.
E	Same as `e`, but use a capital "`E`" in the exponent.
f	Format a number in fixed-point form.
g	General numeric format: use either `f` or `g`, whichever is appropriate.
G	Same as "`g`", but uses a capital "`E`" in the exponential form.

n	For formatting numbers, this format uses the current local setting to insert separator characters. For example, a number that Americans would show as "1,234.56", Europeans would show it as "1.234,56".
o	Display an integer in octal format.
x	Display an integer in hexadecimal (base 16). Digits greater than 9 are displayed as lowercase characters.
X	Display an integer in hexadecimal (base 16). Digits greater than 9 are displayed as uppercase characters.
%	Display a number as a percentage: its value is multiplied by 100, followed by a "%" character.

Examples:

```
>>> "{:b}".format(9)
'1001'
>>> "{:08b}".format(9)
'00001001'
>>> "{:c}".format(97)
'a'
>>> "{:d}".format(0xff)
'255'
>>> from math import pi
>>> "{:e}".format(pi*1e10)
'3.141593e+10'
>>> "{:E}".format(pi*1e10)
'3.141593E+10'
>>> "{:f}".format(pi)
'3.141593'
>>> "{:g}".format(pi)
'3.14159'
>>> "{:g}".format(pi*1e37)
'3.14159e+37'
>>> "{:G}".format(pi*1e37)
'3.14159E+37'
>>> "{:o}".format(255)
'377'
>>> "{:#o}".format(255)
'0o377'
>>> "{:x}".format(105199)
'19aef'
>>> "{:X}".format(105199)
'19AEF'
>>> "{:<#9X}".format(105199)
'0X19AEF  '
>>> "{:%}".format(0.6789)
'67.890000%'
>>> "{:15.3%}".format(0.6789)
'        67.890%'
```

9.2.5. Formatting a field of variable length

Sometimes you need to format a field using a length that is available only once the program is running. To do this, you can use a number or name in `{braces}` *inside* a replacement field at the *`width`* position. This item then refers to either a positional or keyword argument to the `.format()` method as usual.

Here's an example. Suppose you want to format a number `n` using `d` digits. Here are examples showing this with and without left-zero fill:

```
>>> n = 42
>>> d = 8
>>> "{0:{1}d}".format(42, 8)
'      42'
>>> "{0:0{1}d}".format(42, 8)
'00000042'
>>>
```

You can, of course, also use keyword arguments to specify the field width. This trick also works for variable precision.

```
"{count:0{width}d}".format(width=8, count=42)
'00000042'
>>>
```

The same technique applies to substituting any of the pieces of a replacement field.

```
>>> "{:&<14,d}".format(123456)
'123,456&&&&&&&'
>>> "{1:{0}{2}{3},{4}}".format('&', 123456, '<', 14, 'd')
'123,456&&&&&&&'
>>> "{:@^14,d}".format(1234567)
'@@1,234,567@@@'
>>> "{n:{fil}{al}{w},{kind}}".format(
...     kind='d', w=14, al='^', fil='@', n=1234567)
'@@1,234,567@@@'
```

10. Type `bytes`: Immutable sequences of 8-bit integers

11. Type `bytearray`: Mutable sequences of 8-bit integers

12. Type `list`: Mutable sequences of arbitrary objects

13. Type `tuple`: Immutable sequences of arbitrary objects

14. Type `range`: A range of values

15. The set types: `set` and `frozenset`

16. Type `dict`: Mappings

17. Type `None`: The special placeholder value

18. Operators and expressions

Python's operators are shown here from highest precedence to lowest, with a ruled line separating groups of operators with equal precedence:

Table 2. Python operator precedence

`(E)`	Parenthesized expression or tuple.
`[E, ...]`	List.
`{key:value, ...}`	Dictionary or set.
`` `...` ``	Convert to string representation.
`x.attribute`	Attribute reference.
`x[...]`	Subscript or slice; see Section 8.3, "Common operations on sequence types" (p. 13).
`f(...)`	Call function `f`.
`x**y`	`x` to the `y` power.
`-x`	Negation.
`~x`	Bitwise not (one's complement).
`x*y`	Multiplication.
`x/y, x//y`	Division. The "`//`" form discards the fraction from the result. For example, "`13.9//5.0`" returns the value `2.0`.
`x%y`	Modulo (remainder of `x/y`).
`x+y`	Addition, concatenation.
`x-y`	Subtraction.
`x<<y`	`x` shifted left `y` bits.
`x>>y`	`x` shifted right `y` bits.
`x&y`	Bitwise and.
`x^y`	Bitwise exclusive or.
`x|y`	Bitwise or.
`x<y, x<=y, x>y, x>=y, x!=y, x==y`	Comparisons. These operators are all predicates; see Section 18.1, "What is a predicate?" (p. 31).
`x in y, x not in y`	Test for membership.
`x is y, x is not y`	Test for identity.
`not x`	Boolean "not."
`x and y`	Boolean "and."
`x or y`	Boolean "or."

18.1. What is a predicate?

We use the term *predicate* to mean any Python function that tests some condition and returns a Boolean value.

For example, `x < y` is a predicate that tests whether `x` is less than `y`. For example, `5 < 500` returns `True`, while `5 >= 500` returns `False`.

18.2. What is an iterable?

To *iterate over* a sequence means to visit each element of the sequence, and do some operation for each element.

In Python, we say that a value is an *iterable* when your program can iterate over it. In short, an iterable is a value that represents a sequence of one more values.

All instances of Python's sequence types are iterables. These types may be referred to as *container types*: a `unicode` string string is a container for 32-bit characters, and lists and tuples are general-purpose containers that can contain any sequence.

One of the most common uses for an iterable is in a `for` statement, where you want to perform some operation on a sequence of values. For example, if you have a tuple named `celsiuses` containing Celsius temperatures, and you want to print them with their Fahrenheit equivalents, and you have written a function `cToF()` that converts Celsius to Fahrenheit, this code does it:

```
>>> def cToF(c): return c*9.0/5.0 + 32.0
...
>>> celsiuses = (0, 20, 23.6, 100)
>>> for celsius in celsiuses:
...     print "{0:.1f} C = {1:.1f} F".format(celsius, cToF(celsius))
...
0.0 C = 32.0 F
20.0 C = 68.0 F
23.6 C = 74.5 F
100.0 C = 212.0 F
```

However, Python also supports mechanisms for *lazy evaluation*: a piece of program that acts like a sequence, but produces its contained values one at a time.

Keep in mind that the above code works exactly the same if `celsiuses` is an iterator (see Section 23.3, "Iterators: Values that can produce a sequence of values" (p. 47)). You may find many uses for iterators in your programs. For example, `celsiuses` might be a system that goes off and reads an actual thermometer and returns the readings every ten seconds. In this application, the code above doesn't care where `celsiuses` gets the values, it cares only about how to convert and print them.

18.3. Duck typing, or: what is an interface?

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.
>
> —James Whitcomb Riley

The term *duck typing* comes from this quote. In programming terms, this means that the important thing about a value is what it can do, not its type. As the excellent Wikipedia article on duck typing[15] says, "Simply stated: provided you can perform the job, we don't care who your parents are."

One common example of duck typing is in the Python term "file-like object". If you open a file for reading using the `open()` function, you get back a value of type `file`:

```
>>> inFile = open('input')
>>> type(inFile)
<type 'file'>
```

[15] http://en.wikipedia.org/wiki/Duck_typing

Let's suppose that you write a function called `numberIt()` that takes a readable `file` as an argument and prints the lines from a file preceded by five-digit line numbers. Here's the function and an example of its use:

```
>>> def numberIt(f):
...     for lineNo, line in enumerate(f):
...         print "{0:05d} {1}".format(lineNo, line.rstrip())
...
>>> numberIt(inFile)
00000 Kant
00001 Heidegger
00002 Hume
```

The way you have written the `numberIt()` function, it works for files, but it also works for any iterable.

Thus, when you see the statement that some Python feature works with a "file-like object," that means that the object must have an interface like that of the `file` type; Python doesn't care about the type, just the operations that it supports.

In practice, the `enumerate()` function works with any iterable, so your function will also work with any iterable:

```
>>> numberIt(['Kant', 'Heidegger', 'Hume'])
00000 Kant
00001 Heidegger
00002 Hume
```

So in Python when we say that we expect some value to have an *interface*, we mean that it must provide certain methods or functions, but the actual type of the value is immaterial.

More formally, when we say that a value supports the *iterable interface*, that value must provide either of the following features:

- A `.__getitem__()` method as described in Section 25.2.2, "`__getitem__()`: Get one item from a sequence or mapping" (p. 50).
- A `.__iter__()` method as described in Section 25.2.3, "`.__iter__()`: Create an iterator" (p. 50).

18.4. What is the locale?

In order to accommodate different character encodings, your system may have a *locale* setting that specifies a preferred character set.

In the USA, most systems use the ASCII[16] encoding. Good application code should be written in a way that does not depend on this encoding to deal with cultural issues.

For general information on handling locale issues, see the documentation for the `locale` module[17].

[16] http://en.wikipedia.org/wiki/ASCII
[17] http://docs.python.org/py3k/library/locale.html

18.5. Comprehensions

19. Basic built-in functions

This section describes some common built-in Python functions, ones that you will likely use most of the time. A number of more exotic functions are discussed in Section 20, "Advanced functions" (p. 36).

19.1. `abs()`: Absolute value

19.2. `ascii()`: Convert to 8-bit ASCII

19.3. `bool()`: Convert to `bool` type

19.4. `complex()`: Convert to `complex` type

19.5. `input()`: Read a string from standard input

The purpose of this function is to read input from the standard input stream (the keyboard, by default). Without an argument, it silently awaits input. If you would like to prompt the user for the input, pass the desired prompt string as the argument. The result is returned as a `str` (string) value, without a trailing newline character. In the example shown just below, the second line is typed by the user.

```
>>> s = input()
This tobacconist is scratched.
>>> s
s
'This tobacconist is scratched.'
>>> yesNo = input("Is this the right room for an argument? ")
Is this the right room for an argument? I've told you once.
>>> yesNo
"I've told you once."
```

19.6. `int()`: Convert to `int` type

To convert a number of a different type to `int` type, or to convert a string of characters that represents a number:

```
int(ns)
```

where *`ns`* is the value to be converted. If *`ns`* is a `float`, the value will be truncated, discarding the fraction.

If you want to convert a character string *`s`*, expressed in a radix (base) other than 10, to an `int`, use this form, where *`b`* is an integer in the range [2, 36] that specifies the radix.

If the string to be converted obeys the Python prefix conventions (octal values start with `0o`, hex values with `0x`, and binary values with `0b`), use zero as the second argument.

```
int(s, b)
```

Examples:

```
>>> int(True)
1
>>> int(False)
0
>>> int(43.89)
43
>>> int("43")
43
>>> int('77', 8)
63
>>> int('7ff', 16)
2047
>>> int('10101', 2)
21
>>> int('037')
37
>>> int('037', 8)
31
>>> int('0o37', 0)
31
>>> int('0x1f', 0)
31
```

19.7. `iter()`: Return an iterator for a given sequence

19.8. `len()`: How many elements in a sequence?

19.9. `max()`: What is the largest element of a sequence?

19.10. `min()`: What is the smallest element of a sequence?

19.11. `open()`: Open a file

19.12. `ord()`: What is the code point of this character?

The function `ord(s)` operates on a string *s* that containsy exactly one character. It returns the Unicode code point of that character as type `int`.

```
>>> ord('@')
64
>>> ord('\xa0')
160
```

19.13. `repr()`: Printable representation

19.14. `str()`: Convert to `str` type

20. Advanced functions

21. Simple statements

Python statement types are divided into two groups. Simple statements, that are executed sequentially and do not affect the flow of control, are described first. Compound statements, which may affect the sequence of execution, are discussed in Section 22, "Compound statements" (p. 37).

Here, for your convenience, is a table of all the Python statement types, and the sections where they are described. The first one, the assignment statement, does not have an initial keyword: an assignment statement is a statement of the form "*`variable = expression`*".

Expression	Section 21.1, "The expression statement" (p. 37)
Assignment	Section 21.2, "The assignment statement: *`name = expression`*" (p. 37).
`assert`	Section 21.3, "The `assert` statement" (p. 37).
`break`	Section 22.2, "The `break` statement: Exit a `for` or `while` loop" (p. 39).
`continue`	Section 22.3, "The `continue` statement: Jump to the next cycle of a `for` or `while`" (p. 39).
`del`	Section 21.4, "The `del` statement" (p. 37).
`elif`	Section 22.5, "The `if` statement" (p. 41) and Section 22.6, "The `try`…`except` construct" (p. 41).
`else`	Section 22.5, "The `if` statement" (p. 41) and Section 22.6, "The `try`…`except` construct" (p. 41).
`except`	Section 22.6, "The `try`…`except` construct" (p. 41).
`finally`	Section 22.6, "The `try`…`except` construct" (p. 41).
`for`	Section 22.4, "The `for` statement: Iteration over a sequence" (p. 40).
`from`	Section 21.5, "The `import` statement" (p. 37).
`global`	Section 21.6, "The `global` statement" (p. 37).
`if`	Section 22.5, "The `if` statement" (p. 41).
`import`	Section 21.5, "The `import` statement" (p. 37).
`nonlocal`	Section 21.7, "The `nonlocal` statement" (p. 37).
`pass`	Section 21.8, "The `pass` statement" (p. 37).
`raise`	Section 21.9, "The `raise` statement" (p. 37).
`return`	Section 21.10, "The `return` statement" (p. 37).
`try`	Section 22.6, "The `try`…`except` construct" (p. 41).
`with`	Section 22.7, "The `with` statement" (p. 41).
`yield`	Section 22.8, "The `yield` statement: Generate one result of a generator" (p. 41).

21.1. The expression statement

A statement may consist of just a single Python expression by itself. For example, to call a function named `xParrot` with no arguments, this is a valid statement:

```
xParrot()
```

21.2. The assignment statement: *`name = expression`*

21.3. The `assert` statement

21.4. The `del` statement

21.5. The `import` statement

21.6. The `global` statement

21.7. The `nonlocal` statement

21.8. The `pass` statement

21.9. The `raise` statement

21.10. The `return` statement

22. Compound statements

The statements in this section alter the normal sequential execution of a program. They can cause a statement to be executed only under certain circumstances, or execute it repeatedly.

22.1. Python's block structure

One unusual feature of Python is the way that the indentation of your source program organizes it into blocks within blocks within blocks. This is contrary to the way languages like C and Perl organize code blocks by enclosing them in delimiters such as braces `{ ... }`.

Various Python branching statements like `if` and `for` control the execution of blocks of lines.

- At the very top level of your program, all statements must be unindented—they must start in column one.
- Various Python branching statements like `if` and `for` control the execution of one or more subsidiary blocks of lines.

- A block is defined as a group of adjacent lines that are indented the same amount, but indented further than the controlling line. The amount of indentation of a block is not critical.
- You can use either spaces or *tab* characters for indentation. However, mixing the two is perverse and can make your program hard to maintain. Tab stops are assumed to be every eight columns.

Blocks within blocks are simply indented further. Here is an example of some nested blocks:

```
if i < 0:
    print "i is negative"
else:
    print "i is nonnegative"
    if i < 10:
        print "i has one digit"
    else:
        print "i has multiple digits"
```

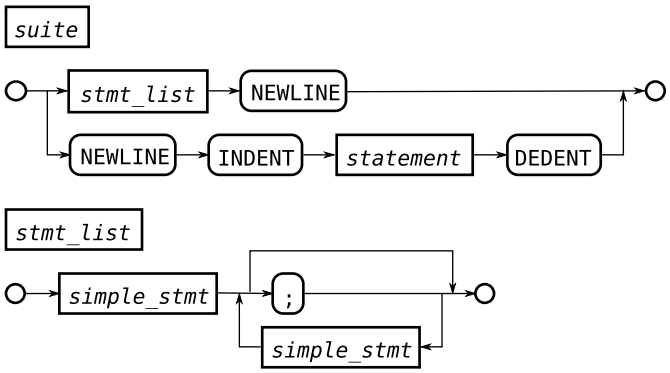
If you prefer a more horizontal style, you can always place statements after the colon (`:`) of a compound statement, and you can place multiple statements on a line by separating them with semicolons (`;`). Example:

```
>>> if 2 > 1: print "Math still works"; print "Yay!"
... else: print "Huh?"
...
Math still works
Yay!
```

You can't mix the block style with the horizontal style: the consequence of an `if` or `else` must either be on the same line or in a block, never both.

```
>>> if 1: print "True"
...     print "Unexpected indent error here."
  File "<stdin>", line 2
    print "Unexpected indent error here."
    ^
IndentationError: unexpected indent
>>>
```

Here is a formal definition of Python's block structure. A *`suite`* is the sequence of statements that are executed following the "`:`" after an `if`, `else`, or other compound statement.



In this diagram, `INDENT` refers to an increase in the amount of indentation, and `DEDENT` is where the indentation decreases to its former level.

Note that a trailing "`;`" is allowed and ignored on any line.

22.2. The `break` statement: Exit a `for` or `while` loop

The purpose of this statement is to jump out of a `for` or `while` loop before the loop would terminate otherwise. Control is transferred to the statement after the last line of the loop. The statement looks like this:

```
break
```

Here's an example.

```
>>> for i in [1, 71, 13, 2, 81, 15]:
...     print(i, end=' ')
...     if (i%2) == 0:
...         break
...
1 71 13 2
```

Normally this loop would be executed six times, once for each value in the list, but the `break` statement gets executed when `i` is set to an even value.

22.3. The `continue` statement: Jump to the next cycle of a `for` or `while`

Use a `continue` statement inside a `for` or `while` loop when you want to jump directly back to the top of the loop and go around again.

- If used inside a `while` loop, the loop's condition expression is evaluated again. If the condition is `False`, the loop is terminated; if the condition is `True`, the loop is executed again.
- Inside a `for` loop, a `continue` statement goes back to the top of the loop. If there are any values remaining in the iterable that controls the loop, the loop variable is set to the next value in the iterable, and the loop body is entered.

  If the `continue` is executed during the last pass through the loop, control goes to the statement after the end of the loop.

Examples:

```
>>> i = 0
>>> while i < 10:
...     print(i, end=' ')
...     i += 1
...     if (i%3) != 0:
...         continue
...     print("num", end=' ')
...
0 1 2 num 3 4 5 num 6 7 8 num 9
>>> for i in range(10):
...     print(i, end=' ')
...     if (i%4) != 0:
...         continue
```

```
...     print("whee", end=' ')
...
0 whee 1 2 3 4 whee 5 6 7 8 whee 9
```

22.4. The `for` statement: Iteration over a sequence

Use a `for` statement to execute a block of statements repeatedly. Here is the general form. (For the definition of a *`suite`*, see Section 22.1, "Python's block structure" (p. 37).)

```
for V in S:
    B
```

- *`V`* is a variable called the *induction variable*.
- *`S`* is any iterable; see Section 18.2, "What is an iterable?" (p. 32).

  This iterable is called the *controlling iterable* of the loop.
- *`B`* is a block of statements.

The block is executed once for each value in *`S`*. During each execution of the block, *`V`* is set to the corresponding value of *`S`* in turn. Example:

```
>>> for color in ['black', 'blue', 'transparent']:
...     print color
...
black
blue
transparent
```

In general, you can use any number of induction variables. In this case, the members of the controlling iterable must themselves be iterables, which are unpacked into the induction variables in the same way as sequence unpacking as described in Section 21.2, "The assignment statement: *`name = expression`*" (p. 37). Here is an example.

```
>>> fourDays = ( ('First', 1, 'orangutan librarian'),
...              ('Second', 5, 'loaves of dwarf bread'),
...              ('Third', 3, 'dried frog pills'),
...              ('Fourth', 2, 'sentient luggages') )
>>> for day, number, item in fourDays:
...     print ( "On the {1} day of Hogswatch, my true love gave "
...             "to me".format(day) )
...     print "{0} {1}".format(number, item)
...
On the First day of Hogswatch, my true love gave to me
1 orangutan librarian
On the Second day of Hogswatch, my true love gave to me
5 loaves of dwarf bread
On the Third day of Hogswatch, my true love gave to me
3 dried frog pills
On the Fourth day of Hogswatch, my true love gave to me
2 sentient luggages
```

You can change the induction variable inside the loop, but during the next pass through the loop, it will be set to the next element of the controlling iterable normally. Modifying the controlling iterable itself won't change anything; Python makes a copy of it before starting the loop.

```
>>> for i in range(4):
...     print "Before:", i,
...     i += 1000
...     print "After:", i
...
Before: 0 After: 1000
Before: 1 After: 1001
Before: 2 After: 1002
Before: 3 After: 1003
>>> L = [7, 6, 1912]
>>> for n in L:
...     L = [44, 55]
...     print n
...
7
6
1912
```

22.5. The `if` statement

22.6. The `try...except` construct

22.7. The `with` statement

22.8. The `yield` statement: Generate one result of a generator

23. `def()`: Defining your own functions

Use the `def` construct to define a functions. Inside a `class`, use the `def` construct to define a method. Each definition has three parts:

- You may supply any number of *decorators* before the line starting the `def`. See Section 23.5, "Decorators" (p. 49).
- A statement starting with the `def` keyword names the function or method you are defining, and describes any parameters that are passed in.
- The body of the function or method is an indented block. This block is executed when the function is called.

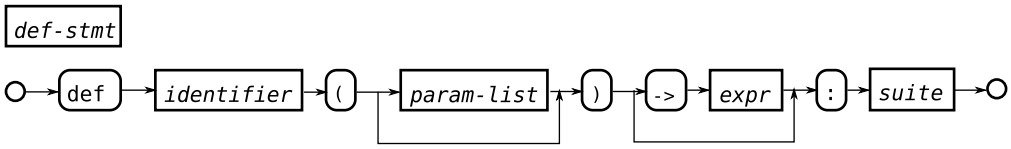
Here is a brief interactive example of a function to compute cube roots.

```
>>> def cubeRoot(x):
...     '''Returns the cube root of a number x.
...     '''
...     return x**(1.0/3.0)
```

```
...
>>> cubeRoot(27.1)
3.00369914061521
>>> cubeRoot.__doc__
'Returns the cube root of a number x.\n    '
```

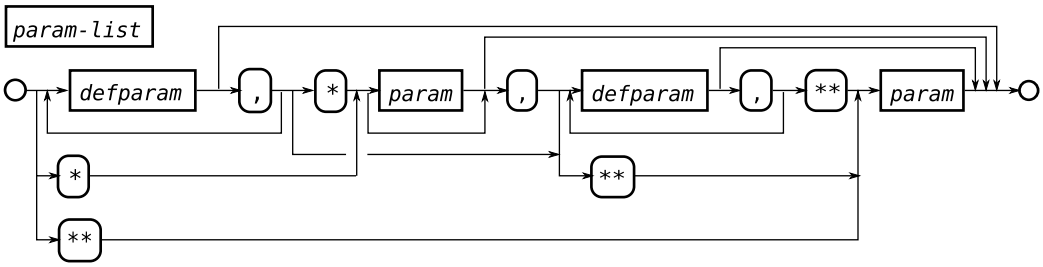
The above example demonstrates use of a *documentation string*: if the first indented line after the `def` is a string constant, that constant is saved as the documentation string for that function, and is available as the `.__doc__` attribute of the function.

Here is the syntax of the `def` statement.



- The `identifier` is the name of the function you are defining.
- The `param_list` defines the parameters to the function. It is optional.
- The "`->`" followed by an expression is optional. It may be used to annotate the expected result type, e.g., "`def f(k) -> int: …`".
- For the definition of `suite`, see Section 22, "Compound statements" (p. 37).

Here is the syntax for `param_list`.



- Each `param` consists of an identifier, optionally followed by a colon and an expression that annotates the expected type. This annotation, if given, is not checked; it is by way of documentation.

  For example, "`def f(i:int)`" declares a parameter "`i`" and annotates that an `int` value is expected.
- Each `defparam` is a `param`, optionally followed by an "`=`" and a default value for that parameter.

Warning

The default value expression is evaluated *when the function is defined*. This value is stored away and bound to the corresponding name each time the function is called without a corresponding argument.

When the default value expresion is a mutable value, this can lead to undesirable side effects. For example:

```
>>> def f(x, L=[]):
...     L.append(x)
...     print(L)
```

```
...
>>> f(5)
[5]
>>> f(10)
[5, 10]
```

When the function is defined, a new empty list is created and saved as the default value of the function. The first call adds the value 5 to the saved default value, so when it is called the second time, the default value that is used already has the value 5 in it.

To avoid this behavior, use `None` as the default value, and then add code that detects this case and creates the default value anew each time:

```
>>> def f2(x, L=None):
...     if L is None:
...         L=[]
...     L.append(x)
...     print(L)
...
>>> f2(5)
[5]
>>> f2(10)
[10]
```

- If a single "*" is given, followed by `param`, that name is bound to a tuple containing any extra positional arguments. See Section 23.1, "Calling a function" (p. 44).

  If a single "*" appears but is not followed by a `param`, it signifies that all the following arguments must be specified using the keyword, "*`name=expr`*" syntax.

  Here is an example. This function has one required positional argument, but it also requires one keyword argument named `b`.

  ```
  >>> def f(a, *, b):
  ...     print(a, b)
  ...
  >>> f(0, b=4)
  0 4
  >>> f(0, 4)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: f() takes exactly 1 positional argument (2 given)
  >>> f(0, 4, b=5)
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  TypeError: f() takes exactly 1 non-keyword positional argument (2 given)
  ```

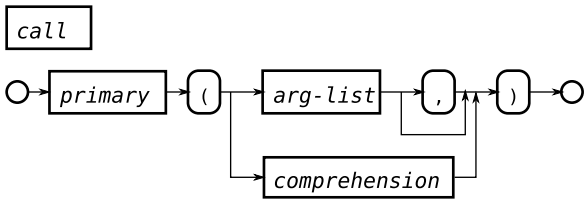
- If the parameter list includes "**" followed by a `param`, that name is bound to a dictionary containing all the extra keyword arguments passed in the function call. For example:

  ```
  >>> def f(*p, **kw):
  ...     print(p, kw)
  ...
  ```

```
>>> f()
() {}
>>> f(4, 88, name='Clem', clan='Bozo')
(4, 88) {'clan': 'Bozo', 'name': 'Clem'}
```

23.1. Calling a function

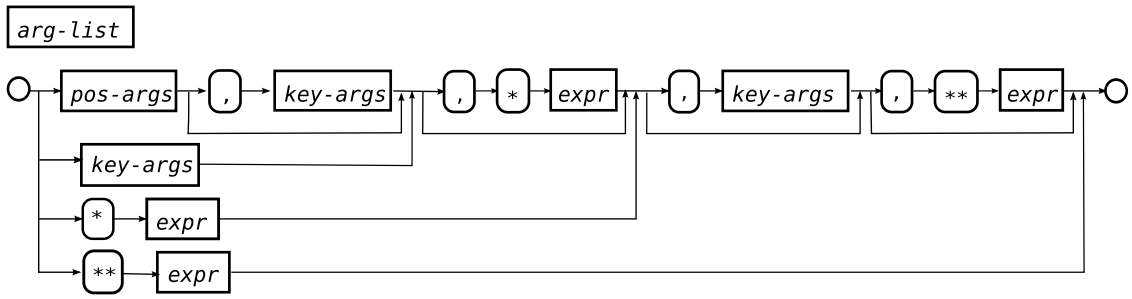
Here is the syntax for a function call.



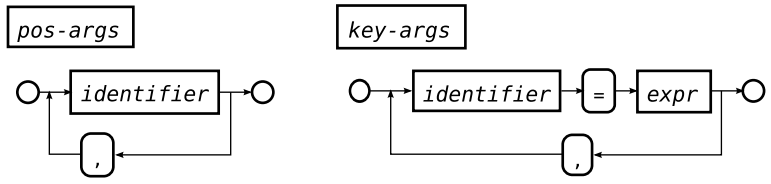
The `primary` is the name of the thing you are calling. Callable values include:

- Built-in functions and methods of builtin types.
- User-defined functions and methods of user-defined classes.
- Instances of any user-defined class that defines the special method Section 25.2.1, "`.__call__()`: What to do when someone calls an instance" (p. 50).

The `arg_list` is the list of arguments you are passing; a trailing comma is ignored if present. You may, instead of the usual argument list, include one `comprehension` expression (see Section 18.5, "Comprehensions" (p. 34)); in that case, the function will be passed one positional argument, which will be a generator.



In the above diagram, `pos_args` refers to a sequence of positional arguments, and `key-args` is a sequence of keyword arguments.



- The sequence "*`*expr`*" indicates that the associated name will be bound to a tuple containing all extra positional arguments. See the rules for evaluation below.

- The sequence "`**expr`" indicates that the associated name will be bound to a dictionary containing all extra keyword arguments.
- If you declare additional `key-args` after the "`*expr`" item, those arguments may be passed *only* as keyword arguments; they may never match a position argument.

This is how the arguments actually passed are matched with the parameters to the function.

1. Python creates a list of unfilled slots, one for each declared parameter.
2. The actual positional arguments (*`pos_args`*) are placed into those slots in order.
3. For each keyword argument (*`key-args`*), if the slot for that name is empty, it is filled with that argument. If the slot was already full, Python raises a `TypeError`.
4. All unfilled slots are filled with the default value. If any unfilled slots do not have a default value, Python raises a `TypeError`.
5. If there are more positional arguments than there are slots for them, they are bound to the "`*expr`" item is a tuple, if there is one; otherwise Python raises a `TypeError`.
6. If there are keyword arguments whose names do not match any of the keyword parameters, they are bound to the "`**expr`" item as a dictionary, if there is one; otherwise Python raises a `TypeError`.

When you call a function, the argument values you pass to it must obey these rules:

- There are two kinds of arguments: positional (also called non-default arguments) and keyword (also called default arguments). A positional argument is simply an expression, whose value is passed to the argument.

  A keyword argument has this form:

```
name=expression
```

- All positional arguments in the function call (if any) must precede all keyword arguments (if any).

```
>>> def wrong(f=1, g):
...     print f, g
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

- You must supply at least as many positional arguments as the function expects.

```
>>> def wantThree(a, b, c):
...     print a,b,c
...
>>> wantThree('nudge', 'nudge', 'nudge')
nudge nudge nudge
>>> wantThree('nudge')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wantThree() takes exactly 3 arguments (1 given)
```

- If you supply more positional arguments than the function expects, the extra arguments are matched against keyword arguments in the order of their declaration in the `def`. Any additional keyword arguments are set to their default values.

```
>>> def f(a, b, c=1, d='elk'):
...     print a,b,c,d
...
>>> f(99, 111)
99 111 1 elk
>>> f(99, 111, 222, 333)
99 111 222 333
>>> f(8, 9, 10, 11, 12, 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes at most 4 arguments (6 given)
```

- You may supply arguments for keyword parameters in any order by using the form *k=v*, where *k* is the keyword used in the declaration of that parameter and *v* is your desired argument.

```
>>> def blackKeys(fish='Eric', dawn='Stafford', attila='Abdul'):
...     print fish, dawn, attila
...
>>> blackKeys()
Eric Stafford Abdul
>>> blackKeys(attila='Gamera', fish='Abdul')
Abdul Stafford Gamera
```

- If you declare a parameter of the form "**name*", the caller can provide any number of additional keyword arguments, and the *name* will be bound to a tuple containing those additional arguments.

```
>>> def posish(i, j, k, *extras):
...     print i,j,k,extras
...
>>> posish(38, 40, 42)
38 40 42 ()
>>> posish(44, 46, 48, 51, 57, 88)
44 46 48 (51, 57, 88)
```

- Similarly, you may declare a final parameter of the form "***name*". If the caller provides any keyword arguments whose names do not match declared keyword arguments, that *name* will be bound to a dictionary containing the additional keyword arguments as key-value pairs.

```
>>> def extraKeys(a, b=1, *c, **d):
...     print a, b, c, d
...
>>> extraKeys(1,2)
1 2 () {}
>>> extraKeys(3,4,6,12, hovercraft='eels', record='scratched')
3 4 (6, 12) {'record': 'scratched', 'hovercraft': 'eels'}
```

23.2. A function's local namespace

Any name that appears in a function's argument list, or any name that is set to a value anywhere in the function, is said to be *local* to the function. If a local name is the same as a name from outside the function (a so-called *global* name), references to that name inside the function will refer to the local name, and the global name will be unaffected. Here is an example:

```
>>> x = 'lobster'
>>> y = 'Thermidor'
>>> def f(x):
...     y = 'crevettes'
...     print x, y
...
>>> f('spam')
spam crevettes
>>> print x, y
lobster Thermidor
```

Keyword parameters have a special characteristic: their names are local to the function, but they are also used to match keyword arguments when the function is called.

23.3. Iterators: Values that can produce a sequence of values

Closely related to Python's concept of sequences is the concept of an *iterator*:

> For a given sequence *S*, an iterator *I* is essentially a set of instructions for producing the elements of *S* as a sequence of zero or more values.

To produce an iterator over some sequence *S*, use this function:

```
iter(S)
```

- The result of this function is an "iterator object" that can be used in a `for` statement.

```
>>> continents = ('AF', 'AS', 'EU', 'AU', 'AN', 'SA', 'NA')
>>> worldWalker = iter(continents)
>>> type(worldWalker)
<type 'tupleiterator'>
>>> for landMass in worldWalker:
...     print "Visit {0}.".format(landMass,)
...
Visit AF. Visit AS. Visit EU. Visit AU. Visit AN. Visit SA. Visit NA.
```

- All iterators have a `.next()` method that you can call to get the next element in the sequence. This method takes no arguments. It returns the next element in the sequence, if any. When there are no more elements, it raises a `StopIteration` exception.

```
>>> trafficSignal = [ 'green', 'yellow', 'red' ]
>>> signalCycle = iter(trafficSignal)
>>> type(signalCycle)
<type 'listiterator'>
>>> signalCycle.next()
'green'
>>> signalCycle.next()
'yellow'
>>> signalCycle.next()
'red'
>>> signalCycle.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Once an iterator is exhausted, it will continue to raise `StopIteration` indefinitely.

- You can also use an iterator as the right-hand operand of the "`in`" operator.

```
>>> signalCycle = iter(trafficSignal)
>>> 'red' in signalCycle
True
```

23.4. Generators: Functions that can produce a sequence of values

Unlike conventional functions that return only a single result, a *generator* is a function that produces a sequence of zero or more results.

Generators are a special case of iterators (see Section 23.3, "Iterators: Values that can produce a sequence of values" (p. 47)), so they can be used as the controlling iterable in `for` statements and the other places where iterators are allowed.

In a conventional function, the body of the function is executed until it either executes a `return` statement, or until it runs out of body statements (which is the equivalent of a "`return None`" statement).

By contrast, when a generator function is called, its body is executed until it either has another value to produce, or until there are no more values.

- When a function wishes to return the next generated value, it executes a statement of this form:

```
    yield e
```

  where the *e* is any Python expression.

  The difference between `yield` and `return` is that when a `return` is executed, the function is considered finished with its execution, and all its current state diasppears.

  By contrast, when a function executes a `yield` statement, execution of the function is expected to resume just after that statement, at the point when the caller of the function needs the next generated value.

- A generator signals that there are no more values by executing this statement:

```
    raise StopIteration
```

For an example of a generator, see Section 22.8, "The `yield` statement: Generate one result of a generator" (p. 41).

If you are writing a container class (that is, a class whose instances are containers for a set of values), and you want to define an iterator (see Section 25.2.3, "`.__iter__()`: Create an iterator" (p. 50)), that method can be a generator. Here is a small example. The constructor for class `Bunch` takes a sequence of values and stores them in instance attribute `.__stuffList`. The iterator method `.__iter__()` generates the elements of the sequence in order, except it wraps each of them in parentheses:

```
>>> class Bunch(object):
...     def __init__(self, stuffList):
...         self.__stuffList  =  stuffList
...     def __iter__(self):
...         for thing in self.__stuffList:
...             yield "({0})".format(thing)
...         raise StopIteration
...
```

```
>>> mess = Bunch(('lobster Thermidor', 'crevettes', 'Mornay'))
>>> for item in mess:
...     print item,
...
(lobster Thermidor) (crevettes) (Mornay)
>>> messWalker = iter(mess)
>>> for thing in messWalker: print thing,
...
(lobster Thermidor) (crevettes) (Mornay)
```

23.5. Decorators

The purpose of a Python decorator is to replace a function or method with a modified version *at the time it is defined*. For example, the original way to declare a static method was like this:

```
    def someMethod(x, y):
        ...
    someMethod = staticmethod(someMethod)
```

Using Python's decorator syntax, you can get the same effect like this:

```
    @staticmethod
    def someMethod(x, y):
         ...
```

In general, a function or method may be preceded by any number of decorator expressions, and you may also provide arguments to the decorators.

- If a function *f* is preceded by a decorator expression of the form "@*d*", it is the equivalent of this code:

  ```
  def f(...):
      ...
  f = d(f)
  ```

- You may provide a parenthesized argument list after the name of your decorator. A decorator expression *d*`(...)` is the equivalent of this code:

  ```
  def f(...):
      ...
  f = d(...)(f)
  ```

  First, the decorator is called with the argument list you provided. It must return a callable object. That callable is then called with one argument, the decorated function. The name of the decorated function is then bound to the returned value.

- If you provide multiple decorators, they are applied inside out, in sequence from the last to the first.

Here is an example of a function wrapped with two decorators, of which the second has additional arguments:

```
@f1
@f2('Pewty')
def f0(...):
    ...
```

This is the equivalent code without using decorators:

```
def f0(...):
    ...
f0 = f1 ( f2('Pewty') ( f0 ) )
```

First function `f2` is called with one argument, the string `'Pewty'`. The return value, which must be callable, is then called with `f0` as its argument. The return value from that call is then passed to `f1`. Name `f0` is then bound to the return value from the call to `f1`.

24. Exceptions

25. Classes: invent your own types

25.1. Defining a class

25.2. Special methods

25.2.1. `.__call__()`: What to do when someone calls an instance

25.2.2. `__getitem__()`: Get one item from a sequence or mapping

If a class defines it, this special method is called whenever a value is retrieved from a sequence or mapping (dictionary-like object) using the syntax "*v*`[`*i*`]`", where *v* is the sequence or mapping and *i* is a position in a sequence, or a key in a mapping.

Here is the calling sequence:

```
    def __getitem__(self, i):
        ...
```

The method either returns the corresponding item or raises an appropriate exception: `IndexError` for sequences or `KeyError` for mappings.

25.2.3. `.__iter__()`: Create an iterator

25.3. Static methods

26. The conversion path from 2.x to 3.x

At this writing, both Python 2.7 and Python 3.2 are officially maintained implementations. The 3.0 release marked the first release in the development of Python that was incompatible with the old one.

If you are using 2.x releases of Python, there is no hurry to convert to the 3.x series. Release 2.7 is guaranteed to be around for many years. Furthermore, there are tools to help you automate much of the conversion process.

- For a discussion of the changes between 2.7 and 3.2, see What's New in Python[18].
- To see what changes must be made in your program to allow automatic conversion to Python 3.x, run Python with this flag:

```
python -3 yourprogram
```

- To convert your program to Python 3.x, first make a copy of the original program, then run this command:

```
python3-2to3 -w yourprogram
```

  The `-w` flag replaces *yourprogram* with the converted 3.x version, and moves the original to "*yourprogram*`.bak`"

[18] http://docs.python.org/py3k/whatsnew/