2. Built-in Functions

The Python interpreter has a number of functions built into it that are always available. They are listed here in alphabetical order.

abs(X)

Return the absolute value of a number. The argument may be a plain or long integer or a floating point number. If the argument is a complex number, its magnitude is returned.

all(*iterable*)

Return True if all elements of the *iterable* are true (or if the iterable is empty). Equivalent to:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

New in version 2.5.

any(*iterable*)

Return True if any element of the *iterable* is true. If the iterable is empty, return False. Equivalent to:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

New in version 2.5.

basestring()

This abstract type is the superclass for str and unicode. It cannot be called or instantiated, but it can be used to test whether an object is an instance of str or unicode. isinstance(obj, basestring) is equivalent to isinstance(obj, (str, unicode)).

New in version 2.3.

bin(X)

Convert an integer number to a binary string. The result is a valid Python expression. If x is not a Python int object, it has to define an __index__() method that returns an integer.

New in version 2.6.

bool([x])

Convert a value to a Boolean, using the standard truth testing procedure. If x is false or omitted, this returns False; otherwise it returns True. bool is also a class, which is a subclass of int. Class bool cannot be subclassed further. Its only instances are False and True.

New in version 2.2.1.

Changed in version 2.3: If no argument is given, this function returns False.

callable(*Object*)

Return **True** if the *object* argument appears callable, **False** if not. If this returns true, it is still possible that a call fails, but if it is false, calling *object* will never succeed. Note that classes are callable (calling a class returns a new instance); class instances are callable if they have a <u>call</u> () method.

$\mathtt{chr}(i)$

Return a string of one character whose ASCII code is the integer *i*. For example, chr(97) returns the string 'a'. This is the inverse of ord(). The argument must be in the range [0..255], inclusive; valueError will be raised if *i* is outside that range. See also unichr().

classmethod(function)

Return a class method for function.

A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:

```
class C:
 @classmethod
 def f(cls, arg1, arg2, ...): ...
```

The @classmethod form is a function *decorator* – see the description of function definitions in *Function definitions* for details.

It can be called either on the class (such as c.f()) or on an instance (such as c().f()). The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.

Class methods are different than C++ or Java static methods. If you want those, see staticmethod() in this section.

For more information on class methods, consult the documentation on the standard type hierarchy in *The standard type hierarchy*.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

cmp(x, y)

Compare the two objects x and y and return an integer according to the outcome. The return value is negative if x < y, zero if x = y and strictly positive if x > y.

compile(source, filename, mode[, flags[, dont_inherit]])

Compile the *source* into a code or AST object. Code objects can be executed by an exec statement or evaluated by a call to eval(). *source* can either be a string or an AST object. Refer to the ast module documentation for information on how to work with AST objects.

The *filename* argument should give the file from which the code was read; pass some recognizable value if it wasn't read from a file ('<string>' is commonly used).

The *mode* argument specifies what kind of code must be compiled; it can be 'exec' if *source* consists of a sequence of statements, 'eval' if it consists of a single expression, or 'single' if it consists of a single interactive statement (in the latter case, expression statements that evaluate to something other than None will be printed).

The optional arguments *flags* and *dont_inherit* control which future statements (see PEP 236) affect the compilation of *source*. If neither is present (or both are zero) the code is compiled with those future statements that are in effect in the code that is calling compile. If the *flags* argument is given and *dont_inherit* is not (or is zero) then the future statements specified by the *flags* argument are used in addition to those that would be used anyway. If *dont_inherit* is a non-zero integer then the *flags* argument is it – the future statements in effect around the call to compile are ignored.

Future statements are specified by bits which can be bitwise ORed together to specify multiple statements. The bitfield required to specify a given feature can be found as the compiler_flag attribute on the _Feature instance in the _future_ module.

This function raises **syntaxError** if the compiled source is invalid, and **TypeError** if the source contains null bytes.

Note: When compiling a string with multi-line statements, line endings must be represented by a single newline character (' \n'), and the input must be terminated by at least one newline character. If line endings are represented by ' $\r\n'$, use str.replace() to change them into ' \n' .

Changed in version 2.3: The flags and dont_inherit arguments were added.

Changed in version 2.6: Support for compiling AST objects.

complex([real[, imag]])

Create a complex number with the value *real* + *imag**j or convert a string or number to a complex number. If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second

parameter can never be a string. Each argument may be any numeric type (including complex). If *imag* is omitted, it defaults to zero and the function serves as a numeric conversion function like *int()*, *long()* and *float()*. If both arguments are omitted, returns <code>0j</code>.

The complex type is described in *Numeric Types — int, float, long, complex*.

delattr(object, name)

This is a relative of setattr(). The arguments are an object and a string. The string must be the name of one of the object's attributes. The function deletes the named attribute, provided the object allows it. For example, delattr(x, 'foobar') is equivalent to del x.foobar.

dict([arg])

Create a new data dictionary, optionally with items taken from *arg*. The dictionary type is described in *Mapping Types* — *dict*.

For other containers see the built in list, set, and tuple classes, and the collections module.

dir([object])

Without arguments, return the list of names in the current local scope. With an argument, attempt to return a list of valid attributes for that object.

If the object has a method named <u>__dir__()</u>, this method will be called and must return the list of attributes. This allows objects that implement a custom <u>__getattr_()</u> or <u>__getattribute_()</u> function to customize the way <u>dir()</u> reports their attributes.

If the object does not provide <u>__dir__()</u>, the function tries its best to gather information from the object's <u>__dict__</u> attribute, if defined, and from its type object. The resulting list is not necessarily complete, and may be inaccurate when the object has a custom <u>__getattr__()</u>.

The default dir() mechanism behaves differently with different types of objects, as it attempts to produce the most relevant, rather than complete, information:

- If the object is a module object, the list contains the names of the module's attributes.
- If the object is a type or class object, the list contains the names of its attributes, and recursively of the attributes of its bases.
- Otherwise, the list contains the object's attributes' names, the names of its class's attributes, and recursively of the attributes of its class's base classes.

The resulting list is sorted alphabetically. For example:

```
>>> import struct
>>> dir()  # doctest: +SKIP
['_builtins_', '_doc_', '_name_', 'struct']
>>> dir(struct)  # doctest: +NORMALIZE_WHITESPACE
['Struct', '_builtins_', '_doc_', '_file_', '_name_',
    '_package__', 'clearcache', 'calcsize', 'error', 'pack', 'pack_into',
    'unpack', 'unpack_from']
>>> class Foo(object):
...    def __dir__(self):
...       return ["kan", "ga", "roo"]
...
>>> f = Foo()
>>> dir(f)
['ga', 'kan', 'roo']
```

Note: Because dir() is supplied primarily as a convenience for use at an interactive prompt, it tries to supply an interesting set of names more than it tries to supply a rigorously or consistently defined set of names, and its detailed behavior may change across releases. For example, metaclass attributes are not in the result list when the argument is a class.

divmod(a, b)

Take two (non complex) numbers as arguments and return a pair of numbers consisting of their quotient and remainder when using long division. With mixed operand types, the rules for binary arithmetic operators apply. For plain and long integers, the result is the same as (a // b, a % b). For floating point numbers the result is (q, a % b), where *q* is usually math.floor(a / b) but may be 1 less than that. In any case q * b + a % b is very close to *a*, if a % b is non-zero it has the same sign as *b*, and $0 \le abs(a \% b) \le abs(b)$.

Changed in version 2.3: Using divmod() with complex numbers is deprecated.

enumerate(sequence[, start=0])

Return an enumerate object. *sequence* must be a sequence, an *iterator*, or some other object which supports iteration. The **next(**) method of the iterator returned by **enumerate(**) returns a tuple containing a count (from *start* which defaults to 0) and the corresponding value obtained from iterating over *iterable*. **enumerate(**) is useful for obtaining an indexed series: (0, seq[0]), (1, seq[1]), (2, seq[2]), For example:

```
>>> for i, season in enumerate(['Spring', 'Summer', 'Fall', 'Winter']):
... print i, season
0 Spring
1 Summer
2 Fall
3 Winter
```

New in version 2.3.

New in version 2.6: The start parameter.

eval(expression[, globals[, locals]])

The arguments are a string and optional globals and locals. If provided, *globals* must be a dictionary. If provided, *locals* can be any mapping object.

Changed in version 2.4: formerly locals was required to be a dictionary.

The *expression* argument is parsed and evaluated as a Python expression (technically speaking, a condition list) using the *globals* and *locals* dictionaries as global and local namespace. If the *globals* dictionary is present and lacks '__builtins__', the current globals are copied into *globals* before *expression* is parsed. This means that *expression* normally has full access to the standard __builtin__ module and restricted environments are propagated. If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where eval() is called. The return value is the result of the evaluated expression. Syntax errors are reported as exceptions. Example:

```
>>> x = 1
>>> print eval('x+1')
2
```

This function can also be used to execute arbitrary code objects (such as those created by compile()). In this case pass a code object instead of a string. If the code object has been compiled with 'exec' as the *kind* argument, eval() 's return value will be None.

Hints: dynamic execution of statements is supported by the exec statement. Execution of statements from a file is supported by the execfile() function. The globals() and locals() functions returns the current global and local dictionary, respectively, which may be useful to pass around for use by eval() or execfile().

execfile(filename[, globals[, locals]])

This function is similar to the exec statement, but parses a file instead of a string. It is different from the import statement in that it does not use the module administration — it reads the file unconditionally and does not create a new module. [1]

The arguments are a file name and two optional dictionaries. The file is parsed and evaluated as a sequence of Python statements (similarly to a module) using the *globals* and *locals* dictionaries as global and local namespace. If provided, *locals* can be any mapping object.

Changed in version 2.4: formerly locals was required to be a dictionary.

If the *locals* dictionary is omitted it defaults to the *globals* dictionary. If both dictionaries are omitted, the expression is executed in the environment where execfile() is called. The return value is None.

Note: The default locals act as described for function locals() below: modifications to

the default *locals* dictionary should not be attempted. Pass an explicit *locals* dictionary if you need to see effects of the code on *locals* after function <code>execfile()</code> returns. <code>execfile()</code> cannot be used reliably to modify a function's locals.

file(filename[, mode[, bufsize]])

Constructor function for the file type, described further in section *File Objects*. The constructor's arguments are the same as those of the open() built-in function described below.

When opening a file, it's preferable to use **open()** instead of invoking this constructor directly. **file** is more suited to type testing (for example, writing <code>isinstance(f, file)</code>).

New in version 2.2.

filter(function, iterable)

Construct a list from those elements of *iterable* for which *function* returns true. *iterable* may be either a sequence, a container which supports iteration, or an iterator. If *iterable* is a string or a tuple, the result also has that type; otherwise it is always a list. If *function* is None, the identity function is assumed, that is, all elements of *iterable* that are false are removed.

Note that filter(function, iterable) is equivalent to [item for item in iterable if function(item)] if function is not None and [item for item in iterable if item] if function is None.

See itertools.filterfalse() for the complementary function that returns elements of *iterable* for which *function* returns false.

float([x])

Convert a string or a number to floating point. If the argument is a string, it must contain a possibly signed decimal or floating point number, possibly embedded in whitespace. The argument may also be [+|-]nan or [+|-]inf. Otherwise, the argument may be a plain or long integer or a floating point number, and a floating point number with the same value (within Python's floating point precision) is returned. If no argument is given, returns 0.0.

Note: When passing in a string, values for NaN and Infinity may be returned, depending on the underlying C library. Float accepts the strings nan, inf and -inf for NaN and positive or negative infinity. The case and a leading + are ignored as well as a leading - is ignored for NaN. Float always represents NaN and infinity as nan, inf or -inf.

The float type is described in *Numeric Types — int, float, long, complex*.

format(value[, format_spec])

Convert a *value* to a "formatted" representation, as controlled by *format_spec*. The interpretation of *format_spec* will depend on the type of the *value* argument, however there

is a standard formatting syntax that is used by most built-in types: *Format Specification Mini-Language*.

Note: format(value, format_spec) **merely calls** value.__format__(format_spec).

New in version 2.6.

frozenset([iterable])

Return a frozenset object, optionally with elements taken from *iterable*. The frozenset type is described in *Set Types* — *set, frozenset*.

For other containers see the built in dict, list, and tuple classes, and the collections module.

New in version 2.4.

getattr(object, name[, default])

Return the value of the named attributed of *object. name* must be a string. If the string is the name of one of the object's attributes, the result is the value of that attribute. For example, getattr(x, 'foobar') is equivalent to x.foobar. If the named attribute does not exist, *default* is returned if provided, otherwise AttributeError is raised.

globals()

Return a dictionary representing the current global symbol table. This is always the dictionary of the current module (inside a function or method, this is the module where it is defined, not the module from which it is called).

hasattr(object, name)

The arguments are an object and a string. The result is **True** if the string is the name of one of the object's attributes, **False** if not. (This is implemented by calling getattr(object, name) and seeing whether it raises an exception or not.)

hash(*Object*)

Return the hash value of the object (if it has one). Hash values are integers. They are used to quickly compare dictionary keys during a dictionary lookup. Numeric values that compare equal have the same hash value (even if they are of different types, as is the case for 1 and 1.0).

help([object])

Invoke the built-in help system. (This function is intended for interactive use.) If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console. If the argument is any other kind of object, a help page on the object is generated.

This function is added to the built-in namespace by the site module.

New in version 2.2.

hex(X)

Convert an integer number (of any size) to a hexadecimal string. The result is a valid Python expression.

Note: To obtain a hexadecimal string representation for a float, use the float.hex() method.

Changed in version 2.4: Formerly only returned an unsigned literal.

id(object)

Return the "identity" of an object. This is an integer (or long integer) which is guaranteed to be unique and constant for this object during its lifetime. Two objects with non-overlapping lifetimes may have the same id() value.

CPython implementation detail: This is the address of the object.

input([prompt])

Equivalent to eval(raw_input(prompt)).

Warning: This function is not safe from user errors! It expects a valid Python expression as input; if the input is not syntactically valid, a <u>syntaxError</u> will be raised. Other exceptions may be raised if there is an error during evaluation. (On the other hand, sometimes this is exactly what you need when writing a quick script for expert use.)

If the **readline** module was loaded, then **input(**) will use it to provide elaborate line editing and history features.

Consider using the raw_input() function for general input from users.

int([x[, base]])

Convert a string or number to a plain integer. If the argument is a string, it must contain a possibly signed decimal number representable as a Python integer, possibly embedded in whitespace. The *base* parameter gives the base for the conversion (which is 10 by default) and may be any integer in the range [2, 36], or zero. If *base* is zero, the proper radix is determined based on the contents of string; the interpretation is the same as for integer literals. (See *Numeric literals.*) If *base* is specified and *x* is not a string, **TypeError** is raised. Otherwise, the argument may be a plain or long integer or a floating point number. Conversion of floating point numbers to integers truncates (towards zero). If the argument is outside the integer range a long object will be returned instead. If no arguments are

given, returns o.

The integer type is described in *Numeric Types — int, float, long, complex*.

isinstance(Object, classinfo)

Return true if the *object* argument is an instance of the *classinfo* argument, or of a (direct or indirect) subclass thereof. Also return true if *classinfo* is a type object (new-style class) and *object* is an object of that type or of a (direct or indirect) subclass thereof. If *object* is not a class instance or an object of the given type, the function always returns false. If *classinfo* is neither a class object nor a type object, it may be a tuple of class or type objects, or may recursively contain other such tuples (other sequence types are not accepted). If *classinfo* is not a class, type, or tuple of classes, types, and such tuples, a TypeError exception is raised.

Changed in version 2.2: Support for a tuple of type information was added.

issubclass(class, classinfo)

Return true if *class* is a subclass (direct or indirect) of *classinfo*. A class is considered a subclass of itself. *classinfo* may be a tuple of class objects, in which case every entry in *classinfo* will be checked. In any other case, a **TypeError** exception is raised.

Changed in version 2.3: Support for a tuple of type information was added.

iter(o[, sentinel])

Return an *iterator* object. The first argument is interpreted very differently depending on the presence of the second argument. Without a second argument, *o* must be a collection object which supports the iteration protocol (the <u>__iter_()</u> method), or it must support the sequence protocol (the <u>__getitem_()</u> method with integer arguments starting at o). If it does not support either of those protocols, TypeError is raised. If the second argument, *sentinel*, is given, then *o* must be a callable object. The iterator created in this case will call *o* with no arguments for each call to its <u>next()</u> method; if the value returned is equal to *sentinel*, stopIteration will be raised, otherwise the value will be returned.

One useful application of the second form of *iter(*) is to read lines of a file until a certain line is reached. The following example reads a file until "STOP" is reached:

```
with open("mydata.txt") as fp:
    for line in iter(fp.readline, "STOP"):
        process_line(line)
```

New in version 2.2.

len(S)

Return the length (the number of items) of an object. The argument may be a sequence (string, tuple or list) or a mapping (dictionary).

list([iterable])

Return a list whose items are the same and in the same order as *iterable*'s items. *iterable* may be either a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a list, a copy is made and returned, similar to iterable[:]. For instance, list('abc') returns ['a', 'b', 'c'] and list((1, 2, 3)) returns [1, 2, 3]. If no argument is given, returns a new empty list, [].

list is a mutable sequence type, as documented in *Sequence Types* — *str*, *unicode*, *list*, *tuple*, *buffer*, *xrange*. For other containers see the built in dict, set, and tuple classes, and the collections module.

locals()

Update and return a dictionary representing the current local symbol table. Free variables are returned by locals() when it is called in function blocks, but not in class blocks.

Note: The contents of this dictionary should not be modified; changes may not affect the values of local and free variables used by the interpreter.

long([x[, base]])

Convert a string or number to a long integer. If the argument is a string, it must contain a possibly signed number of arbitrary size, possibly embedded in whitespace. The *base* argument is interpreted in the same way as for *int()*, and may only be given when *x* is a string. Otherwise, the argument may be a plain or long integer or a floating point number, and a long integer with the same value is returned. Conversion of floating point numbers to integers truncates (towards zero). If no arguments are given, returns OL.

The long type is described in *Numeric Types — int, float, long, complex*.

map(function, iterable, ...)

Apply *function* to every item of *iterable* and return a list of the results. If additional *iterable* arguments are passed, *function* must take that many arguments and is applied to the items from all iterables in parallel. If one iterable is shorter than another it is assumed to be extended with None items. If *function* is None, the identity function is assumed; if there are multiple arguments, map() returns a list consisting of tuples containing the corresponding items from all iterables (a kind of transpose operation). The *iterable* arguments may be a sequence or any iterable object; the result is always a list.

max(iterable[, args...][, key])

With a single argument *iterable*, return the largest item of a non-empty iterable (such as a string, tuple or list). With more than one argument, return the largest of the arguments.

The optional key argument specifies a one-argument ordering function like that used for <code>list.sort()</code>. The key argument, if supplied, must be in keyword form (for example, <code>max(a,b,c,key=func)</code>).

Changed in version 2.5: Added support for the optional key argument.

min(iterable[, args...][, key])

With a single argument *iterable*, return the smallest item of a non-empty iterable (such as a string, tuple or list). With more than one argument, return the smallest of the arguments.

The optional key argument specifies a one-argument ordering function like that used for <code>list.sort()</code>. The key argument, if supplied, must be in keyword form (for example, min(a,b,c,key=func)).

Changed in version 2.5: Added support for the optional key argument.

next(iterator[, default])

Retrieve the next item from the *iterator* by calling its **next()** method. If *default* is given, it is returned if the iterator is exhausted, otherwise **stopIteration** is raised.

New in version 2.6.

object()

Return a new featureless object. Object is a base for all new style classes. It has the methods that are common to all instances of new style classes.

New in version 2.2.

Changed in version 2.3: This function does not accept any arguments. Formerly, it accepted arguments but ignored them.

oct(x)

Convert an integer number (of any size) to an octal string. The result is a valid Python expression.

Changed in version 2.4: Formerly only returned an unsigned literal.

open(filename[, mode[, bufsize]])

Open a file, returning an object of the file type described in section *File Objects*. If the file cannot be opened, **IDError** is raised. When opening a file, it's preferable to use **open()** instead of invoking the file constructor directly.

The first two arguments are the same as for stdio's fopen(): filename is the file name to be opened, and *mode* is a string indicating how the file is to be opened.

The most commonly-used values of *mode* are 'r' for reading, 'w' for writing (truncating the file if it already exists), and 'a' for appending (which on *some* Unix systems means that *all* writes append to the end of the file regardless of the current seek position). If *mode* is omitted, it defaults to 'r'. The default is to use text mode, which may convert '\n' characters to a platform-specific representation on writing and back on reading. Thus,

when opening a binary file, you should append 'b' to the *mode* value to open the file in binary mode, which will improve portability. (Appending 'b' is useful even on systems that don't treat binary and text files differently, where it serves as documentation.) See below for more possible values of *mode*.

The optional *bufsize* argument specifies the file's desired buffer size: 0 means unbuffered, 1 means line buffered, any other positive value means use a buffer of (approximately) that size. A negative *bufsize* means to use the system default, which is usually line buffered for tty devices and fully buffered for other files. If omitted, the system default is used. [2]

Modes 'r+', 'w+' and 'a+' open the file for updating (note that 'w+' truncates the file). Append 'b' to the mode to open the file in binary mode, on systems that differentiate between binary and text files; on systems that don't have this distinction, adding the 'b' has no effect.

In addition to the standard fopen() values *mode* may be 'U' or 'rU'. Python is usually built with universal newline support; supplying 'U' opens the file as a text file, but lines may be terminated by any of the following: the Unix end-of-line convention '\n', the Macintosh convention '\r', or the Windows convention '\r\n'. All of these external representations are seen as '\n' by the Python program. If Python is built without universal newline support a *mode* with 'U' is the same as normal text mode. Note that file objects so opened also have an attribute called newlines which has a value of None (if no newlines have yet been seen), '\n', '\r', '\r\n', or a tuple containing all the newline types seen.

Python enforces that the mode, after stripping 'U', begins with 'r', 'w' or 'a'.

Python provides many file handling modules including fileinput, os, os.path, tempfile, and shutil.

Changed in version 2.5: Restriction on first letter of mode string introduced.

ord(C)

Given a string of length one, return an integer representing the Unicode code point of the character when the argument is a unicode object, or the value of the byte when the argument is an 8-bit string. For example, ord('a') returns the integer 97, ord(u'\u2020') returns 8224. This is the inverse of chr() for 8-bit strings and of unichr() for unicode objects. If a unicode argument is given and Python was built with UCS2 Unicode, then the character's code point must be in the range [0..65535] inclusive; otherwise the string length is two, and a TypeError will be raised.

pow(x, y[, z])

Return *x* to the power *y*; if *z* is present, return *x* to the power *y*, modulo *z* (computed more efficiently than $pow(x, y) \otimes z$). The two-argument form pow(x, y) is equivalent to using the power operator: x^{**y} .

The arguments must have numeric types. With mixed operand types, the coercion rules for

binary arithmetic operators apply. For int and long int operands, the result has the same type as the operands (after coercion) unless the second argument is negative; in that case, all arguments are converted to float and a float result is delivered. For example, 10^{**2} returns 100, but 10^{**-2} returns 0.01. (This last feature was added in Python 2.2. In Python 2.1 and before, if both arguments were of integer types and the second argument was negative, an exception was raised.) If the second argument is negative, the third argument must be omitted. If *z* is present, *x* and *y* must be of integer types, and *y* must be non-negative. (This restriction was added in Python 2.2. In Python 2.1 and before, floating 3-argument pow() returned platform-dependent results depending on floating-point rounding accidents.)

print([object, ...][, sep=' '][, end='\n'][, file=sys.stdout])

Print *object*(s) to the stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like str() does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be None, which means to use the default values. If no *object* is given, print() will just write *end*.

The file argument must be an object with a write(string) method; if it is not present or None, sys.stdout will be used.

Note: This function is not normally available as a built-in since the name print is recognized as the print statement. To disable the statement and use the print() function, use this future statement at the top of your module:

```
from __future__ import print_function
```

New in version 2.6.

property([fget[, fset[, fdel[, doc]]]])

Return a property attribute for *new-style classes* (classes that derive from object).

fget is a function for getting an attribute value, likewise *fset* is a function for setting, and *fdel* a function for del'ing, an attribute. Typical use is to define a managed attribute x:

```
class C(object):
    def __init__(self):
        self._x = None
    def getx(self):
        return self._x
    def setx(self, value):
        self._x = value
    def delx(self):
        del self._x
        x = property(getx, setx, delx, "I'm the 'x' property.")
```

If given, *doc* will be the docstring of the property attribute. Otherwise, the property will copy *fget*'s docstring (if it exists). This makes it possible to create read-only properties easily using **property**() as a *decorator*:

```
class Parrot(object):
    def __init__(self):
        self._voltage = 100000
    @property
    def voltage(self):
        """Get the current voltage."""
        return self. voltage
```

turns the voltage() method into a "getter" for a read-only attribute with the same name.

A property object has getter, setter, and deleter methods usable as decorators that create a copy of the property with the corresponding accessor function set to the decorated function. This is best explained with an example:

```
class C (object):
    def __init__(self):
        self._x = None
    @property
    def x(self):
        """I'm the 'x' property."""
        return self._x
    @x.setter
    def x(self, value):
        self._x = value
    @x.deleter
    def x(self):
        del self. x
```

This code is exactly equivalent to the first example. Be sure to give the additional functions the same name as the original property ($_{\times}$ in this case.)

The returned property also has the attributes fget, fset, and fdel corresponding to the constructor arguments.

New in version 2.2.

Changed in version 2.5: Use fget's docstring if no doc given.

Changed in version 2.6: The getter, setter, and deleter attributes were added.

range([start], stop[, step])

This is a versatile function to create lists containing arithmetic progressions. It is most often used in for loops. The arguments must be plain integers. If the *step* argument is omitted, it defaults to 1. If the *start* argument is omitted, it defaults to 0. The full form returns a list of plain integers [start, start + step, start + 2 * step, ...]. If *step* is positive, the last element is the largest start + i * step less than *stop*; if *step* is negative, the last element is the smallest start + i * step greater than *stop*. *step* must not be zero (or else valueError is raised). Example:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> range(0, 30, 5)
[0, 5, 10, 15, 20, 25]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(0, -10, -1)
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> range(0)
[]
>>> range(1, 0)
[]
```

raw_input([prompt])

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, **EOFError** is raised. Example:

```
>>> s = raw_input('--> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the **readline** module was loaded, then **raw_input()** will use it to provide elaborate line editing and history features.

reduce(function, iterable[, initializer])

Apply *function* of two arguments cumulatively to the items of *iterable*, from left to right, so as to reduce the iterable to a single value. For example, reduce (lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates ((((1+2)+3)+4)+5). The left argument, x, is the accumulated value and the right argument, y, is the update value from the *iterable*. If the optional *initializer* is present, it is placed before the items of the iterable in the calculation, and serves as a

default when the iterable is empty. If *initializer* is not given and *iterable* contains only one item, the first item is returned.

reload(module)

Reload a previously imported *module*. The argument must be a module object, so it must have been successfully imported before. This is useful if you have edited the module source file using an external editor and want to try out the new version without leaving the Python interpreter. The return value is the module object (the same as the *module* argument).

When reload(module) is executed:

- Python modules' code is recompiled and the module-level code reexecuted, defining a new set of objects which are bound to names in the module's dictionary. The init function of extension modules is not called a second time.
- As with all other objects in Python the old objects are only reclaimed after their reference counts drop to zero.
- The names in the module namespace are updated to point to any new or changed objects.
- Other references to the old objects (such as names external to the module) are not rebound to refer to the new objects and must be updated in each namespace where they occur if that is desired.

There are a number of other caveats:

If a module is syntactically correct but its initialization fails, the first *import* statement for it does not bind its name locally, but does store a (partially initialized) module object in sys.modules. To reload the module you must first *import* it again (this will bind the name to the partially initialized module object) before you can **reload()** it.

When a module is reloaded, its dictionary (containing the module's global variables) is retained. Redefinitions of names will override the old definitions, so this is generally not a problem. If the new version of a module does not define a name that was defined by the old version, the old definition remains. This feature can be used to the module's advantage if it maintains a global table or cache of objects — with a try statement it can test for the table's presence and skip its initialization if desired:

```
try:
    cache
except NameError:
    cache = {}
```

It is legal though generally not very useful to reload built-in or dynamically loaded modules, except for sys, <u>main</u> and <u>builtin</u>. In many cases, however, extension modules are not designed to be initialized more than once, and may fail in arbitrary ways when reloaded.

If a module imports objects from another module using from ... import ..., calling reload()

for the other module does not redefine the objects imported from it — one way around this is to re-execute the from statement, another is to use import and qualified names (*module*.*name*) instead.

If a module instantiates instances of a class, reloading the module that defines the class does not affect the method definitions of the instances — they continue to use the old class definition. The same is true for derived classes.

repr(Object)

Return a string containing a printable representation of an object. This is the same value yielded by conversions (reverse quotes). It is sometimes useful to be able to access this operation as an ordinary function. For many types, this function makes an attempt to return a string that would yield an object with the same value when passed to eval(), otherwise the representation is a string enclosed in angle brackets that contains the name of the type of the object together with additional information often including the name and address of the object. A class can control what this function returns for its instances by defining a __repr__() method.

reversed(seq)

Return a reverse *iterator*. *seq* must be an object which has a <u>__reversed__()</u> method or supports the sequence protocol (the <u>_len_()</u> method and the <u>__getitem_()</u> method with integer arguments starting at 0).

New in version 2.4.

Changed in version 2.6: Added the possibility to write a custom <u>__reversed__()</u> method.

round(x[, n])

Return the floating point value x rounded to n digits after the decimal point. If n is omitted, it defaults to zero. The result is a floating point number. Values are rounded to the closest multiple of 10 to the power minus n; if two multiples are equally close, rounding is done away from 0 (so. for example, round (0.5) is 1.0 and round (-0.5) is -1.0).

set([iterable])

Return a new set, optionally with elements are taken from *iterable*. The set type is described in *Set Types* — *set, frozenset*.

For other containers see the built in dict, list, and tuple classes, and the collections module.

New in version 2.4.

setattr(Object, name, value)

This is the counterpart of getattr(). The arguments are an object, a string and an arbitrary value. The string may name an existing attribute or a new attribute. The function assigns the value to the attribute, provided the object allows it. For example, setattr(x, 'foobar',

123) is equivalent to x.foobar = 123.

slice([start], stop[, step])

Return a *slice* object representing the set of indices specified by range(start, stop, step). The *start* and *step* arguments default to None. Slice objects have read-only data attributes **start**, **stop** and **step** which merely return the argument values (or their default). They have no other explicit functionality; however they are used by Numerical Python and other third party extensions. Slice objects are also generated when extended indexing syntax is used. For example: a[start:stop:step] Or a[start:stop, i]. See **itertools.islice()** for an alternate version that returns an iterator.

sorted(iterable[, cmp[, key[, reverse]]])

Return a new sorted list from the items in *iterable*.

The optional arguments *cmp*, *key*, and *reverse* have the same meaning as those for the <code>list.sort()</code> method (described in section *Mutable Sequence Types*).

cmp specifies a custom comparison function of two arguments (iterable elements) which should return a negative, zero or positive number depending on whether the first argument is considered smaller than, equal to, or larger than the second argument: cmp=lambda x,y: cmp(x.lower(), y.lower()). The default value is **None**.

key specifies a function of one argument that is used to extract a comparison key from each list element: key=str.lower. The default value is None.

reverse is a boolean value. If set to **True**, then the list elements are sorted as if each comparison were reversed.

In general, the *key* and *reverse* conversion processes are much faster than specifying an equivalent *cmp* function. This is because *cmp* is called multiple times for each list element while *key* and *reverse* touch each element only once. To convert an old-style *cmp* function to a *key* function, see the CmpToKey recipe in the ASPN cookbook.

New in version 2.4.

staticmethod(function)

Return a static method for function.

A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```
class C:
  @staticmethod
  def f(arg1, arg2, ...): ...
```

The <code>@staticmethod</code> form is a function *decorator* – see the description of function definitions in *Function definitions* for details.

It can be called either on the class (such as c.f()) or on an instance (such as c().f()). The instance is ignored except for its class.

Static methods in Python are similar to those found in Java or C++. For a more advanced concept, see classmethod() in this section.

For more information on static methods, consult the documentation on the standard type hierarchy in *The standard type hierarchy*.

New in version 2.2.

Changed in version 2.4: Function decorator syntax added.

str([object])

Return a string containing a nicely printable representation of an object. For strings, this returns the string itself. The difference with repr(object) is that str(object) does not always attempt to return a string that is acceptable to **eval()**; its goal is to return a printable string. If no argument is given, returns the empty string, ''.

For more information on strings see Sequence Types — str, unicode, list, tuple, buffer, *xrange* which describes sequence functionality (strings are sequences), and also the string-specific methods described in the *String Methods* section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. In addition see the *String Services* section. See also unicode().

sum(iterable[, start])

Sums *start* and the items of an *iterable* from left to right and returns the total. *start* defaults to 0. The *iterable*'s items are normally numbers, and are not allowed to be strings. The fast, correct way to concatenate a sequence of strings is by calling ''.join(sequence). Note that sum(range(n), m) is equivalent to reduce(operator.add, range(n), m) To add floating point values with extended precision, see math.fsum().

New in version 2.3.

super(type[, object-or-type])

Return a proxy object that delegates method calls to a parent or sibling class of *type*. This is useful for accessing inherited methods that have been overridden in a class. The search order is same as that used by getattr() except that the *type* itself is skipped.

The <u>mro</u> attribute of the *type* lists the method resolution search order used by both getattr() and super(). The attribute is dynamic and can change whenever the inheritance hierarchy is updated.

If the second argument is omitted, the super object returned is unbound. If the second argument is an object, <code>isinstance(obj, type)</code> must be true. If the second argument is a type, <code>issubclass(type2, type)</code> must be true (this is useful for classmethods).

Note: super() only works for *new-style classes*.

There are two typical use cases for *super*. In a class hierarchy with single inheritance, *super* can be used to refer to parent classes without naming them explicitly, thus making the code more maintainable. This use closely parallels the use of *super* in other programming languages.

The second use case is to support cooperative multiple inheritance in a dynamic execution environment. This use case is unique to Python and is not found in statically compiled languages or languages that only support single inheritance. This makes it possible to implement "diamond diagrams" where multiple base classes implement the same method. Good design dictates that this method have the same calling signature in every case (because the order of calls is determined at runtime, because that order adapts to changes in the class hierarchy, and because that order can include sibling classes that are unknown prior to runtime).

For both use cases, a typical superclass call looks like this:

```
class C(B):
    def method(self, arg):
        super(C, self).method(arg)
```

Note that super() is implemented as part of the binding process for explicit dotted attribute
lookups such as super().__getitem__(name). It does so by implementing its own
__getattribute__() method for searching classes in a predictable order that supports
cooperative multiple inheritance. Accordingly, super() is undefined for implicit lookups
using statements or operators such as super() [name].

Also note that **super()** is not limited to use inside methods. The two argument form specifies the arguments exactly and makes the appropriate references.

New in version 2.2.

tuple([iterable])

Return a tuple whose items are the same and in the same order as *iterable*'s items. *iterable* may be a sequence, a container that supports iteration, or an iterator object. If *iterable* is already a tuple, it is returned unchanged. For instance, tuple('abc') returns ('a', 'b', 'c') and tuple([1, 2, 3]) returns (1, 2, 3). If no argument is given, returns a new empty tuple, ().

tuple is an immutable sequence type, as documented in Sequence Types — str, unicode, list, tuple, buffer, xrange. For other containers see the built in dict, list, and set classes, and the collections module.

type(object)

Return the type of an object. The return value is a type object. The isinstance() built-in

function is recommended for testing the type of an object.

With three arguments, type() functions as a constructor as detailed below.

type(name, bases, dict)

Return a new type object. This is essentially a dynamic form of the **class** statement. The *name* string is the class name and becomes the <u>_name_</u> attribute; the *bases* tuple itemizes the base classes and becomes the <u>_bases_</u> attribute; and the *dict* dictionary is the namespace containing definitions for class body and becomes the <u>_dict_</u> attribute. For example, the following two statements create identical type objects:

```
>>> class X(object):
... a = 1
...
>>> X = type('X', (object,), dict(a=1))
```

New in version 2.2.

unichr(i)

Return the Unicode string of one character whose Unicode code is the integer *i*. For example, unichr(97) returns the string u'a'. This is the inverse of ord() for Unicode strings. The valid range for the argument depends how Python was configured – it may be either UCS2 [0..0xFFFF] or UCS4 [0..0x10FFFF]. valueError is raised otherwise. For ASCII and 8-bit strings see chr().

New in version 2.0.

unicode([object[, encoding[, errors]]])

Return the Unicode string version of *object* using one of the following modes:

If *encoding* and/or *errors* are given, <code>unicode()</code> will decode the object which can either be an 8-bit string or a character buffer using the codec for *encoding*. The *encoding* parameter is a string giving the name of an encoding; if the encoding is not known, <code>LookupError</code> is raised. Error handling is done according to *errors*; this specifies the treatment of characters which are invalid in the input encoding. If *errors* is 'strict' (the default), a <code>ValueError</code> is raised on errors, while a value of 'ignore' causes errors to be silently ignored, and a value of 'replace' causes the official Unicode replacement character, <code>U+FFFD</code>, to be used to replace input characters which cannot be decoded. See also the codecs module.

If no optional parameters are given, unicode() will mimic the behaviour of str() except that it returns Unicode strings instead of 8-bit strings. More precisely, if *object* is a Unicode string or subclass it will return that Unicode string without any additional decoding applied.

For objects which provide a <u>_______()</u> method, it will call this method without arguments to create a Unicode string. For all other objects, the 8-bit string version or representation is requested and then converted to a Unicode string using the codec for the

default encoding in 'strict' mode.

For more information on Unicode strings see *Sequence Types* — *str, unicode, list, tuple, buffer, xrange* which describes sequence functionality (Unicode strings are sequences), and also the string-specific methods described in the *String Methods* section. To output formatted strings use template strings or the % operator described in the *String Formatting Operations* section. In addition see the *String Services* section. See also str().

New in version 2.0.

Changed in version 2.2: Support for <u>__unicode__()</u> added.

vars([object])

Without an argument, act like locals().

Note: The returned dictionary should not be modified: the effects on the corresponding symbol table are undefined. [3]

xrange([start], stop[, step])

This function is very similar to range(), but returns an "xrange object" instead of a list. This is an opaque sequence type which yields the same values as the corresponding list, without actually storing them all simultaneously. The advantage of xrange() over range() is minimal (since xrange() still has to create the values when asked for them) except when a very large range is used on a memory-starved machine or when all of the range's elements are never used (such as when the loop is usually terminated with break).

CPython implementation detail: xrange() is intended to be simple and fast. Implementations may impose restrictions to achieve this. The C implementation of Python restricts all arguments to native C longs ("short" Python integers), and also requires that the number of elements fit in a native C long. If a larger range is needed, an alternate version can be crafted using the *itertools* module: *islice(count(start,*

step), (stop-start+step-1)//step).

zip([iterable, ...])

This function returns a list of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The returned list is truncated in length to the length of the shortest argument sequence. When there are multiple arguments which are all of the same length, zip() is similar to map() with an initial argument of None. With a single sequence argument, it returns a list of 1-tuples. With no arguments, it returns an empty list.

The left-to-right evaluation order of the iterables is guaranteed. This makes possible an idiom for clustering a data series into n-length groups using zip(*[iter(s)]*n).

zip() in conjunction with the * operator can be used to unzip a list:

```
>>> x = [1, 2, 3]
>>> y = [4, 5, 6]
>>> zipped
[(1, 4), (2, 5), (3, 6)]
>>> x2, y2 = zip(*zipped)
>>> x == list(x2) and y == list(y2)
True
```

New in version 2.0.

Changed in version 2.4: Formerly, zip() required at least one argument and zip() raised a TypeError instead of returning an empty list.

_import__(name[, globals[, locals[, fromlist[, level]]]])

Note: This is an advanced function that is not needed in everyday Python programming.

This function is invoked by the import statement. It can be replaced (by importing the builtins module and assigning to builtins.__import__) in order to change semantics of the import statement, but nowadays it is usually simpler to use import hooks (see PEP 302). Direct use of __import__() is rare, except in cases where you want to import a module whose name is only known at runtime.

The function imports the module *name*, potentially using the given *globals* and *locals* to determine how to interpret the name in a package context. The *fromlist* gives the names of objects or submodules that should be imported from the module given by *name*. The standard implementation does not use its *locals* argument at all, and uses its *globals* only to determine the package context of the *import* statement.

level specifies whether to use absolute or relative imports. The default is -1 which indicates both absolute and relative imports will be attempted. 0 means only perform absolute imports. Positive values for *level* indicate the number of parent directories to search relative to the directory of the module calling __import__().

When the *name* variable is of the form package.module, normally, the top-level package (the name up till the first dot) is returned, *not* the module named by *name*. However, when a non-empty *fromlist* argument is given, the module named by *name* is returned.

For example, the statement import spam results in bytecode resembling the following code:

spam = __import__('spam', globals(), locals(), [], -1)

The statement import spam.ham results in this call:

spam = __import__('spam.ham', globals(), locals(), [], -1)

Note how <u>_import_()</u> returns the toplevel module here because this is the object that is bound to a name by the <u>import</u> statement.

On the other hand, the statement from spam.ham import eggs, sausage as saus results in

```
_temp = __import__('spam.ham', globals(), locals(), ['eggs', 'sausage'], -1)
eggs = _temp.eggs
saus = temp.sausage
```

Here, the spam.ham module is returned from <u>_import_()</u>. From this object, the names to import are retrieved and assigned to their respective names.

If you simply want to import a module (potentially within a package) by name, you can call ________() and then look it up in sys.modules:

```
>>> import sys
>>> name = 'foo.bar.baz'
>>> __import__(name)
<module 'foo' from ...>
>>> baz = sys.modules[name]
>>> baz
<module 'foo.bar.baz' from ...>
```

Changed in version 2.5: The level parameter was added.

Changed in version 2.5: Keyword support for parameters was added.

3. Non-essential Built-in Functions

There are several built-in functions that are no longer essential to learn, know or use in modern Python programming. They have been kept here to maintain backwards compatibility with programs written for older versions of Python.

Python programmers, trainers, students and book writers should feel free to bypass these functions without concerns about missing something important.

apply(function, args[, keywords])

The *function* argument must be a callable object (a user-defined or built-in function or method, or a class object) and the *args* argument must be a sequence. The *function* is called with *args* as the argument list; the number of arguments is the length of the tuple. If the optional *keywords* argument is present, it must be a dictionary whose keys are strings. It specifies keyword arguments to be added to the end of the argument list. Calling <code>apply()</code> is different from just calling <code>function(args)</code>, since in that case there is always exactly one argument. The use of <code>apply()</code> is equivalent to <code>function(*args, **keywords)</code>.

Deprecated since version 2.3: Use the extended call syntax with *args and **keywords instead.

buffer(object[, offset[, size]])

The *object* argument must be an object that supports the buffer call interface (such as strings, arrays, and buffers). A new buffer object will be created which references the *object* argument. The buffer object will be a slice from the beginning of *object* (or from the specified *offset*). The slice will extend to the end of *object* (or will have a length given by the *size* argument).

coerce(X, Y)

Return a tuple consisting of the two numeric arguments converted to a common type, using the same rules as used by arithmetic operations. If coercion is not possible, raise TypeError.

intern(string)

Enter *string* in the table of "interned" strings and return the interned string – which is *string* itself or a copy. Interning strings is useful to gain a little performance on dictionary lookup – if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. Normally, the names used in Python programs are automatically interned, and the dictionaries used to hold module, class or instance attributes have interned keys.

Changed in version 2.3: Interned strings are not immortal (like they used to be in Python 2.2 and before); you must keep a reference to the return value of *intern()* around to benefit from it.

Footnotes

- [1] It is used relatively rarely so does not warrant being made into a statement.
- [2] Specifying a buffer size currently has no effect on systems that don't have setvbuf(). The interface to specify the buffer size is not done using a method that calls setvbuf(), because that may dump core when called after any I/O has been performed, and there's no reliable way to determine whether this is the case.
- [3] In the current implementation, local variable bindings cannot normally be affected this way, but variables retrieved from other scopes (such as modules) can be. This may change.