Python 2.2 quick reference

John W. Shipman

2008-01-02 21:34

Abstract

A quick reference for the Python programming language, version 2.2.

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/python22/
[2] http://www.nmt.edu/tcc/help/pubs/python22/python22.pdf

1. What is Python?

Python is a recent, general-purpose, high-level programming language. It is freely available and runs pretty much everywhere.

- For local documentation, see the Python help page.[3]

[3] http://www.nmt.edu/tcc/help/lang/python/

- Complete documentation will be found at the `python.org` homepage[4].

2. Starting Python

The Python language can be run interactively in "calculator mode" using the command:

```
python
```

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

```
python filename.py
```

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

```
#!/usr/local/bin/python
```

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 an open `(`, `[`, or `{`.

4. Reserved words

`and`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `not`, `or`, `pass`, `print`, `raise`, `return`, `try`, `while`, and `yield`.

5. Basic types

Here is a list of the most common Python types. For a full list, see the `types` module below.

Type name	Values	Examples
`int`	Integers in the range [-2147483648, 2147483647]. See the `int` type below.	`42`, `-3`
`long`	Integers of any size, limited only by total memory size. See the `long` type below.	`42L`, `-3L`, `10000000000000L`
`float`	Floating-point numbers. See the `float` type below.	`-3.14159`, `6.0235e23`
`complex`	Complex numbers. See the `complex` type below.	`(3.2+4.9j)`, `(0+3.42e-3j)`
`str`	Ordinary strings of ASCII characters. If you use three sets of quotes around a string, it can extend over multiple lines. See the `str` type below.	`'Fred'`, `""`, `'''...'''`, `"""..."""`
`unicode`	Strings of Unicode characters, an extended international character set. See the `unicode` type below.	`u'Fred'`, `u'\u03fa'`

[4] http://www.python.org/

Type name	Values	Examples
`list`	A sequence of values that is *mutable*, that is, its contained values can be changed, added, or deleted. See list type (p. 12) below.	`['red', 23]`, `[]`, `[(x,y) for x in range(10,30,5) for y in ("a", "b")]`
`tuple`	A tuple is a sequence of values that is *immutable* (cannot be modified). See tuple type (p. 14) below.	`('red', 23)`, `()`, `("singleton",)`
`None`	`None` is a special, unique value that may be used as a placeholder where a value is expected but there is no obvious value. For example, `None` is the value returned by functions if they don't return a specific value.	`None`

6. Numeric types

Python has four types to represent numbers: `int`, `long`, `float`, and `complex`.

6.1. Integers: `int` and `long` types

Values of `int` and `long` types represent whole numbers. The `int` type holds numbers in the range [-2,147,483,648, 2,147,483,647], while the precision of the `long` type is limited only by the available storage. Starting in version 2.2, Python will automatically switch to `long` type when it is needed to avoid overflow.

The *factory functions* used to convert other types to integer types are:

`int(`*`n_s`*`)`
If *`n_s`* is any kind of number, or string containing a number, this function converts it to type `int`.

`int(`*`s`*`, `*`r`*`)`
If *`s`* is a string representation of a number in base (radix) *`r`*, this function converts it to type `int`.

For example, `int("0f", 16)` returns 15; `int("101", 2)` returns 5.

`long(`*`n_s`*`)`
Converts a number or string to type `long`.

`long(`*`s`*`, `*`r`*`)`
Converts a string *`s`* representing a number in base *`r`* to type `long`.

6.2. The `float` type

Values of this type represent real numbers, with the usual limitations of IEEE floating point type: it cannot represent very large or very small numbers, and the precision is limited to only about 15 digits.

Functions:

`float(`*`n_s`*`)`
Converts a number or string *`n_s`* to a floating-point number.

For common transcendental functions like cosine and exponential, see the `math` module below.

6.3. The `complex` type

A complex number has two parts, a real component and and imaginary component.

To refer to the components of an imaginary number *I*:

`I.real`	The real component of *I*.
`I.imag`	The imaginary component.

Functions:

`complex(r[,i])`
The *r* argument is the real component, and the *i* second argument is the imaginary component. Each argument can be either a string or a number. For example, `complex(3.2,4.9)` produces the value `(3.2+4.9j)`.

6.4. Intrinsics for numbers

By *intrinsics* we mean built-in functions of Python. These functions operate on all the different kinds of numbers:

`abs(x)`
Returns the absolute value of *x*. For complex values, returns the magnitude.

`coerce(a,b)`
Returns a tuple `(a',b')` of the arguments coerced to the same type.

`divmod(a,b)`
Returns a tuple `(q, r)` where *q* is the quotient `a/b` and *r* is the remainder `a%b`.

`pow(x,y[,z])`
Computes *x* to the *y* power. If the optional third argument is given, it computes `(x**y)%z`.

`round(x)`
Returns *x* rounded up or down to the nearest integral float, e.g., `round(-3.5)` yields -4.

For common transcendental functions like square root and logarithm, see the `math` module below.

7. The sequence types

The next four types described (`str`, `unicode`, `list` and `tuple`) are collectively referred to as *sequence* types.

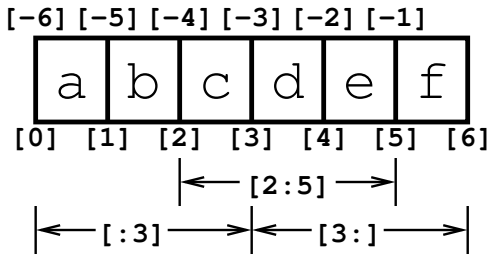
All of them represent an ordered set in the mathematical sense, that is, a collection of things in a specific order.

- A value of `str` (8-bit string) or `unicode` (Unicode string) type represents a sequence of zero or more characters. Python strings are immutable or atomic, that is, their contents cannot be modified in place.
- A `list` is a sequence of zero or more values of any type (or combination of types). Lists are mutable, meaning that you can make changes to them by deleting, inserting, or replacing elements.
- A `tuple` is also a sequence of zero or more values of any type, but it is immutable: once you have formed a tuple, you cannot delete, insert, or replace any of its elements.

7.1. Intrinsic functions common to all sequences

All four of the basic sequence types (`str`, `unicode`, `list`, and `tuple`) share a common set of functions. In the table below, *S* means any sequence:

`len(`*S*`)`	The number of elements in *S*.
`max(`*S*`)`	The largest element of *S*.
`min(`*S*`)`	The smallest element of *S*.
x `in` *S*	True if any members of *S* are equal to *x*. A predicate function, like the "`>=`" and other comparison operators..
x `not in` *S*	True if none of the elements of *S* are equal to *x*.
S_1+S_2	Concatenate two sequences.
`tuple(`*S*`)`	Convert the members of *S* to a tuple.
`list(`*S*`)`	Convert S to a list.
S`*`*n*	A new sequence containing *n* copies of the contents of *S*. For example, `[0]*5` is a list of five zeroes.
S`[`*i*`]`	Subscripting: Refers to element *i* of sequence *S*, counting from zero. For example, `"abcd"[2]` is `"c"`. A negative index value is counted from the end back toward the front. Element -1 refers to the last element, -2 to the next-to-last, and so on, so `"abcdef"[-1]` is `"f"`.
S`[`*i*`:`*j*`]`	Retrieve another sequence that is a *slice* of sequence *S* starting at element *i* (counting from zero) and going up to *but not including* element *j*. For example, `"abcdef"[2:4]` is `"cd"`. You can omit *i* to start the slice at the beginning of *s*; you can omit *j* to take a slice all the way to the end of *s*. For example, `"abcdef"[:3]` is `"abc"`, and `"abcdef"[3:]` is `"def"`. See the diagram below. You can even omit both parts; *S*`[:]` gives you a new sequence that is a copy of the old one.



In the diagram above, if string `s` is `"abcdef"`, slice `s[2:5]` is `"cde"`, slice `s[:3]` is `"abc"`, and `s[3:]` is `"def"`.

7.2. Strings: the `str` and `unicode` types

Python has two string types. Type `str` holds strings of zero or more 8-bit characters, while `unicode` strings provide full support of the expanded Unicode character set (see the Unicode homepage[5]).

[5] http://www.unicode.org/

7.2.1. String constants

There are many forms for string constants:

- `'...'`: Enclose the string in single quotes.
- `"..."`: Enclose it in double quotes.
- `'''...'''`: Enclose it between three single quotes in a row. The difference is that you can continue such a string over multiple lines, and the line breaks will be included in the string as newline characters.
- `"""..."""`: You can use three sets of double quotes. As with three sets of single quotes, line breaks are allowed and preserved as `"\n"` characters.

The above forms give you regular strings. To get a unicode string, prefix the string with `u`. For example:

```
u"klarn"
```

is a five-character Unicode string.

In addition, you can use any of these *escape sequences* inside a string constant:

`\`*`newline`*	A backslash at the end of a line is ignored.
`\\`	Backslash (\)
`\'`	Closing single quote (')
`\"`	Double-quote character (")
`\n`	Newline (ASCII LF or linefeed)
`\b`	Backspace (in ASCII, the BS character)
`\f`	Formfeed (ASCII FF)
`\r`	Carriage return (ASCII CR)
`\t`	Horizontal tab (ASCII HT)
`\v`	Vertical tab (ASCII VT)
`\`*`ooo`*	The character with octal code *ooo*, e.g., `'\177'`.
`\x`*`hh`*	The character with hexadecimal value *hh*, e.g., `"\xFF"`.
`\u`*`hhhh`*	The Unicode character with hexadecimal value *hhhh*, e.g., `u"\uFFFF"`.

Raw strings: If you need to use a lot of backslashes inside a string constant, and doubling them is too confusing, you can prefix any string with the letter `r` to suppress the interpretation of the escape sequences above. For example, `'\\\\'` contains two backslashes, but `r'\\\\'` contains four. Raw strings are particularly useful with the regular expression module (p. 41).

7.2.2. The string format operator

In addition to the operations common to all sequences, strings support the operator

```
f % v
```

Format values from a tuple *v* using a *format string f*; the result is a single string with all the values formatted. See the table of format codes below.

All format codes start with %; the other characters of *f* appear unchanged in the result. A conversational example:

```
>>> print "We have %d pallets of %s today." % (49, "kiwis")
We have 49 pallets of kiwis today.
```

In general, format codes have the form

```
%[p][m[.n]]c
```

where:

p	is an optional prefix; see the table of format code prefixes (p. 8) below.
m	specifies the total desired field width. The result will never be shorter than this value, but may be longer if the value doesn't fit; so, `"%5d" % 1234` yields `" 1234"`, but `"%2d" % 1234` yields `"1234"`.
n	specifies the number of digits after the decimal point for float types.
c	indicates the type of formatting.

Here are the format codes *c*:

`%s`	String; e.g., `"%-3s" % "xy"` yields `"xy "` (because the `"-"` prefix forces left alignment).
`%d`	Decimal conversion, e.g., `"%3d" % -4` yields the string `" -4"`.
`%e`	Exponential format; allow four characters for the exponent. Examples: `"%08.1e" % 1.9783` yields `"0002.0e+00"`.
`%E`	Same as `%e`, but an uppercase `E` is used for the exponent.
`%f`	For float type. E.g., `"%4.1f" % 1.9783` yields `" 2.0"`.
`%g`	General numeric format. Use `%f` if it fits, otherwise use `%e`.
`%G`	Same as `%G`, but an uppercase `E` is used for the exponent if there is one.
`%o`	Octal, e.g., `"%o" % 13` yields `"15"`.
`%x`	Hexadecimal, e.g., `"%x" % 247` yields `"f7"`.
`%X`	Same as `%x`, but capital letters are used for the digits A-F, e.g., `"%04X" % 247` yields `"00F7"`.
`%c`	Convert an integer to the corresponding ASCII code; for example, `"%c" % 0x61` yields the string `"a"`.
`%%`	Places a percent sign (%) in the result. Does not require a corresponding value.

Format prefixes include:

`+`	For numeric types, forces the sign to appear even for positive values.
`-`	Left-justifies the value in the field.
`0`	For numeric types, use zero fill. For example, `"%04d" % 2` produces the value "0002".
`#`	With the `%o` (octal) format, append a leading `"0"`; with the `%x` (hexadecimal) format, append a leading `"0x"`; with the `%g` (general numeric) format, append all trailing zeroes. Examples:

```
>>> "%4o" % 127
' 177'
>>> "%#4o" % 127
'0177'
>>> "%x" % 127
```

```
'7f'
>>> "%#x" % 127
'0x7f'
>>> "%10.5g" % 0.5
'       0.5'
>>> "%#10.5g" % 0.5
'   0.50000'
```

7.2.3. String formatting from a dictionary

You can use the string format operator % to format a set of values from a dictionary *D*:

```
f % D
```

In this form, the general form for a format code is:

```
%(k)[p][m[.n]]c
```

where *k* is a key in dictionary *D*, and the rest of the format code is as in the usual string format operator. For each format code, the value of *D*[*k*] is used.

For example, suppose D is the dictionary `{'baz':39, 'foo':'X'}`; then `("=%(foo)s=%(baz)03d=" % D)` yields `'=X=039='`.

7.2.4. String functions

Functions:

str(*obj*)
Converts *obj*, an object of any type, to a string. For example, `str(17)` produces the string `'17'`.

unicode(*s*[,*enc*[,*errs*]])
Converts an object *s*, of any type, to a Unicode string. The optional *enc* argument specifies an encoding, and the optional *errs* argument specifies what to do in case of errors (see the *Python Library Reference*[6] for details).

raw_input(*p*)
Prompt for input with string *p*, then return a line entered by the user, without the newline. *p* may be omitted for unprompted input.

7.2.5. String methods

These methods are available on any string or Unicode object *S*:

***S*.capitalize()**
Return *S* with its first character capitalized.

***S*.center(*w*)**
Return *S* centered in a string of width *w*, padded with spaces. If *w*<=`len(`*S*`)`, the result is a copy of *S*. Example: `'x'.center(4)` returns `' x  '`.

[6] http://www.python.org/doc/current/lib/lib.html

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.

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*.

S.expandtabs([tabsize])
Returns a copy of *S* with all tabs expanded to spaces using. The optional *tabsize* argument specifies the number of spaces between tab stops; the default is 8.

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*. For example, `"banana".find("an")` returns `1`. The optional *start* and *end* arguments restrict the search to slice *S*[*start*:*end*].

S.index(t[,start[,end]])
Works like `.find()`, but if *t* is not found, it raises a `ValueError` exception.

S.isalnum()
Predicate that tests whether *S* is nonempty and all its characters are alphanumeric.

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

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

S.islower()
Predicate that tests whether *S* is nonempty and all its characters are lowercase letters.

S.isspace()
Predicate that tests whether *S* is nonempty and all its characters are whitespace characters.

In Python, the characters considered whitespace include `' '` (space, called SP in ASCII), `'\n'` (newline, NL), `'\r'` (return, CR), `'\t'` (tab, HT), `'\f'` (form feed, FF), and `'\v'` (vertical tab, VT).

S.isupper()
Predicate that tests whether *S* is nonempty and all its characters are uppercase letters.

S.join(L)
L must be a sequence. Returns a string containing the members of the sequence with copies of string *S* inserted between them. For example, `'/'.join(['foo', 'bar', 'baz'])` returns the string `'foo/bar/baz'`.

S.ljust(w)
Return a copy of *S* left-justified in a field of width *w*, padded with spaces. If *w*<=`len(`*S*`)`, the result is a copy of *S*. Example: `"Ni".ljust(4)` returns `"Ni  "`.

S.lower()
Returns a copy of S with all uppercase letters replaced by their lowercase equivalent.

S.lstrip([c])
Return *S* with all leading characters from string *c* removed. The default value for *c* is a string containing all the whitespace characters (p. 10).

S.replace(old,new[,max])
Return a copy of *S* with all occurrences of string *old* replaced by string *new*. Normally, all occurrences are replaced; if you want to limit the number of replacements, pass that limit as the *max* argument.

S.rfind(t[,start[,end]])
Like `.find()`, but if *t* occurs in *S*, this method returns the *highest* starting index.

For example, `"banana".rfind("an")` returns 3.

S.rjust(w)
Return a copy of *S* right-justified in a field of width `w`, padded with spaces. If *w*`<=len(`*S*`)`, the result is a copy of *S*.

S.rstrip([c])
Return *S* with all trailing characters from string *c* removed. The default value for *c* is a string containing all the whitespace characters (p. 10).

S.split([d[,max]])
Returns a list of strings [s_0, s_1, ...] made by splitting *S* into pieces wherever the delimiter string *d* is found. The default is to split up *S* into pieces wherever clumps of one or more whitespace (p. 10) characters are found. Some examples:

```
>>> "I'd annex \t \r the Sudetenland" .split()
["I'd", 'annex', 'the', 'Sudetenland']
>>> '3/crunchy frog/ Bath & Wells'.split('/')
['3', 'crunchy frog', ' Bath & Wells']
>>> '//Norwegian Blue/'.split('/')
['', '', 'Norwegian Blue', '']
>>> 'never<*>pay<*>plan<*>'.split('<*>')
['never', 'pay', 'plan', '']
```

The optional *max* argument limits the number of pieces removed from the front of *S*. For example, `'a/b/c/d/e'.split('/',2)` yields the list `['a', 'b', 'c/d/e']`.

S.splitlines([keepends])
Splits *S* into lines and returns a list of the lines as strings. Discards the line separators unless the optional *keepends* arguments is true.

S.startswith(t[,start[,end]])

Predicate to test whether *S* starts with string *t*. Otherwise similar to `.endswith()`.

S.strip([c])
Return *S* with all leading and trailing characters from string *c* removed. The default value for *c* is a string containing all the whitespace characters (p. 10).

S.swapcase()
Return a copy of S with each lowercase character replaced by its uppercase equivalent, and vice versa.

S.translate(new[,drop])
This function is used to translate or remove each character of *S*. The *new* argument is a string of exactly 256 characters, and each character *x* of the result is replaced by `new[ord(`*x*`)]`. If you would like certain characters removed from *S* before the translation, provide a string of those characters as the *drop* argument.

***S*.upper()**
Return a copy of *S* with all lowercase characters replaced by their uppercase equivalents.

***S*.zfill(*w*)**
Return a copy of *S* left-filled with `'0'` characters to width *w*. For example, `'12'.zfill(5)` returns `'00012'`.

7.3. The `list` type

Python's list type represents a sequence of objects of any type. Lists support all the operations described above under intrinsics for sequences (p. 6), in addition to the operations described below.

7.3.1. Constructing a list

To create a list, place zero or more objects between a pair of square brackets, separated by commas. Examples:

```
[]                    # An empty list.
["baked beans"]       # List containing one item.
[23, 30.9, "x"]       # List containing three items.
```

7.3.2. List comprehensions

In versions 2.2 and beyond, you can also use a *list comprehension* to create a list. The general form is:

```
[ e for v1 in s1
      for v2 in s2 ...
        ...
          if c ]
```

where *e* is some expression, followed by zero or more `for` clauses, optionally followed by an `if` clause. The equivalent in pre-2.2 Python would be:

```
temp = []
for v1 in s1:
    for v2 in s2 ...:
        ...
        if c:
            temp.append ( e )
```

The value of the expression is the final value of `temp`.

A few examples of list comprehensions, taken from actual evaluations in Python calculator mode:

```
>>> [(n,a) for n in range(3) for a in ('a', 'b')]
[(0, 'a'), (0, 'b'), (1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
>>> [(i,j) for i in range(1,4) for j in range(1,4) if i != j]
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
>>> [10*x+y for x in range(1,4) for y in range(x)]
[10, 20, 21, 30, 31, 32]
```

7.3.3. Slice assigment

Because lists are mutable, lists with slice operators can appear as destinations in an `assign` statement. The expression being assigned must be a sequence, and the elements of that slice of the list are replaced by the elements of that sequence. For example:

```
>>> L=[0,1,2,3,4,5,6,7,8]
>>> L[4:6]
[4, 5]
>>> L[4:6] = [17, 18, 19, 20]
>>> L
[0, 1, 2, 3, 17, 18, 19, 20, 6, 7, 8]
>>>
```

7.3.4. List functions

`list(`*`i`*`)`
Creates a list, a sequence of values that is mutable (its contained values can be changed, added, or deleted). The value *`i`* is a sequence of values or any object that can be iterated over.

`range([`*`start`*`,]`*`limit`*`[,`*`step`*`])`
Creates a list containing a sequence of integers. There are three forms:

- `range(`*`limit`*`)` creates the list `[0, 1, .., `*`limit`*`-1]`. Note that the list stops just before element *`limit`*. For example, `range(3)` returns the list `[0, 1, 2]`.
- `range(`*`start`*`,`*`limit`*`)` creates the list `[`*`start`*`, `*`start`*`+1, ..., `*`limit`*`-1]`.
- `range(`*`start`*`,`*`limit`*`,`*`step`*`)` returns the list `[`*`start`*`,`*`start`*`+`*`step`*`,`*`start`*`+2*`*`step`*`,...]`, stopping just before the element that would equal or exceed *`limit`* in the direction of the step.

  For example, `range(4,10,2)` returns `[4, 6, 8]`, and `range(3,-1,-1)` returns `[3, 2, 1, 0]`.

`xrange([`*`start`*`,]`*`limit`*`[,`*`step`*`])`
Effectively produces the same values as the `range()` function, but doesn't take up all the space, which is nice for iterating over huge sequences.

7.3.5. List methods

All list objects have these methods, where *`L`* is any list:

***`L`*`.append(`*`x`*`)`**
Append a new element *`x`* to the end of list *`L`*. Does not return a value.

***`L`*`.pop([`*`i`*`])`**
Remove and return element *`i`* from *`L`*. The default value for *`i`* is -1, so if you pass no argument, the last element is removed.

***`L`*`.count(`*`x`*`)`**
Return the number of elements of *`L`* that compare equal to *`x`*.

***`L`*`.index(`*`x`*`)`**
If *`L`* contains any elements that equal *`x`*, return the index of the first such element, otherwise raise a `ValueError` exception.

`L.insert(i,x)`
Insert a new element *x* into list *L* just before the *i*th element, shifting all higher-number elements to the right. No value is returned. Example: if L is `[0,1,2]`, then `L.insert(1,99)` changes L to `[0,99,1,2]`.

`L.remove(x)`
Remove the first element of *L* that is equal to *x*. If there aren't any such elements, raises `ValueError`.

`L.reverse()`
Reverses the elements of *L in place; does* not *return a result.*

`L.sort()`
Sort list *L in place* using the default rules for comparing objects. *Does not return a result.*

`L.sort(f)`
Sort list *L in place* using function *f* to compare pairs of objects to see in which order they should go. The calling sequence and result of this function should be the same as for the `cmp()` function.

For example, here is a function that can be used to sort numbers and strings in descending order, demonstrated in calculator mode:

```
>>> def descendingCmp(a,b):
...     return  - cmp(a,b)
...
>>> L=[0,39,18,99,40]
>>> L.sort()
>>> L
[0, 18, 39, 40, 99]
>>> L.sort(descendingCmp)
>>> L
[99, 40, 39, 18, 0]
>>>
```

7.4. The `tuple` type

A tuple, like a list, represents a sequence of zero or more objects of any type. Unlike mutable lists, though, tuples cannot be modified in place; that is, they are immutable.

To create a tuple, place zero or more values between parentheses with commas between them. If there is only one value, place a comma after it. The examples shown below are tuples containing four, one, and zero values respectively:

```
("red", 255, 0, 0)
(10,)
()
```

All the operations described in intrinsics for sequences apply to tuples.

When to use tuples:

- Whenever you want to form a sequence whose contents will not change.
- Tuples are required by Python in some places, such as when using the string format operator.
- Creating a tuple is faster and uses less memory than creating a list of the same size.

8. The dictionary type

Python dictionaries are one of its more powerful built-in types. They are generally used for look-up tables and many similar applications.

A Python dictionary represents a set of zero or more ordered pairs (k_i, v_i) such that:

- Each k_i value is called a *key*;
- each key is unique and immutable; and
- the associated *value* v_i can be of any type.

Another term for this structure is *mapping*, since it maps the set of keys onto the set of values.

8.1. Operations on dictionaries

These operations are available on any dictionary object *D*:

`len(`*D*`)`	Returns the number of entries in *D*.
D`[`*k*`]`	If *D* has an entry with key *k*, returns the corresponding value. If there is no such key, raises a `KeyError` exception.
D`[`*k*`] = `*v*	Sets the entry for key *k* to value *v*. If there was previously a value for that key, it is replaced.
`del` *D*`[`*k*`]`	If there is an entry for key *k*, that entry is deleted. If not, a `KeyError` exception is raised.
D`.has_key(`*k*`)`	Predicate that returns true if *D* has a key *k*.
D`.items()`	A list of (*key*,*value*) tuples from *D*.
`.keys()`	A list of all keys in *D*.
D`.values()`	A list of all values in *D*, in the same order as *D*`.keys()`.
D`.update(`*E*`)`	Merge dictionary *E* into *D*. If the same key exists in both, use the value from *E*.
D`.get(`*k*`,`*x*`)`	Same as *D*`[`*k*`]`, but if no entry exists for key *k*, returns *x*, or `None` if *x* is omitted.
D`.setdefault(`*k*`[,`*x*`])`	If *D*`[`*k*`]` exists, returns that value. If there is no element *D*`[`*k*`]`, sets *D*`[`*k*`]` to *x*, defaulting to `None`, and returns that value.
D`.iteritems()`	Returns an iterator over the (*key*, *value*) pairs of *D*.
D`.iterkeys()`	Returns an iterator over the keys of *D*.
D`.itervalues()`	Returns an iterator over the values of *D*.
D`.popitem()`	Returns an arbitrary entry from *D* as a (*key*, *value*) tuple, and also removes that entry. If *D* is empty, raises a `KeyError` exception.
x `in` *D*	True if *x* is a key in *D*.
x `not in` *D*	True if *x* is not a key in *D*.

9. Input and output: File objects

To open a file:

```
f = open(name[,mode[,bufsize]])
```

This function returns a file object. The *name* is the file's pathname. The *bufsize* argument, if given, is used to specify the buffer size (but this is not usually necessary). The *mode* argument is a string with this syntax:

```
"access[+][b]"
```

where *access* is `r` for read access (the default), `w` for write access, and `a` for append. The `+` flag specifies update access, and `b` forces binary mode.

Methods defined on file objects include:

f.read()
Read the entire file *f* and return it as a string.

f.read(n)
Read the next *n* characters from file *f*. If the file is exhausted, it returns an empty string (""); if fewer than *n* characters remain, you get all of them.

f.readline()
Returns the next line from *f*, including its line terminator, if any. Returns an empty string when the file is exhausted.

f.readlines()
Read all the lines from file *f* and return them as a list of strings, including line termination characters.

f.write(s)
Write string *s* to file *f*.

f.writelines(L)
Write a list of strings *L* to file *f*.

f.seek(p[,w])
Change the file position. The value of *w* determines how *p* is used:

- 0: Set the position to *p*; this is the default.
- 1: Move the position forward *p* bytes from its current position.
- 2: Set the position to *p* bytes before the end of file.

f.truncate([p])
Remove any contents of the file past position *p*, which defaults to the current position.

f.tell()
Returns the current file position.

f.flush()
Flush the buffer, completing all transactions against *f*.

f.isatty()
Predicate: Is this file a terminal?

f.close()
Close file *f*.

10. Expressions

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

Table 1. Python 2.2 operator precedences

`(E)`	Parenthesized expression or tuple.
`[E, ...]`	List.
`{key:value, ...}`	Dictionary.
`` `...` ``	Convert to string representation.
`x.attribute`	Attribute reference.
`x[...]`	Subscript or slice; see above under sequence types.
`f(...)`	Call function *f*.
`x**y`	*x* to the *y* power.
`-x`	Negation.
`~x`	Bitwise not (one's complement).
`x*y`	Multiplication.
`x/y`	Division.
`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 below).
`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."

10.1. What is a predicate?

We use the term *predicate* to mean any Python function that tests some condition and returns a true or false value. For example, `x < y` is a predicate that tests whether `x` is less than `y`. Predicates generally return 1 for true and 0 for false. For example, `5 < 500` returns 1, while `5 >= 500` returns 0.

11. Built-in functions for multiple types

Here are some useful functions that operate on objects of many types:

chr(*n*)
Returns a string containing the character whose ASCII code is the integer *n*.

cmp(*x*,*y*)
Compare two objects. The value returned is zero if *x* and *y* are equal; negative if *x* precedes *y*; and positive if *x* follows *y*.

This may not work for all types, or when *x* and *y* have different types. It will certainly work for numbers and strings.

dir(*x*)
Returns a list of all the names defined in *x*. If *x* is a module, the list shows the names of its members and functions. If the argument is omitted, it returns a list of the names defined in the local name space.

filter(*f*,*S*)
S must be a sequence, and *f* a function that takes one argument and returns true or false (that is, a predicate). The result is a new sequence consisting of only those elements of *S* for which *f*(*S*) is true. If *f* is `None`, the result is a sequence consisting of the true elements of *S*; see the definition of true-false values below.

globals()
Returns a dictionary whose keys are the names defined in the current module's global scope, and whose values are the values bound to those names.

locals()
Returns a dictionary whose keys are the names defined in the current local scope, and whose values are the values bound to those names.

map(*f*,s_0[,s_1,...])
Applies a function *f* to each member of a sequence s_0 and returns a list of the results [*f*(s_0[0]), *f*(s_0[1]), ...]. If there are additional sequence arguments, the function must take as many arguments as there are sequences, and the [i^{th}] element of the result is equal to *f*(s_0[*i*],s_1[*i*], ...).

ord(*c*)
The argument *c* must be a string containing exactly one character. The return value is the character code of that character as an integer.

reduce(*f*,*s*[,*i*])
Apply the two-argument function *f* pairwise to the elements of sequence *s*. Starts by computing t_0=*f*(*s*[0],*s*[1]), then t_1=*f*(t_0,*s*[1]), and so forth over the elements of *s*, then returns the last t_i.

The optional third argument *i* is a starting value. If it is given, the function starts by computing t_0=*f*(*i*,*s*[0]), and then continues as before.

repr(*x*)
Return a string containing a printable representation of *x*. You can abbreviate this function as `` `x` ``.

type(*x*)
Returns the type of *x*, or more precisely, the *type object* for the type of x.

There is one type object for each built-in type; the type objects are named `int`, `long`, and so forth, and are summarized in the table of basic types.

You can compare type objects using expressions like "`if type(x) is list...`".

`unichr(`*`n`*`)`
Returns a Unicode string containing the character whose code is the integer *n*.

12. Python statements

Here is a complete list of all the Python statement types. In each case but one, the keyword shown in the first column appears at the beginning of the statement. The exception is the Python assignment statement: any line containing "`=`" is considered an assignment statement. A single "`=`" is *not an operator* in Python; it cannot be embedded in an expression as in some other languages such as C.

`=`	Assignment. See Section 13.1, "Assignment" (p. 20).
`assert`	Assertion facility. See Section 13.2, "The `assert` statement" (p. 20).
`break`	Break out of a loop. See Section 14.5, "The `break` statement" (p. 24).
`class`	Define a new class of objects. See Section 17, "Create your own types: The `class` construct" (p. 28).
`continue`	Go to the top of a loop. See Section 14.6, "The `continue` statement" (p. 24).
`def`	Define a function. See Section 16, "Defining and calling functions" (p. 27).
`del`	Delete a variable or part of an object. See Section 13.3, "The `del` statement" (p. 21).
`elif`	See Section 14.2, "The `if` construct: choice" (p. 23).
`else`	See Section 14.2, "The `if` construct: choice" (p. 23).
`except`	See Section 14.7, "The `try` construct: catching exceptions" (p. 24).
`exec`	Dynamically execute Python code. See Section 13.4, "The `exec` statement" (p. 21).
`finally`	See Section 14.7, "The `try` construct: catching exceptions" (p. 24).
`for`	Iterate over the members of a sequence. See Section 14.3, "The `for` construct: iteration" (p. 24).
`from`	Import names from a module. See Section 13.5, "The `from` statement" (p. 21).
`global`	Use a global variable. See Section 13.6, "The `global` statement" (p. 22).
`if`	Conditional branching. See Section 14.2, "The `if` construct: choice" (p. 23).
`import`	Import an external module. See Section 13.7, "The `import` statement" (p. 22).
`pass`	A placeholder, no-operation statement. See Section 13.8, "The `pass` statement" (p. 22).
`print`	Display values to output. See Section 13.9, "The `print` statement" (p. 22).
`raise`	Raise an exception. See Section 14.8, "The `raise` statement: throwing exceptions" (p. 25).
`return`	See Section 16.2, "`return`: Return the result of a function" (p. 28).
`try`	Anticipate possible exceptions. See Section 14.7, "The `try` construct: catching exceptions" (p. 24).
`while`	Repeat a group of statements. See Section 14.4, "The `while` construct: looping" (p. 24).
`yield`	Return one of a sequence of values from a generator. See Section 18.2, "Generators" (p. 33).

13. Simple statements

Simple (non-branching) statement types are summarized below.

13.1. Assignment

Assignment is a statement, not an operator, in Python. The general form is:

```
L0=L1=...=E
```

This statement first evaluates some expression *E*, and then assigns that value to one or more *destinations* L_0, L_1, and so on. Each destination can be either of:

- A variable name. The variable is *bound* to the value of the expression, that is, that variable now has that value, and any previous value it may have had is forgotten. For example, the statement

  ```
  beanCount=m=0
  ```

  sets variables `beanCount` and `m` to the integer value 0.

- Part of a mutable object that contains multiple values. For example, if `L` is a list, the statement

  ```
  L[0]='xyz'
  ```

  would set the first element of `L` to the string `'xyz'`.

  As another example, suppose D is a dictionary. The statement

  ```
  D["color"]="red"
  ```

  would associate key `"color"` with value `"red"` in that dictionary.

  You can also assign to slices of a list:

  ```
  L[1:3] = [10, 20, 30]
  ```

  This statement would delete elements 1 and 2 of list L and replace them with three new elements 10, 20, and 30.

  Finally, if an object `X` has an attribute `X.klarn`, you can assign a value of 73 to it using:

  ```
  X.klarn = 73
  ```

- If the expression *E* is a sequence, the destination can be a list of variables, and the sequence is *unpacked* into the variables in order. For example, this statement

  ```
  x, y, z  =  [10, 11, 12]
  ```

  sets `x` to 10, `y` to 11, and `z` to 12.

13.2. The `assert` statement

To check for "shouldn't happen" errors, you can use an `assert` statement:

```
assert e1
assert e1, e2
```

where e_1 is some condition that should be true. If the condition fails, an `AssertionError` exception is raised (see exceptions below). If a second expression e_2 is provided, the value of that expression is passed with the exception.

Assertion checking can be disabled by running Python with the `-O` (optimize) option.

13.3. The `del` statement

The purpose of the `del` statement is to delete things. The general form is:

```
del L0, L1, ...
```

where each L_i is a destination (described above under the assignment statement). You can delete:

- A variable. For example, the statement

  ```
  del i, j
  ```

  causes variables `i` and `j` to become *unbound*, that is, undefined.

- An element or slice of a list. For example,

  ```
  del L[5], M[-2:]
  ```

  would delete the sixth element of list `L` and the last two elements of list `M`.

- One entry in a dictionary. For example, if `D` is a dictionary,

  ```
  del D["color"]
  ```

  would delete from `D` the entry for key `"color"`.

13.4. The exec statement

To dynamically execute Python code, use this statement:

```
exec E0 [in E1 [, E2]]
```

Expression E_0 specifies what to execute, and may be a string containing Python source code, an open file, or a code object. If E_1 is omitted, the code is executed in the local scope. If E_1 is given but E_2 is not, E_1 is a dictionary used to define the names in the global and local scopes. If E_2 is given, E_1 is a dictionary defining the global scope, and E_2 is a dictionary defining the local scope.

13.5. The `from` statement

The purpose of the `from` statement is to use things (variables, functions, and so on) from *modules*. There are two general forms:

```
from M import *
from M import n0, n1, ...
```

where M is the name of a Python module. In the first form, all objects from that module are added to the local name space. In the second form, only the named objects n_0, n_1, ... are added.

There are two types of modules:

- Python has a number of built-in modules; see commonly used modules (p. 37) below.
- You can also create your own modules by simply placing the definitions of variables and functions in a file whose name has the form *f*.py, and then using a statement like this:

```
from f import *
```

Compare this statement with the `import` statement below.

13.6. The `global` statement

To declare that you want to access global variables, use a statement of the form

```
global v0, v1, ...
```

This is not actually necessary unless you are assigning a value to the variable before using it, which normally leads Python to conclude that the variable is local.

13.7. The `import` statement

Like the `from` statement above, the `import` statement gives your program access to objects from external modules. The general form is:

```
import M0, M1, ...
```

where each M_i is the name of a module.

However, unlike the `from` statement, the `import` statement does not add the objects to your local name space. Instead, it adds the name of the module to your local name space, and you can refer to items in that module using the syntax *M*.*name* where *M* is the name of the module and *name* is the name of the object.

For example, there is a standard module named `math` that contains a variable named `pi` and a function named `sqrt()`. If you import it using

```
import math
```

then you can refer to the variable as `math.pi` and the function as `math.sqrt()`.

If instead you did this:

```
from math import *
```

then you could refer to them simply as `pi` and `sqrt()`.

13.8. The `pass` statement

This statement does nothing. It is used as a placeholder where a statement is expected. Its syntax:

```
pass
```

13.9. The `print` statement

To print the values of one or more expressions e_0, e_1, ..., use a statement of this form:

```
print e0, e1, ...
```

The values are converted to strings if necessary and then printed with one space between values.

Normally, a newline is printed after the last value. However, you can suppress this behavior by appending a comma to the end of the list. For example, this statement

```
print "State your name:",
```

would print the string followed by one space and leave the cursor at the end of that line.

14. Compound statements

For branching statements, *B* is a block of all the statements that are indented further, up to but not including the next statement that is not indented further; blank lines don't count.

You can put the block *B* on the same line as the branching statement, after the colon (`:`). Multiple statements can be so placed, with semicolons (`;`) separating them.

14.1. Boolean values: true and false

In the sections below on the "`if`" and "`while`" statements, we speak of values being true or false. For Python's purposes, the following values are considered false:

- Any number equal to zero: `0`, `0.0`, `0L`, `0j`.
- Any empty sequence (such as the empty list "`[]`" or the empty tuple "`()`").
- An empty mapping, such as the empty dictionary "`{}`".
- The special value `None`.

Any other value is considered true.

14.2. The `if` construct: choice

For conditional branching:

```
if E0:
    B0
elif E1:
    B1
elif ...
else:
    Bf
```

This is the most general form, and means: if expression E_0 is true, execute block B_0; otherwise, if E_1 is true, execute block B_1; and so forth. If all the conditions are false, execute B_f.

The `elif` clause is optional, and there can be any number of them. The `else` clause is also optional.

When evaluating whether an expression is true or false, false values include `None`, a number equal to zero, an empty sequence, or an empty dictionary. All other values are true.

Here's an example:

```
if x < 0:
    print "But it's negative!"
else:
    print "The square root of", x, "is", x**0.5
```

14.3. The `for` construct: iteration

To iterate over a section of the program:

```
for L in E:
        B
```

This statement executes a block of code once for each member of a sequence *E*. For each member, the destination *L* is set to that member, and then the block *B* is executed.

The destination *L* follows the same rules for destinations as in the assignment statement.

14.4. The `while` construct: looping

To execute a section of the program repeatedly:

```
while E:
    B_t
else:
    B_f
```

The `else:` part is optional. Here's how the statement is executed:

1. Evaluate the expression *E*. If that expression is false, go to Step 3 (p. 24). Otherwise go to Step 2 (p. 24).
2. Execute block B_t, and then return to Step 1 (p. 24).
3. If there is an `else:` block B_f, execute it, otherwise do nothing.

14.5. The `break` statement

To exit a `for` or `while` loop, use this statement:

```
break
```

14.6. The `continue` statement

Within a `for` or `while` statement, you can jump back to the top of the loop and start the next iteration with this statement:

```
continue
```

14.7. The `try` construct: catching exceptions

Python has a wonderfully flexible and general error-handling mechanism using *exceptions*. Whenever a program cannot proceed normally due to some problem, it can *raise* an exception. For example, there

is a built-in exception called `ZeroDivisionError` that occurs whenever someone tries to divide by zero. You can also define your own kinds of exceptions. See exceptions (p. 26) below for more information about exception types.

Typically, an exception will terminate execution of the program. However, you can use Python's `try` construct to *handle* exceptions, that is, take some other action when they occur.

The most general form of the construct looks like this:

```
try:
    Bt
except E1, L1:
    B1
except E2, L2:
    B2
  ...
else:
    Bf
```

This construct specifies how you want to handle one or more exceptions that may occur during the execution of block B_t. Each `except` clause specifies one kind of exception you want to handle; the E_i part specifies which exception or exceptions, and the L_i parts are destinations that receive data about the exception when it occurs.

- You can omit the L_i part.
- You can omit both the L_i and the E_i parts. In that case, the `except` clause matches all types of exceptions.

Here is how a `try` block works:

1. If no exception is raised during the execution of block B_t, the `else` block B_f is executed if there is one.
2. If an exception is raised during block B_t, and it matches some exception type E_i, the corresponding block B_i is executed.
3. If B_t raises an exception that doesn't match any of the `except` clauses, program execution is terminated and a message shows the exception and a traceback of the program.

The built-in exceptions are arranged in a hierarchy structure of base classes and classes derived from them. An exception X matches an `except:` clause E_i if they are the same exception, or if X is a subclass of E_i. See the discussion of the class structure of exceptions below.

There is another form of `try` block that is used to force execution of a cleanup block B_c no matter whether or not another block B_t causes an exception:

```
try:
    Bt
finally:
    Bc
```

If B_t raises any exception, block B_c is executed, then the same exception is raised again.

14.8. The `raise` statement: throwing exceptions

To raise an exception:

```
raise e, p
```

where *e* is the exception to be raised and *p* is a value to be passed back to any `try` block that may handle this exception. You may omit *p*, in which case a value of `None` is returned.

15. Exceptions

Python's exception mechanism allows you to signal errors in a flexible, general way, and also to handle errors. See the `try` and `raise` statements above.

A variety of exceptions are built into Python, and they are arranged in a hierarchy so that you can handle either specific individual exceptions or larger groupings of them.

Here is the current exception hierarchy. Indentation shows the inheritance: for example, all exceptions inherit from `Exception`; `ArithmeticError` inherits from `StandardError`; and so on.

- `Exception`: Base class for all exceptions.
  - `SystemExit`: Normal program termination. This exception is not considered an error.
  - `StopIteration`: This is a special exception to be used with iterators and generators; see iterators (p. 32) below.
  - `StandardError`: Base class for all exceptions that are considered errors.
    - `ArithmeticError`: Errors from numeric operations.
      - `FloatingPointError`: Failure in a floating point operation.
      - `OverflowError`: A numeric operation that led to a number too large to be represented.
      - `ZeroDivisionError`: Attempt to divide by zero.
    - `LookupError`: Base class for erroneous in retrieving items from a sequence or mapping.
      - `IndexError`: Trying to retrieve an element of a sequence when the index is out of range.
      - `KeyError`: Trying to retrieve an item from a mapping that does not contain the given key.
    - `EnvironmentError`: Base class for external errors. For the exceptions in this class, the associated value is an object with the error number in its `.errno` attribute and the error message in its `.strerror` attribute.
      - `IOError`: Input/output error.
      - `OSError`: Operating system error.
    - `AssertionError`: An `assert` statement has failed.
    - `AttributeError`: Attempt to retrieve or set a nonexistent attribute of an object.
    - `ImportError`: Attempt to import a nonexistent module, or a nonexistent name from a module.
    - `KeyboardInterrupt`: Someone pressed interrupt (e.g., control-C under Unix.
    - `MemoryError`: No more memory is available.
    - `NameError`: Reference to an undefined name.
    - `NotImplementedError`: Use this exception for functions that aren't written yet. It's also recommended for virtual methods in base classes, those intended to be overridden.
    - `SyntaxError`: Attempt to import or execute syntactically invalid Python source code.

- `SystemError`: An internal error in Python.
- `TypeError`: Attempt to use an operation that isn't defined for objects of the type it's to operate on.
- `ValueError`: Attempt to operate on an inappropriate value.

16. Defining and calling functions

To define a function named *n*:

```
def n(p0[=e0][,p1[=e1]]...[,*pv][,**pd]):
    B
```

Function *n* is defined as block *B*. A function can have any number of parameters p_1, p_2, ...

If there is an expression e_i for parameter p_i, that expression specifies the default value for that parameter used if the caller does not supply a value. Such parameters are called *keyword parameters*, and all keyword parameters must follow all positional (non-keyword) parameters.

If there is a $*p_v$ parameter, that parameter gets a (possibly empty) list of all extra positional arguments passed to the function.

If there is a $**p_d$ parameter, that parameter gets a dictionary of all extra keyword arguments passed to the function.

To sum up, a function's parameters must start with zero or more positional parameters, followed by zero or more keyword parameters, followed optionally by a parameter to receive extra positional arguments, followed optionally by a parameter to receive extra keyword arguments.

16.1. Calling a function

The arguments you supply to a function must satisfy these rules:

- You must supply all positional arguments.
- If you supply additional arguments beyond the positional arguments and the function has keyword arguments, the additional arguments are matched to those keyword parameters by position. Any unmatched keywoard parameters assume their default values.
- You can 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.
- If the function is declared with a $**p_d$ parameter, you can supply keyword arguments that don't match the keywords of the parameters. Those extras will be packaged as a dictionary with the keywords as the keys and their values as the values.

For example, suppose a function is defined with this parameter list:

```
def foo(a, b, c=9, d="blue", *e, **f): ...
```

This function has two positional parameters `a` and `b` and two keyword parameters `c` and `d`. Callers must supply at least two positional arguments. If a third or fourth positional argument is supplied, they are bound to parameters `c` and `d` respectively; additional positional arguments are bound in a list to `e`.

If the caller supplies keyword arguments of the form `c=`*`value`* or `d=`*`value`*, those values are bound to parameters `c` and `d` respectively. Any other keyword arguments are packaged as a dictionary and bound to parameter `f`.

16.2. `return`: Return the result of a function

To return the result value from a function, execute a statement of this form:

```
return e
```

where *e* is any expression.

You can omit the expression. In this case, the special value `None` is returned to the caller.

If execution reaches the end of a function without reaching a `return` statement, the effect is the same as

```
return None
```

17. Create your own types: The `class` construct

You can define new types, or *classes*, with Python's `class` declaration. Some definitions:

class
A type (built-in or user-defined).

object
One value of a given type. As the number `23` is an object of type `int`, so an object in general is one instance of a user-defined type. It is useful to think of a class a cookie cutter (that is, a pattern), and an object as a cookie that follows that pattern. Like individual cookies, individual objects all start out looking the same, but inside them are values that may change.

The type or class defines the behaviors common to all objects of the same type. For example, the unary `-` operator works on all values of type `float`, and changes the sign of the value.

instance
Same as "object."

member
One of the values inside an instance. For example, a complex number `C` has two `float` values inside it: the real pattern is referred to as `C.real` and the imaginary part as `C.imag`. Ultimately all the values inside an instance are the atomic types listed in the table of basic types, or other objects that themselves contain atomic types, and so on.

method
A function that operates primarily on an object. Unlike regular Python functions, which are called with the function name (such as "`sqrt(x)`"), to call a method you use the syntax:

```
O.m(...)
```

where *O* is an object, *m* is the method name, and you supply zero or more arguments between parentheses.

To define a new class of objects:

```
class C:
    B_i
    def n_0 ( self, ... ):
        B_0
    def n_1 ( self, ... ):
```

```
        B1
    ...
```

This construct defines a new class *C*.

- Block B_i is executed once when the class is first scanned, and can be used to define class variables and methods.
- The functions n_i define the methods of the class. Note that these `def` statements must be indented below the `class` declaration.
- When you are defining methods, you must always include `self` as the first argument. When you call a method, you omit that first argument. For example, if a method is defined as:

  ```
  class Antenna:
      def rotate(self, degrees):
          ...
  ```

  then, if you had an instance of this class named `hill`, you might call the method as

  ```
  hill.rotate(40.3)
  ```

  Inside a method, the name `self` is used to refer to the class's members. For example, inside the `.rotate()` method, you might refer to member `height` as `self.height`.

To define a derived class that inherits from superclasses P_0, P_1, ...:

```
class C(P0,P1,...):
```

You can inherit from built-in classes such as `int` and `dict`. See below under the class constructor (p. 29) for more details.

You can use a number of special method names to define certain behaviors of your objects. All these names start and end with two underbars (__). The most commonly used one is `__init__()`, the class constructor, but there are many more to define how your objects act when operated on by various operators.

17.1. `__init__()`: The class constructor

The *constructor* is a method that is executed to create a new object of the class. Its name is always `__init__()`.

Its parameter list must always include `self` as the first argument; any remaining parameters must be provided by the caller. For example, if your constructor has four parameters including `self`, a caller must provide three arguments.

You can think of `self` as representing the name space inside the instance. To add new members to an instance, just assign a value to `self.`*v* where *v* is the member name.

Here's a complete example of a class `FeetInches` that represents dimensions in feet and inches. Internally the dimension is represented as only inches, and that value is stored in a member named `.__inches` (member names starting with two underbars, like this, one, are *private* members, and hence accessible only from inside the object).

The constructor takes two arguments named `feet` and `inches` and converts them to the private member `.__inches`. Then we define a method called `.show()` that converts the internal dimension

to a string of the form *f* `ft` *i* `in` where *f* is the whole feet and *i* is the inches part. Here's the class definition:

```
class FeetInches:
    def __init__ ( self, feet, inches ):
        self.__inches  =  (feet * 12.0) + inches

    def show(self):
        feet  =  int ( self.__inches / 12.0 )
        inches  =  self.__inches - feet * 12.0
        return "%d ft %.2f in" % (feet, inches)
```

To create an object of this type we might say:

```
fiveFootFour = FeetInches(5, 4)
```

The method call `fiveFootFour.show()` would then yield the string `"5 ft 4.00 in"`.

17.2. Special method names

Below are most of the commonly used special methods you can define in your class. Refer to the *Python Reference Manual* for a complete list.

`.__abs__(self)`
Defines the behavior of the `abs()` function when applied to an object of your class.

`.__add__(self,`*`other`*`)`
Defines the behavior of this class for `self+`*`other`*.

`.__and__(self,`*`other`*`)`
Defines the behavior of `self&`*`other`*.

`.__cmp__(self,`*`other`*`)`
Compares two values, `self` and *`other`*, to see which way they should be ordered. Return a negative integer if `self` should come first; return a positive integer if the *`other`* value should come first; and return an integer `0` if they are considered equal.

`.__complex__(self)`
Defines the behavior of the builtin function `complex()`, to convert `self` to a complex value.

`.__contains__(self,`*`x`*`)`
You can use this special method to define how the "`in`" and "`not in`" operators work to test whether an object *x* is a member of `self`. The method returns true if *x* is in `self`, false otherwise.

`.__del__(self)`
Destructor: this method is called when the instance is about to be destroyed because there are no more references to it.

`.__delitem__(self,`*`key`*`)`
Called to delete `self[`*`key`*`]`. Raise `IndexError` if the *`key`* is not valid.

`.__div__(self,`*`other`*`)`
Defines the behavior of `self/`*`other`*.

`.__divmod__(self,`*`other`*`)`
Defines the behavior of `divmod(self,`*`other`*`).`

.__float__(self)
Defines the behavior of the builtin `float()` function; should return a float value.

.__getattr__(self,*name*)
Get `self`'s attribute whose name is the string *name*. If the name is not a legal attribute name, this method should raise `AttributeError`. This method is called only if the object doesn't have a regular attribute by the given name.

.__getitem__(self,*key*)
Called on access to `self[`*key*`]`. Raise `IndexError` if the *key* is not valid.

.__int__(self)
Defines the behavior of the builtin `int()` function; should return a value of type `int`.

.__invert__(self)
Defines the behavior of the unary ~ operator.

.__iter__(self)
If `self` is a container object, return an iterator that iterates over all the items in that object. See iterators (p. 32) below. For mapping objects, the iterator should iterate over all the keys.

.__hex__(self)
Defines the behavior of the builtin `hex()` function; should return a string of hexadecimal characters.

.__len__(self)
Return the length of `self` as a nonnegative integer.

.__long__(self)
Defines the behavior of the builtin `long()` function.

.__lshift__(self,*other*)
Defines the behavior of `self<<`*other*.

.__mod__(self,*other*)
Defines the behavior of `self%`*other*.

.__mul__(self,*other*)
Defines the behavior of `self*`*other*.

.__neg__(self)
Implements the unary negate operator `-self`.

.__nonzero__(self)
Called to test an object's truth value. Return 1 for true, 0 for false. If you don't define this method, Python tries computing the length with the `.__len__()` method; zero length is considered false and all other values true.

.__oct__(self)
Defines the behavior of the builtin `oct()` function; should return a string of octal characters.

.__or__(self,*other*)
Defines the behavior of `self|`*other*.

.__pow__(self,*other*[,*modulo*])
Defines the behavior of the exponentiation operator `self**`*other*. It also implements the built-in function `pow()`, which takes an optional third argument *modulo*; see that function for details about the third argument..

.__rshift__(self,*other*)
Defines the behavior of `self>>`*other*.

.__setattr__(self,*name*,*value*)
Called for assignments of the form

```
0.name = value
```

If your method needs to store a value in the instance, don't just assign a value, because that will force `.__setattr__()` to be called again, creating an infinite loop. Instead, use an assignment of the form

```
0.__dict__[name] = value
```

.__setitem__(self,*key*,*value*)
Called for assignments to `self[key]`.

.__str__(self)
Returns self in the form of a string. The built-in function `str()` and the `print` statement use this method if it is defined.

.__sub__(self,*other*)
Defines the behavior of `self-other`.

.__xor__(self,*other*)
Defines the behavior of `self^other`.

17.3. Intrinsics for objects

These functions can apply to classes, instances, and modules.

delattr(*0*,*n*)
Delete the attribute whose name is *n* from object *0*.

getattr(*0*,*n*)
Get the attribute whose name is *n* from object *0*.

hasattr(*0*,*n*)
Does object *0* have an attribute named *n*?

setattr(*0*,*n*,*v*)
Set the attribute of object *0* named *n* to value *v*.

18. Recent features

The Python language is in active development. Sections below describe some major new features since version 2.0.

18.1. Iterators

New features in Python 2.2 generalize the process of visiting the elements of a sequence.

These features all affect the way `for` statements work. In the general case, a `for` statement looks like this:

```
for v in S:
    B
```

where the block *B* is executed once for each element of the sequence *S*, with some destination *v* set to each element of *S* in turn.

An iterator is a new concept that generalizes the sequence *S* so that you can use objects other than regular sequence types like lists and tuples. An iterator is an object that knows how to visit a sequence of values.

The way `for` statements actually work now is that it calls the built-function `iter(`*S*`)` to convert the sequence into an iterator that knows how to visit the elements of *S*. Calling `iter(`*S*`)` where *S* is a list or tuple returns an iterator that visits each element of the list or tuple in turn, with index values 0, 1, 2, ...

Any object with the special `.__getitem__()` method can be used as the sequence in a `for` statement. Values 0, 1, ... are used for the index, and the `for` statement terminates when `.__getitem__()` raises `IndexError`.

Objects may also have an `.__iter__()` method, which supersedes the `.__getitem__()` method in `for` statements. This method implements the `iter()` function for the class, and returns an iterator for the instance.

An iterator must be an object with a `.next()` method that takes no arguments and returns the next element in sequence. This method should raise the special exception `StopIteration` to signify that there are no more elements.

18.2. Generators

Generators are a completely new type of iterator that allows a function to do lazy evaluation of a sequence, that is, to produce the values of the sequence one by one on demand.

Normally, when a function it is called, it runs until it either provides a value with a `return` statement, or falls off the bottom of the function block. But starting in Python 2.2, it is possible for a function to return a value and *also* suspend its entire internal state for later resumption. This state includes the current point of execution inside the function, and all its local variables. Such a function is called a generator.

A generator *G* can be used in a `for` statement:

```
for v in G():
    B
```

In this context, every time the generator *G* returns a value, *v* is set to that value, and the block *B* is executed.

The beauty of this construct is that function *G* doesn't have to compute all its values at once. It can do lazy evaluation, producing new values only as they are needed by the caller.

To write your own generator, you must have this special `import` statement to enable the feature:

```
from __future__ import generators
```

In your function definition, use the `yield` statement to generate the next value:

```
yield E
```

Executing this statement returns the value of expression *E*, but it also saves the state of the function to allow later resumption.

Here's an example. This function generates the even numbers from 0 up to the argument you give it:

```
def evens(n):
    i = 0
    while 2*i <= n:
        yield 2*i
        i = i + 1
```

Here's how that function would work in a `for` statement:

```
>>>for v in evens(30):
...    print v,
...
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30
```

18.3. Static methods

Sometimes you want to associate a function with a particular class, but it doesn't always have an instance (`self`) to operate on. In these cases, you can declare a *static method* within the class. Such a method is like any other method, except that you won't include a first argument of `self` in the argument list. To make a method static, include a line of this form just after the method's declaration:

```
f = staticmethod ( f )
```

where *f* is the name of the method.

To call a static method, use this syntax:

```
C.f(...)
```

where *C* is the class name and *f* is the method name.

Here's an example. Suppose you have a class called `Celsius` that represents a temperature, and you want to be able to convert an object of that class to a 6-character string for display. Normally you would declare a `.__str__()` method in the class to do the conversion.

However, suppose you have a variable named `outdoor` that may contain a `Celsius` object, but sometimes the temperature is unknown and `outdoor` is set to `None`. You want to write a function that will convert the temperature if it is known, but render the value as `'******'` if not. You could write a standalone function that does that, but it really is associated with the class. So you write a static function like this:

```
class Celsius:
    ...
    def show(temp):
        if  temp is None:
            return '******'
        else:
            return str(temp)
    show  =  staticmethod(show)
```

Then you could call this method as "`Celsius.show(outdoor)`".

18.4. Class methods

A similar new feature is the *class method*. As with static methods, a class method is not passed the instance as its first (`self`) argument. However, a class method *is* always passed a first argument containing the class object itself. You declare a method as a class method by following its declaration with a line of the form

```
f = classmethod ( f )
```

Here's an example:

```
class SpecialClass:
    ...
    def g(co, z):
        ...
    g  =  classmethod(g)
```

If called as `SpecialClass.g(4)`, within method `g` `z` would be bound to the integer `4` and `co` would be bound to the class object `SpecialClass`.

19. The Python debugger

The Python debugger allows you to monitor and control execution of a Python program, to examine the values of variables during execution, and to examine the state of a program after abnormal termination (post-mortem analysis).

19.1. Starting up pdb

To execute any Python statement under the control of `pdb`:

```
>>> import pdb
>>> import yourModule
>>> pdb.run('yourModule.test()')  # Or any other statement
```

where `yourModule.py` contains the source code you want to debug.

To debug a Python script named `myscript.py`:

```
python /usr/lib/python2.2/pdb.py myscript.py
```

To perform post-mortem analysis:

```
>>> import pdb
>>> import yourModule
>>> yourModule.test()
[crash traceback appears here]
>>> pdb.pm()
(pdb)
```

Then you can type debugger commands at the `(pdb)` prompt.

19.2. Functions exported by pdb

The `pdb` module exports these functions:

pdb.run(*stmt*[,*globals*[,*locals*]])
Executes any Python statement. You must provide the statement *as a string*. The debugger prompts you before beginning execution. You can provide your own global name space by providing a *globals* argument; this should be a dictionary mapping global names to values. Similarly, you can provide a local namespace by passing a dictionary *locals*.

pdb.runeval(*expr*[,*globals*[,*locals*]])
This is similar to the `pdb run` command, but it evaluates an expression, rather than a statement. The *expr* is any Python expression in string form.

pdb.runcall(*func*[,*arg*]...)
Calls a function under `pdb` control. The *func* must be either a function or a method. Arguments after the first argument are passed to your function.

19.3. Commands available in pdb

The debugger prompts with a line like this:

```
(pdb)
```

At this prompt, you can type any of the `pdb` commands discussed below. You can abbreviate any command by omitting the characters in square brackets. For example, the `where` command can be abbreviated as simply `w`.

h[elp] [*cmd*]
Without an argument, prints a list of valid commands. Use the *cmd* argument to get help on command *cmd*.

!*stmt*
Execute a Python statement *stmt*. The "`!`" may be omitted if the statement does not resemble a `pdb` command.

w[here]
Shows your current location in the program as a stack traceback, with an arrow (`->`) pointing to the current stack frame.

q[uit]
Exit `pdb`.

l[ist] [*begin*[,*end*]]
Shows you Python source code. With no arguments, it shows 11 lines centered around the current point of execution. The line about to be executed is marked with an arrow (`->`), and the letter `B` appears at the beginning of lines with breakpoints set.

To look at a given range of source lines, use the *begin* argument to list 11 lines around that line number, or provide the ending line number as an `end` argument. For example, the command `list 50,65` would list lines 50-65.

(empty line)
If you press Enter at the `(pdb)` prompt, the previous command is repeated. The `list` command is an exception: an empty line entered after a `list` command shows you the next 11 lines after the ones previously listed.

b[reak] [[*filename*:]*lineno*[,*condition*]]b[reak] [*function*[,*condition*]]
The `break` command sets a breakpoint at some location in your program. If execution reaches a breakpoint, execution will be suspended and you will get back to the `(pdb)` prompt.

The first form sets a breakpoint at a specific line in a source file. Specify the line number within your source file as *lineno*; add the *filename*: if you are working with multiple source files, or if your source file hasn't been loaded yet.

The second form sets a breakpoint on the first executable statement of the given *function*.

You can also specify a conditional breakpoint, that is, one that interrupts execution only if a given *condition* evaluates as true. For example, the command `break 92,i>5` would break at line 92 only when `i` is greater than 5.

When you set a breakpoint, `pdb` prints a "breakpoint number." You will need to know this number to clear the breakpoint.

tbreak
Same options and behavior as `break`, but the breakpoint is temporary, that is, it is removed after the first time it is hit.

cl[ear] [*lineno*]
If used without an argument, clears all breakpoints. To clear one breakpoint, give its breakpoint number (see `break` above).

s[tep]
Single step: execute the current line. If any functions are called in the current line, `pdb` will break upon entering the function.

n[ext]
Like `step`, but does not stop on entering a called function.

c[ont[inue]]
Resume execution until the next breakpoint (if any).

r[eturn]
Resume execution until the current function returns.

a[rgs]
Display the argument names and values to the currently executing function.

p *expr*
Evaluate an expression *expr* and print its value.

20. Commonly used modules

The sections below discuss only a tiny fraction of the official and unofficial module library of Python. For a full set, find the online and printed versions of the *Python Library Reference*[7] *Library Reference* at the Python documentation site[8].

If you want to use any of these modules, you must import them. See the `import` statement and the `from` statement above.

20.1. The `math` module

This module provides the usual basic transcendental mathematical functions. All trig functions use angles in radians. The module has two members:

[7] http://www.python.org/doc/current/lib/lib.html
[8] http://www.python.org/doc/

`pi`	The constant 3.14159...
`e`	The base of natural logarithms, 2.1828...

Functions in this module include:

`acos(x)`	Angle (in radians) whose cosine is *x*, that is, arccosine of *x*.
`asin(x)`	Arcsine of *x*.
`atan(x)`	Arctangent of *x*.
`atan2(y,x)`	Angle whose slope is *y/x*, even if *y* is zero.
`ceil(x)`	True ceiling function; `ceil(3.9)` yields 4.0, while `ceil(-3.9)` yields -3.0.
`cos(x)`	Cosine of *x*, where *x* is expressed in radians.
`cosh(x)`	Hyperbolic cosine of *x*.
`exp(x)`	`e` to the *x* power.
`floor(x)`	True floor function; `floor(3.9)` is 3.0, and `floor(-3.9)` is -4.0.
`frexp(x)`	Given a float *x*, returns a tuple `(m,e)` where *m* is the mantissa and *e* the binary exponent of *x*, and *x* is equal to `m * (2**e)`. If *x* is zero, returns the tuple `(0.0, 0)`.
`fmod(x,y)`	`(x-int(x/y)*y)`
`ldexp(x, i)`	Returns `x * (2**i)`.
`hypot(x,y)`	The square root of (x^2+y^2).
`log(x)`	Natural log of *x*.
`log10(x)`	Common log (base 10) of *x*.
`modf(x)`	Returns a tuple `(f, i)` where *f* is the fractional part of *x*, *i* is the integral part (as a float), and both have the same sign as *x*.
`sin(x)`	Sine of *x*.
`sinh(x)`	Hyperbolic sine of *x*.
`sqrt(x)`	Square root of *x*.
`tan(x)`	Tangent of *x*.
`tanh(x)`	Hyperbolic tangent of *x*.

20.2. The `cmath` module: complex math

For computations involving complex numbers, the `cmath` module includes all the functions of the `math` except for `atan2`, `ceil`, `floor`, `fmod`, `frexp`, `hypot`, `ldexp`, `modf`, and `pow`.

In addition, it provides these functions:

`acosh(x)`	Hyperbolic arc cosine of *x*.
`asinh(x)`	Hyperbolic arc sine of *x*.
`atanh(x)`	Hyperbolic arc tangent of *x*.

20.3. The types module

This module contains the *type objects* for all the different Python types. For any two objects of the same type or class, applying the `type()` function to them returns the same object:

```
>>> import types
>>> type(2.5)
<type 'float'>
>>> type(2.5) is types.FloatType
1
```

Here are most of the members of the `types` module. (A few more obscure types exist; see the *Python Library Reference*[9] if you are working with the more exotic features of the language.)

`BuiltinFunctionType`	Built-in function such as `dir()`.
`BuiltinMethodType`	Built-in method such as the `.append()` method on a list object.
`ClassType`	User-defined class.
`ComplexType`	Complex number.
`DictType`	Dictionary.
`FileType`	File object.
`FloatType`	Floating point number.
`FunctionType`	User-defined function.
`GeneratorType`	Generator (see generators above).
`InstanceType`	Instance of a user-defined class.
`IntType`	Integer.
`ListType`	List.
`LongType`	Long integer.
`MethodType`	Method of a user-defined class.
`NoneType`	The unique object `None`.
`StringType`	String.
`TupleType`	Tuple.
`TypeType`	Type object.
`UnicodeType`	Unicode string.
`XRangeType`	Result returned by `xrange()`.

In Python 2.2 and beyond, the name of the type's constructor function is the type object. This works for `int`, `long`, `float`, `str` (strings), `unicode`, `tuple`, `list`, and `dict` (dictionaries). For example:

```
>>>  type(2.5) is float
1
```

[9] http://www.python.org/doc/current/lib/lib.html

20.4. The `string` module

This module contains additional features for manipulating strings.

20.4.1. Variables in the `string` module

Variables defined in the `string` module include:

`digits`
The string `"0123456789"`.

`lowercase`
A string containing all the lowercase letters, `"abcdefghijklmnopqrstuvwxyz"`.

`uppercase`
A string containing all the uppercase letters, `"ABCDEFGHIJKLMNOPQRSTUVWXYZ"`.

`letters`
The concatenation of `lowercase` and `uppercase`.

`punctuation`
String of characters that are considered punctuation marks: `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`

`whitespace`
Characters considered whitespace, generally `" \t\n\r\f\v"`.

`printable`
String containing all the printable characters, the union of `letters`, `digits`, `punctuation`, and `whitespace`.

`octdigits`
The string `"01234567"`.

`hexdigits`
`"0123456789abcdefABCDEF"`.

20.4.2. Functions in the `string` module

In addition to the variables and functions in this module, there are a large number of functions that duplicate the built-in methods on strings. For example, instead of

```
s.split(",", 1)
```

you can get the same function with

```
import string
string.split(s, ",", 1)
```

In general, for any built-in method

```
s.m(arg1,arg2,...),
```

you can import the `string` module and get the same effect with

```
string.m(s,arg1,arg2,...),
```

Additional functions include:

maketrans(s,t)

Builds a translation table to be used as the first argument to the *S*.translate() string method. The arguments *s* and *t* are two strings of the same length; the result is a translation table that will convert each character of *s* to the corresponding character of *t*.

Here's an example:

```
>>> import string
>>> bingo=string.maketrans("lLrR", "rRlL")
>>> "Cornwall Llanfair".translate(bingo)
'Colnwarr Rranfail'
```

join(*L*[,*d*])

L must be a sequence. Returns a string containing the members of the sequence with copies of string *d* inserted between them. The default value of *d* is one space. For example, string.join(['baked', 'beans, 'are', 'off']) returns the string 'baked beans are off'.

20.5. Regular expression matching with the re module

The re module provides functions for matching strings against regular expressions. See the O'Reilly book *Mastering Regular Expressions* by Friedl and Oram for the whys and hows of regular expressions. We discuss only the commonest functions here. Refer to the *Python Library Reference*[10] for the full feature set.

Note: The raw string notation r'...' is most useful for regular expressions; see raw strings (p. 7), above.

These characters have special meanings in regular expressions:

.	Matches any character except a newline.
^	Matches the start of the string.
$	Matches the end of the string.
*r**	Matches zero or more repetitions of regular expression *r*.
r+	Matches one or more repetitions of *r*.
r?	Matches zero or one *r*.
*r**?	Non-greedy form of *r**; matches as few characters as possible. The normal * operator is greedy: it matches as much text as possible.
r+?	Non-greedy form of *r*+.
r??	Non-greedy form of *r*?.
r{*m*,*n*}	Matches from *m* to *n* repetitions of *r*. For example, r'x{3,5}' matches between three and five copies of letter 'x'; r'0{4}' matches the string '0000'.
r{*m*,*n*}?	Non-greedy version of the previous form.
[...]	Matches one character from a set of characters. You can put all the allowable characters inside the brackets, or use *a*-*b* to mean all characters from *a* to *b* inclusive. For example, regular expression r'[abc]' will match either 'a', 'b', or 'c'. Pattern r'[0-9a-zA-Z]' will match any single letter or digit.

[10] http://www.python.org/doc/current/lib/lib.html

`[^...]`	Matches any character *not* in the given set.
rs	Matches expression *r* followed by expression *s*.
r|*s*	Matches either *r* or *s*.
(*r*)	Matches *r* and forms it into a group that can be retrieved separately after a match; see `MatchObject`, below. Groups are numbered starting from 1.
(?:*r*)	Matches *r* but does not form a group for later retrieval.
(?P<*n*>*r*)	Matches *r* and forms it into a named group, with name *n*, for later retrieval.

These special sequences are recognized:

n	Matches the same text as a group that matched earlier, where *n* is the number of that group. For example, `r'([a-zA-Z]+):\1'` matches the string `"foo:foo"`.
\A	Matches only at the start of the string.
\b	Matches the empty string but only at the start or end of a word (where a word is set off by whitespace or a non-alphanumeric character). For example, `r'foo\b'` would match `"foo"` but not `"foot"`.
\B	Matches the empty string when *not* at the start or end of a word.
\d	Matches any digit.
\D	Matches any non-digit.
\s	Matches any whitespace character (p. 10).
\S	Matches any non-whitespace character.
\w	Matches any alphanumeric character.
\W	Matches any non-alphanumeric character.
\Z	Matches only at the end of the string.
\\	Matches a backslash (\) character.

There are two ways to use a `re` regular expression. Assuming you import the module with `import re`, you can test whether a regular expression *r* matches a string *s* with the construct `re.match(`*r*`,`*s*`)`.

However, if you will be matching the same regular expression many times, the performance will be better if you compile the regular expression using `re.compile(`*r*`)`, which returns a compiled regular expression object. You can then check a string s for matching by using the `.match(`*s*`)` method on that object.

Here are the functions in module `re`:

`compile(`*r*`[,`*f*`])`
Compile regular expression *r*. Returns a compiled r.e. object; see the table of methods on such objects below. To get case-insensitive matching, use `re.I` as the *f* argument. There are other flags that may be passed to the *f* argument; see the *Python Library Reference*[11].

`match(`*r*`,`*s*`[,`*f*`])`
If *r* matches the start of string *s*, return a `MatchObject` (see below), otherwise return `None`.

`search(`*r*`,`*s*`[,`*f*`])`
Like the `match()` method, but matches *r* anywhere in *s*, not just at the beginning.

[11] http://www.python.org/doc/current/lib/lib.html

split(*r*,*s*[,maxsplit=*m*])
Splits string *s* into pieces where pattern *r* occurs. If *r* does not contain groups, returns a list of the parts of *s* that match *r*, in order. If *r* contains groups, returns a list containing all the characters from *s*, with parts matching *r* in separate elements from the non-matching parts. If the *m* argument is given, it specifies the maximum number of pieces that will be split, and the leftovers will be returned as an extra string at the end of the list.

sub(*r*,*R*,*s*[,count=*c*])
Replace the leftmost non-overlapping parts of *s* that match *r* using *R*; returns *s* if there is no match. The *R* argument can be a string or a function that takes one `MatchObject` argument and returns the string to be substituted. If the *c* argument is supplied (defaulting to 0), no more than *c* replacements are done, where a value of 0 means do them all.

20.5.1. Compiled regular expression objects

Compiled regular expression objects returned by `re.compile()` have these methods:

.match(*s*[,[p_s][,p_e]])
If the start of string *s* matches, return a `MatchObject`; if there is no match, return `None`. If p_s is given, it specifies the index within *s* where matching is to start; this defaults to 0. If p_e is given, it specifies the maximum length of *s* that can be used in matching.

.search(*s*[,[p_s][,p_e]])
Like `match()`, but matches anywhere in *s*.

.split(*s*[,maxsplit=*m*])
Like `re.split()`.

.sub(*R*,*s*[,count=*c*])
Like `re.sub()`.

.pattern
The string from which this object was compiled.

20.5.2. Methods on a `MatchObject`

A `MatchObject` is the object returned by `.match()` or other methods. Such an object has these methods:

.group([*n*])
Retrieves the text that matched. If there are no arguments, returns the entire string that matched. To retrieve just the text that matched the *n*th group, pass in an integer *n*, where the groups are numbered starting at 1. For example, for a `MatchObject` *m*, `m.group(2)` would return the text that matched the second group, or `None` if there were no second group.

If you have named the groups in your regular expression using a construct of the form `(?P<`*name*`>...)`, the *n* argument can be the *name* as a string. For example, if you have a group `(?P<year>[\d]{4})` (which matches four digits), you can retrieve that field using *m*`.group("year")`.

.groups()
Return a tuple $(s_1, s_2, \ldots)$ containing all the matched strings, where s_i is the string that matched the *i*th group.

.start([*n*])
Returns the location where a match started. If no argument is given, returns the index within the string where the entire match started. If an argument *n* is given, returns the index of the start of the match for the *n*th group.

.end([*n*])
Returns the location where a match ended. If no argument is given, returns the index of the first character past the match. If *n* is given, returns the index of the first character past where the *n*th group matched.

.span([*n*])
Returns a 2-tuple (*m*.start(*n*),*m*.end(*n*)).

.pos
The effective p_s value passed to .match() or .search().

.endpos
The effective p_e value passed to .match() or .search().

.re
The regular expression object used to produce this MatchObject.

.string
The *s* argument passed to .match() or .search().

20.6. The sys module

The services in this module give you access to command line arguments, standard input and output streams, and other system-related facilities.

argv
sys.argv[0] is the name of your Python script, or "-c" if in interactive mode. The remaining elements, sys.argv[1:], are the command line arguments, if any.

builtin_module_names
A list of the names of the modules compiled into your installation of Python.

exit(*n*)
Terminate execution with status *n*.

modules
A dictionary of the modules already loaded.

path
The search path for modules, a list of strings in search order.

Note: You can modify this list. For example, if you want Python to search directory /u/dora/python/lib for modules to import before searching any other directory, these two lines will do it:

```
import sys
sys.path.insert(0, "/u/dora/python/lib")
```

platform
A string identifying the software architecture.

stdin
The standard input stream as a file object.

stdout
The standard output stream as a file object.

stderr
The standard error stream as a file object.

20.7. The `random` module: random number generation

randrange([*start*,]*stop*[,*step*])
Return a random element from the sequence `range(`*start*`,`*stop*`,`*step*`)` .

randint(*x*,*y*)
Returns a random integer in the closed interval [*x*,*y*]; that is, any result r will satisfy *x* `<=` *r* `<=` *y*.

choice(*L*)
Returns a randomly selected element from a sequence *L*.

shuffle(*L*)
Randomly permute the elements of a sequence *L*.

random()
Returns a random float in the half-open interval [0.0, 1.0); that is, for any result *r*, `0.0 <=` *r* `<` `1.0`.

uniform(*x*,*y*)
Returns a random float in the half-open interval [*x*,*y*).

normalrand(*m*,*s*)
Generate a normally distributed pseudorandom number with mean *m* and standard deviation *s*.

An assortment of other pseudorandom distributions is available: Beta, circular uniform, exponential, gamma, Gaussian, log normal, Von Mises, Pareto, and Weibull distributions. See the *Python Library Reference*[12] for details.

20.8. The `time` module: dates and times

- *Epoch time* is the time in seconds since some arbitrary starting point. For example, Unix measures time in seconds since January 1, 1970.
- *UTC* is Coordinated Universal Time, the time on the zero meridian (which goes through London).
- *DST* refers to Daylight Savings Time.

A *time tuple* is a 9-tuple *T* with these elements, all integers:

T`[0]`	Four-digit year.	*T*`[5]`	Second, in [0,59].
T`[1]`	Month, 1 for January, 12 for December.	*T*`[6]`	Day of week, 0 for Monday, 6 for Sunday.
T`[2]`	Day of month, in [1,31].	*T*`[7]`	Ordinal day of the year, in [1,366].
T`[3]`	Hour, in [0,23].	*T*`[8]`	DST flag: `1` if the time is DST, `0` if it is not DST, and `-1` if unknown.
T`[4]`	Minute, in [0,59].		

[12] http://www.python.org/doc/current/lib/lib.html

Contents of the `time` module:

`altzone`
The local DST offset, in seconds west of UTC (negative for east of UTC).

`asctime([T])`
For a time-tuple *T*, returns a string of exactly 24 characters with this format:

```
"Thu Jun 12 15:25:31 1997"
```

The default time is now.

`clock()`
The accumulated CPU time of the current process in seconds, as a float.

`ctime([E])`
Converts an epoch time *E* to a string with the same format as `asctime()`. The default time is now.

`daylight`
Nonzero if there is a DST value defined locally.

`gmtime([E])`
Returns the time-tuple corresponding to UTC at epoch time *E*; the DST flag will be zero. The default time is now.

`localtime([E])`
Returns the time-tuple corresponding to local time at epoch time *E*. The default time is now.

`mktime(T)`
Converts a time-tuple *T* to epoch time as a float, where *T* is the *local* time.

`sleep(s)`
Suspend execution of the current process for *s* seconds, where *s* can be a float or integer.

`strftime(f[,t])`
Time formatting function; formats a time-tuple *t* according to format string *f*. The default time *t* is now. As with the string format operator, format codes start with %, and other text appears unchanged in the result. See the table of codes below.

`time()`
The current epoch time, as a float.

`timezone`
The local non-DST time zone as an offset in seconds west of UTC (negative for east of UTC). This value applies when daylight savings time is not in effect.

`tzname`
A 2-tuple (*s*, *d*) where *s* is the name of the non-DST time zone locally and *d* is the name of the local DST time zone. For example, in Socorro, NM, you get `('MST', 'MDT')`.

Format codes for the `strftime` function include:

%a	Abbreviated weekday name, e.g., "Tue".
%A	Full weekday name, e.g., "Tuesday".
%b	Abbreviated month name, e.g., "Jul".
%B	Full month name, e.g., "July".
%d	Day of the month, two digits with left zero fill; e.g. "03".

%H	Hour on the 24-hour clock, two digits with zero fill.
%I	Hour on the 12-hour clock, two digits with zero fill.
%j	Day of the year as a decimal number, three digits with zero fill, e.g. "366".
%m	Decimal month, two digits with zero fill.
%M	Minute, two digits with zero fill.
%p	Either "AM" or "PM". Midnight is considered AM and noon PM.
%S	Second, two digits with zero fill.
%w	Numeric weekday: 0 for Sunday, 6 for Saturday.
%y	Two-digit year. Not recommended!
%Y	Four-digit year.
%Z	If there is a time zone, a string representing that zone; e.g., "PST".
%%	Outputs the character %.

20.9. The os module: operating system interface

The variables and methods in the os module allow you to interact with files and directories. In most cases the names and functionalities are the same as the equivalent C language functions, so refer to Kernighan and Ritchie, *The C Programming Language*, second edition, or the equivalent for more details.

environ
A dictionary whose keys are the names of all currently defined environmental variables, and whose values are the values of those variables.

error
The exception raised for errors in this module.

chdir(*p*)
Change the current working directory to that given by string *p*

chmod(*p*,*m*)
Change the permissions for pathname *p* to *m*. See module stat, below, for symbolic constants to be used in making up *m* values.

chown(*p*,*u*,*g*)
Change the owner of pathname *p* to user id *u* and group id *g*.

execv(*p*,*A*)
Replace the current process with a new process executing the file at pathname *p*, where *A* is a list of the strings to be passed to the new process as command line arguments.

execve(*p*,*A*,*E*)
Like execv(), but you supply a dictionary *E* that defines the environmental variables for the new process.

_exit(*n*)
Exit the current process and return status code *n*. This method should be used only by the child process after a fork(); normally you should use sys.exit().

fork()
Fork a child process. In the child process, this function returns 0; in the parent, it returns the child's process ID.

getcwd()
Returns the current working directory name as a string.

getegid()
Returns the effective group ID.

geteuid()
Returns the effective user ID.

getgid()
Returns the current process's group ID.

getpid()
Returns the current process's process ID.

getppid()
Returns the parent process's PID (process ID).

getuid()
Returns the current process's user ID.

kill(*p*,*s*)
Send signal *s* to the process whose process ID is *p*.

link(*s*,*d*)
Create a hard link to *s* and call the link *d*.

listdir(*p*)
Return a list of the names of the files in the directory whose pathname is *p*. This list will never contain the special entries `"."` and `".."` for the current and parent directories. The entries may not be in any particular order.

lstat(*p*)
Like `stat()`, but if *p* is a link, you will get the status tuple for the link itself, rather than the file it points at.

mkfifo(*p*,*m*)
Create a named pipe with name *p* and open mode *m*. The server side of the pipe should open it for reading, and the client side for writing. This function does not actually open the fifo, it just creates the rendezvous point.

mkdir(*p*[,*m*])
Create a directory at pathname *p*. You may optionally specify permissions *m*; see module `stat` below for the interpretation of permission values.

nice(*i*)
Renice (change the priority) of the current process by adding *i* to its current priority.

readlink(*p*)
If *p* is the pathname to a soft (symbolic) link, this function returns the pathname to which that link points.

remove(*p*)
Removes the file with pathname *p*, as in the Unix `rm` command. Raises `OSError` if it fails.

rename(p_o, p_n)
Rename path p_o to p_n.

rmdir(*p*)
Remove the directory at path *p*.

stat(*p*)
Return a status tuple describing the file or directory at pathname *p*. See module `stat`, below, for the interpretation of a status tuple. If *p* is a link, you will get the status tuple of the file to which *p* is linked.

symlink(*s*,*d*)
Create a symbolic link to path *s*, and call the link *d*.

system(*c*)
Execute the command in string *c* as a sub-shell. Returns the exit status of the process.

times()
Returns a tuple of statistics about the current process's elapsed time. This tuple has the form `(`*u*`,`*s*`,`*u*`',`*s*`',`*r*`)` where *u* is user time, *s* is system time, *u'* and *s'* are user and system time including all child processes, and *r* is elapsed real time. All values are in seconds as floats.

umask(*m*)
Sets the "*umask*" that determines the default permissions for newly created files. Returns the previous value. Each bit set in the umask corresponds to a permission that is *not* granted by default.

uname()
Returns a tuple of strings descriping the operating system's version: `(s,n,r,v,m)` where *s* is the name of the operating system, *n* is the name of the processor (node) where you are running, *r* is the operating system's version number, *v* is the major version, and *m* describes the type of processor.

utime(*p*,*t*)
The *t* argument must be a tuple `(`*a*`, `*m*`)` where *a* and *m* are epoch times. For pathname *p*, set the last access time to *a* and the last modification to *m*.

wait()
Wait for the termination of a child process. Returns a tuple `(`*p*`,`*e*`)` where *p* is the child's process ID and *e* is its exit status.

waitpid(*p*,*o*)
Like `wait()`, but it waits for the process whose ID is *p*. The option value *o* specifies what to do if the child is still running. If *o* is 0, you wait for the child to terminate. Use a value of `os.WNOHANG` if you don't want to wait.

WNOHANG
See `waitpid()` above.

20.10. The `stat` module: file statistics

The `stat` module contains a number of variables used in encoding and decoding various items returned by certain methods in the `os` module, such as `stat()` and `chmod()`.

First, there are constants for indexing the components of a "status tuple" such as that returned by os.stat():

ST_MODE	The file's permissions.
ST_INO	The i-node number.
ST_DEV	The device number.
ST_NLINK	The number of hard links.
ST_UID	The user ID.

ST_GID	The group ID.
ST_SIZE	The current size in bytes.
ST_ATIME	The epoch time of last access (see the `time` module for interpretation of times).
ST_MTIME	The epoch time of last modification.
ST_CTIME	The epoch time of the file's creation.

The following functions are defined in the `stat` module for testing a mode value *m*, where *m* is the ST_MODE element of the status tuple. Each function is a predicate:

S_ISDIR(*m*)	Is this a directory?
S_ISCHR(*m*)	Is this a character device?
S_ISBLK(*m*)	Is this a block device?
S_ISREG(*m*)	Is this an ordinary file?
S_ISFIFO(*m*)	Is this a FIFO?
S_ISLNK(*m*)	Is this a soft (symbolic) link?
S_ISSOCK(*m*)	Is this a socket?

These constants are defined for use as mask values in testing and assembling permission values such as those returned by `os.stat()`:

S_ISUID	SUID (set user ID) bit.
S_ISGID	SGID (set group ID) bit.
S_IRUSR	Owner read permission.
S_IWUSR	Owner write permission.
S_IXUSR	Owner execute permission.
S_IRGRP	Group read permission.
S_IWGRP	Group write permission.
S_IXGRP	Group execute permission.
S_IROTH	World read permission.
S_IWOTH	World write permission.
S_IXOTH	World execute permission.

20.11. The path module: file and directory interface

These functions allow you to deal with path names and directory trees. To use them, import the `os` module and then use `os.path`. For example, to get the base name of a path *p*, use `os.path.basename(p)`.

basename(*p*)
Return the base name portion of a path name string *p*. See `split()`, below.

commonprefix(*L*)
For a list *L* containing pathname strings, return the longest string that is a prefix of each element in *L*.

exists(*p*)
Predicate for testing whether pathname *p* exists.

expanduser(*p*)
If *p* is a pathname starting with a tilde character (~), return the equivalent full pathname; otherwise return *p*.

isabs(*p*)
Predicate for testing whether *p* is an absolute pathname (e.g., starts with a slash on Unix systems).

isfile(*p*)
Predicate for testing whether *p* refers to a regular file, as opposed to a directory, link, or device.

islink(p)
Predicate for testing whether *p* is a soft (symbolic) link.

ismount(*p*)
Predicate for testing whether *p* is a mount point, that is, whether *p* is on a different device than its parent directory.

join(*p*,*q*)
If *q* is an absolute path, returns *q*. Otherwise, if *p* is empty or ends in a slash, returns *p*+*q*, but otherwise it returns *p*+'/'+*q*.

normcase(*p*)
Return pathname *p* with its case normalized. On Unix systems, this does nothing, but on Macs it lowercases *p*.

samefile(*p*,*q*)
Predicate for testing whether *p* and *q* are the same file (that is, the same inode on the same device). This method may raise an exception if `os.stat()` fails for either argument.

split(*p*)
Return a 2-tuple (*H*,*T*) where *T* is the tail end of the pathname (not containing a slash) and *H* is everything up to the tail. If *p* ends with a slash, returns (*p*,''). If *p* contains no slashes, returns (''*p*). The returned *H* string will have its trailing slash removed unless *H* is the root directory.

splitext(*p*)
Returns a 2-tuple (*R*,*E*) where *E* is the "extension" part of the pathname and *R* is the "root" part. If *p* contains at least one period, *E* will contain the last period and everything after that, and *R* will be everything up to but not including the last period. If *p* contains no periods, returns (*p*,'').

walk(*p*,*V*,*a*)
Walks an entire directory structure starting at pathname *p*. See below for more information.

The `os.path.walk(`*p*,*V*,*a*`)` function does the following for every directory at or below *p* (including *p* if *p* is a directory), this method calls the "visitor function" *V* with arguments

```
V(a,d,N)
```

where:

a	The same *a* passed to `os.path.walk()`. You can use *a* to provide information to the *V*() function, or to accumulate information throughout the traversal of the directory structure.
d	A string containing the name of the directory being visited.
N	A list of all the names within directory *d*. You can remove elements from this list in place if there are some elements of *d* that you don't want `walk()` to visit.

20.12. Low-level file functions in the os module

This group of functions operates on low-level file descriptors, which use integers for file handles. In these functions, the *f* argument is a low-level-file descriptor. These functions are part of module `os`.

Use these functions only when you really need low-level I/O. Most applications will use a "file object" as produced by the built-in `open()` function; see file objects above.

`close(`*`f`*`)`
Close file *f*.

`dup(`*`f`*`)`
Returns a new file descriptor that is a duplicate of *f*.

`fstat(`*`f`*`)`
Return the file's status tuple in the same format as `os.stat()`.

`lseek(`*`f`*`,`*`p`*`,`*`w`*`)`
Change the current position of *f*. The *p* and *f* arguments are interpreted as in the `.seek()` method for file objects.

`open(`*`p`*`,`*`f`*`,`*`m`*`)`
Open the file at pathname *p*. The value *f* describes various options, such as read or write access, and whether to create the file if it doesn't exist; see `/usr/include/fcntl.h` for C-language definitions for the *f* value. If you are creating the file, you can supply *m* to specify the initial permissions of the file as in `chmod()`. Returns a low-level file descriptor.

`pipe()`
Create a pipe. Returns a tuple (f_r, f_w) of two file descriptors, f_r for reading and f_w for writing.

`read(`*`f`*`,`*`n`*`)`
Read no more than *n* bytes. Returns the data as a string.

`write(`*`f`*`,`*`s`*`)`
Write string *s*.