BBA Sem-III

Notes

SICSR

By

Sanjay Sharnangat

List

Python knows a number of compound data types, used to group together other values. The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

**>>>** squares = [1, 4, 9, 16, 25]

**>>>** squares

[1, 4, 9, 16, 25]

**>>** squares[0] *# indexing returns the item*

1

**>>>** squares[-1]

25

**>>>** squares[-3:] *# slicing returns a new list*

[9, 16, 25]

Slicing example

**>>>** squares[:]

[1, 4, 9, 16, 25]

Concatenation Example

**>>>** squares + [36, 49, 64, 81, 100]

[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]

**>>>** cubes = [1, 8, 27, 65, 125] *# something's wrong here*

**Mutable example**

**>>>** 4 \*\* 3 *# the cube of 4 is 64, not 65!*

64

**>>>** cubes[3] = 64 *# replace the wrong value*

**>>>** cubes

[1, 8, 27, 64, 125]

Adding items at end of List

**>>>** cubes.append(216) *# add the cube of 6*

**>>>** cubes.append(7 \*\* 3) *# and the cube of 7*

**>>>** cubes

[1, 8, 27, 64, 125, 216, 343]

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

>>>

**>>>** letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** letters

['a', 'b', 'c', 'd', 'e', 'f', 'g']

**>>>** *# replace some values*

**>>>** letters[2:5] = ['C', 'D', 'E']

**>>>** letters

['a', 'b', 'C', 'D', 'E', 'f', 'g']

**>>>** *# now remove them*

**>>>** letters[2:5] = []

**>>>** letters

['a', 'b', 'f', 'g']

**>>>** *# clear the list by replacing all the elements with an empty list*

**>>>** letters[:] = []

**>>>** letters

[]

The built-in function [len()](https://docs.python.org/3/library/functions.html" \l "len" \o "len) also applies to lists:

>>>

**>>>** letters = ['a', 'b', 'c', 'd']

**>>>** len(letters)

4

It is possible to nest lists (create lists containing other lists), for example:

>>>

**>>>** a = ['a', 'b', 'c']

**>>>** n = [1, 2, 3]

**>>>** x = [a, n]

**>>>** x

[['a', 'b', 'c'], [1, 2, 3]]

**>>>** x[0]

['a', 'b', 'c']

**>>>** x[0][1]

'b'

String

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes ('...') or double quotes ("...") with the same result [2](https://docs.python.org/3/tutorial/introduction.html#id4). \ can be used to escape quotes:

>>>

**>>>** 'spam eggs' *# single quotes*

'spam eggs'

**>>>** 'doesn**\'**t' *# use \' to escape the single quote...*

"doesn't"

**>>>** "doesn't" *# ...or use double quotes instead*

"doesn't"

**>>>** '"Yes," they said.'

'"Yes," they said.'

**>>>** "**\"**Yes,**\"** they said."

'"Yes," they said.'

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

**>>>** print('"Isn**\'**t," they said.')

"Isn't," they said.

**>>>** s = 'First line.**\n**Second line.' *# \n means newline*

**>>>** s *# without print(), \n is included in the output*

'First line.\nSecond line.'

**>>>** print(s) *# with print(), \n produces a new line*

First line.

Second line.

Example

**>>>** '"Isn**\'**t," they said.'

'"Isn\'t," they said.'

**>>>** print('"Isn**\'**t," they said.')

"Isn't," they said.

**>>>** s = 'First line.**\n**Second line.' *# \n means newline*

**>>>** s *# without print(), \n is included in the output*

'First line.\nSecond line.'

**>>>** print(s) *# with print(), \n produces a new line*

First line.

Second line.

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it’s possible to prevent this by adding a \ at the end of the line. The following example:

print("""**\**

Usage: thingy [OPTIONS]

-h Display this usage message

-H hostname Hostname to connect to

""")

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

>>>

**>>>** word = 'Python'

**>>>** word[0] *# character in position 0*

'P'

**>>>** word[5] *# character in position 5*

'n'

Indices may also be negative numbers, to start counting from the right:

>>>

**>>>** word[-1] *# last character*

'n'

**>>>** word[-2] *# second-last character*

'o'

**>>>** word[-6]

'P'

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

>>>

**>>>** word[0:2] *# characters from position 0 (included) to 2 (excluded)*

'Py'

**>>>** word[2:5] *# characters from position 2 (included) to 5 (excluded)*

'tho'

Note how the start is always included, and the end always excluded. This makes sure that s[:i] + s[i:] is always equal to s:

>>>

**>>>** word[:2] + word[2:]

'Python'

**>>>** word[:4] + word[4:]

'Python'

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

>>>

**>>>** word[:2] *# character from the beginning to position 2 (excluded)*

'Py'

**>>>** word[4:] *# characters from position 4 (included) to the end*

'on'

**>>>** word[-2:] *# characters from the second-last (included) to the end*

'on'

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of *n* characters has index *n*, for example:

+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+

0 1 2 3 4 5 6

-6 -5 -4 -3 -2 -1

Tuples

A tuple consists of a number of values separated by commas, for instance:

>>>

**>>>** t = 12345, 54321, 'hello!'

**>>>** t[0]

12345

**>>>** t

(12345, 54321, 'hello!')

**>>>** *# Tuples may be nested:*

**...** u = t, (1, 2, 3, 4, 5)

**>>>** u

((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))

**>>>** *# Tuples are immutable:*

**...** t[0] = 88888

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'tuple' object does not support item assignment

**>>>** *# but they can contain mutable objects:*

**...** v = ([1, 2, 3], [3, 2, 1])

**>>>** v

([1, 2, 3], [3, 2, 1])

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are [immutable](https://docs.python.org/3/glossary.html#term-immutable), and usually contain a heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of [namedtuples](https://docs.python.org/3/library/collections.html" \l "collections.namedtuple" \o "collections.namedtuple)). Lists are [mutable](https://docs.python.org/3/glossary.html#term-mutable), and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

>>>

**>>>** empty = ()

**>>>** singleton = 'hello', *# <-- note trailing comma*

**>>>** len(empty)

0

**>>>** len(singleton)

1

**>>>** singleton

('hello',)

The statement t = 12345, 54321, 'hello!' is an example of *tuple packing*: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

>>>

**>>>** x, y, z = t

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4. Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries. Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the [set()](https://docs.python.org/3/library/stdtypes.html#set) function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

>>>

**>>>** basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}

**>>>** print(basket) *# show that duplicates have been removed*

{'orange', 'banana', 'pear', 'apple'}

**>>>** 'orange' **in** basket *# fast membership testing*

True

**>>>** 'crabgrass' **in** basket

False

**>>>** *# Demonstrate set operations on unique letters from two words*

**...**

**>>>** a = set('abracadabra')

**>>>** b = set('alacazam')

**>>>** a *# unique letters in a*

{'a', 'r', 'b', 'c', 'd'}

**>>>** a - b *# letters in a but not in b*

{'r', 'd', 'b'}

**>>>** a | b *# letters in a or b or both*

{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}

**>>>** a & b *# letters in both a and b*

{'a', 'c'}

**>>>** a ^ b *# letters in a or b but not both*

{'r', 'd', 'b', 'm', 'z', 'l'}

Similarly to [list comprehensions](https://docs.python.org/3/tutorial/datastructures.html#tut-listcomps), set comprehensions are also supported:

>>>

**>>>** a = {x **for** x **in** 'abracadabra' **if** x **not** **in** 'abc'}

**>>>** a

{'r', 'd'}

Dictionaries

Another useful data type built into Python is the *dictionary* (see [Mapping Types — dict](https://docs.python.org/3/library/stdtypes.html#typesmapping)). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like append() and extend().

It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing list(d) on a dictionary returns a list of all the keys used in the dictionary, in insertion order (if you want it sorted, just use sorted(d) instead). To check whether a single key is in the dictionary, use the [in](https://docs.python.org/3/reference/expressions.html#in) keyword.

Here is a small example using a dictionary:

>>>

**>>>** tel = {'jack': 4098, 'sape': 4139}

**>>>** tel['guido'] = 4127

**>>>** tel

{'jack': 4098, 'sape': 4139, 'guido': 4127}

**>>>** tel['jack']

4098

**>>> del** tel['sape']

**>>>** tel['irv'] = 4127

**>>>** tel

{'jack': 4098, 'guido': 4127, 'irv': 4127}

**>>>** list(tel)

['jack', 'guido', 'irv']

**>>>** sorted(tel)

['guido', 'irv', 'jack']

**>>>** 'guido' **in** tel

True

**>>>** 'jack' **not** **in** tel

False

The [dict()](https://docs.python.org/3/library/stdtypes.html" \l "dict" \o "dict) constructor builds dictionaries directly from sequences of key-value pairs:

>>>

**>>>** dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])

{'sape': 4139, 'guido': 4127, 'jack': 4098}

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

>>>

**>>>** {x: x\*\*2 **for** x **in** (2, 4, 6)}

{2: 4, 4: 16, 6: 36}

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

>>>

**>>>** dict(sape=4139, guido=4127, jack=4098)

{'sape': 4139, 'guido': 4127, 'jack': 4098}

Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

>>>

**>>>** knights = {'gallahad': 'the pure', 'robin': 'the brave'}

**>>> for** k, v **in** knights.items():

**...**  print(k, v)

**...**

gallahad the pure

robin the brave

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) function.

>>>

**>>> for** i, v **in** enumerate(['tic', 'tac', 'toe']):

**...**  print(i, v)

**...**

0 tic

1 tac

2 toe

To loop over two or more sequences at the same time, the entries can be paired with the [zip()](https://docs.python.org/3/library/functions.html#zip) function.

>>>

**>>>** questions = ['name', 'quest', 'favorite color']

**>>>** answers = ['lancelot', 'the holy grail', 'blue']

**>>> for** q, a **in** zip(questions, answers):

**...**  print('What is your *{0}*? It is *{1}*.'.format(q, a))

**...**

What is your name? It is lancelot.

What is your quest? It is the holy grail.

What is your favorite color? It is blue.

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the [reversed()](https://docs.python.org/3/library/functions.html#reversed) function.

>>>

**>>> for** i **in** reversed(range(1, 10, 2)):

**...**  print(i)

**...**

9

7

5

3

1

To loop over a sequence in sorted order, use the [sorted()](https://docs.python.org/3/library/functions.html#sorted) function which returns a new sorted list while leaving the source unaltered.

>>>

**>>>** basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']

**>>> for** f **in** sorted(set(basket)):

**...**  print(f)

**...**

apple

banana

orange

pear

It is sometimes tempting to change a list while you are looping over it; however, it is often simpler and safer to create a new list instead.

>>>

**>>> import** **math**

**>>>** raw\_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]

**>>>** filtered\_data = []

**>>> for** value **in** raw\_data:

**...**  **if** **not** math.isnan(value):

**...**  filtered\_data.append(value)

**...**

**>>>** filtered\_data

[56.2, 51.7, 55.3, 52.5, 47.8]

More on Conditions

The conditions used in while and if statements can contain any operators, not just comparisons.

The comparison operators in and not in check whether a value occurs (does not occur) in a sequence. The operators is and is not compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison0020operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, a < b == c tests whether a is less than b and moreover b equals c.

Comparisons may be combined using the Boolean operators and and or, and the outcome of a comparison (or of any other Boolean expression) may be negated with not. These have lower priorities than comparison operators; between them, not has the highest priority and or the lowest, so that A and not B or C is equivalent to (A and (not B)) or C. As always, parentheses can be used to express the desired composition.

The Boolean operators and and or are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if A and C are true but B is false, A and B and C does not evaluate the expression C. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

>>>

**>>>** string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'

**>>>** non\_null = string1 **or** string2 **or** string3

**>>>** non\_null

'Trondheim'

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode code point number to order individual characters. Some examples of comparisons between sequences of the same type:

(1, 2, 3) < (1, 2, 4)

[1, 2, 3] < [1, 2, 4]

'ABC' < 'C' < 'Pascal' < 'Python'

(1, 2, 3, 4) < (1, 2, 4)

(1, 2) < (1, 2, -1)

(1, 2, 3) == (1.0, 2.0, 3.0)

(1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError) exception.

## Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

>>>

**>>> while** **True** print('Hello world')

File "<stdin>", line 1

**while** **True** print('Hello world')

^

SyntaxError: invalid syntax

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function [print()](https://docs.python.org/3/library/functions.html#print), since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

>>>

**>>>** 10 \* (1/0)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ZeroDivisionError: division by zero

**>>>** 4 + spam\*3

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: name 'spam' is not defined

**>>>** '2' + 2

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: Can't convert 'int' object to str implicitly

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are [ZeroDivisionError](https://docs.python.org/3/library/exceptions.html" \l "ZeroDivisionError" \o "ZeroDivisionError), [NameError](https://docs.python.org/3/library/exceptions.html" \l "NameError" \o "NameError) and [TypeError](https://docs.python.org/3/library/exceptions.html" \l "TypeError" \o "TypeError). The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using Control-C or whatever the operating system supports); note that a user-generated interruption is signalled by raising the [KeyboardInterrupt](https://docs.python.org/3/library/exceptions.html" \l "KeyboardInterrupt" \o "KeyboardInterrupt) exception.

>>>

**>>> while** **True**:

**...**  **try**:

**...**  x = int(input("Please enter a number: "))

**...**  **break**

**...**  **except** ValueError:

**...**  print("Oops! That was no valid number. Try again...")

**...**

The [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement works as follows.

* First, the *try clause* (the statement(s) between the [try](https://docs.python.org/3/reference/compound_stmts.html#try) and [except](https://docs.python.org/3/reference/compound_stmts.html#except) keywords) is executed.
* If no exception occurs, the *except clause* is skipped and execution of the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement is finished.
* If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the [except](https://docs.python.org/3/reference/compound_stmts.html#except) keyword, the except clause is executed, and then execution continues after the [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement.
* If an exception occurs which does not match the exception named in the except clause, it is passed on to outer [try](https://docs.python.org/3/reference/compound_stmts.html#try) statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A [try](https://docs.python.org/3/reference/compound_stmts.html#try) statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same try statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

... **except** (RuntimeError, TypeError, NameError):

... **pass**

A class in an [except](https://docs.python.org/3/reference/compound_stmts.html#except) clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an except clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

**class** **B**(Exception):

**pass**

**class** **C**(B):

**pass**

**class** **D**(C):

**pass**

**for** cls **in** [B, C, D]:

**try**:

**raise** cls()

**except** D:

print("D")

**except** C:

print("C")

**except** B:

print("B")

Note that if the except clauses were reversed (with except B first), it would have printed B, B, B — the first matching except clause is triggered.

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

**import** **sys**

**try**:

f = open('myfile.txt')

s = f.readline()

i = int(s.strip())

**except** OSError **as** err:

print("OS error: *{0}*".format(err))

**except** ValueError:

print("Could not convert data to an integer.")

**except**:

print("Unexpected error:", sys.exc\_info()[0])

**raise**

he use of the else clause is better than adding additional code to the [try](https://docs.python.org/3/reference/compound_stmts.html#try) clause because it avoids accidentally catching an exception that wasn’t raised by the code being protected by the try … except statement.

When an exception occurs, it may have an associated value, also known as the exception’s *argument*. The presence and type of the argument depend on the exception type.

The except clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in instance.args. For convenience, the exception instance defines [\_\_str\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__str__) so the arguments can be printed directly without having to reference .args. One may also instantiate an exception first before raising it and add any attributes to it as desired.

>>>

**>>> try**:

**...**  **raise** Exception('spam', 'eggs')

**... except** Exception **as** inst:

**...**  print(type(inst)) *# the exception instance*

**...**  print(inst.args) *# arguments stored in .args*

**...**  print(inst) *# \_\_str\_\_ allows args to be printed directly,*

**...**  *# but may be overridden in exception subclasses*

**...**  x, y = inst.args *# unpack args*

**...**  print('x =', x)

**...**  print('y =', y)

**...**

<class 'Exception'>

('spam', 'eggs')

('spam', 'eggs')

x = spam

y = eggs

If an exception has arguments, they are printed as the last part (‘detail’) of the message for unhandled exceptions.

Exception handlers don’t just handle exceptions if they occur immediately in the try clause, but also if they occur inside functions that are called (even indirectly) in the try clause. For example:

>>>

**>>> def** this\_fails():

**...**  x = 1/0

**...**

**>>> try**:

**...**  this\_fails()

**... except** ZeroDivisionError **as** err:

**...**  print('Handling run-time error:', err)

**...**

Handling run-time error: division by zero

Raising Exceptions

The [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows the programmer to force a specified exception to occur. For example:

>>>

**>>> raise** NameError('HiThere')

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

NameError: HiThere

The sole argument to [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from [Exception](https://docs.python.org/3/library/exceptions.html#Exception)). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

**raise** ValueError *# shorthand for 'raise ValueError()'*

If you need to determine whether an exception was raised but don’t intend to handle it, a simpler form of the [raise](https://docs.python.org/3/reference/simple_stmts.html#raise) statement allows you to re-raise the exception:

>>>

**>>> try**:

**...**  **raise** NameError('HiThere')

**... except** NameError:

**...**  print('An exception flew by!')

**...**  **raise**

**...**

An exception flew by!

Traceback (most recent call last):

File "<stdin>", line 2, in <module>

NameError: HiThere

User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see [Classes](https://docs.python.org/3/tutorial/classes.html#tut-classes) for more about Python classes). Exceptions should typically be derived from the [Exception](https://docs.python.org/3/library/exceptions.html#Exception) class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

**class** **Error**(Exception):

*"""Base class for exceptions in this module."""*

**pass**

**class** **InputError**(Error):

*"""Exception raised for errors in the input.*

*Attributes:*

*expression -- input expression in which the error occurred*

*message -- explanation of the error*

*"""*

**def** \_\_init\_\_(self, expression, message):

self.expression = expression

self.message = message

**class** **TransitionError**(Error):

*"""Raised when an operation attempts a state transition that's not*

*allowed.*

*Attributes:*

*previous -- state at beginning of transition*

*next -- attempted new state*

*message -- explanation of why the specific transition is not allowed*

*"""*

**def** \_\_init\_\_(self, previous, next, message):

self.previous = previous

self.next = next

self.message = message

Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python’s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data. As is true for modules, classes partake of the dynamic nature of Python: they are created at runtime, and can be modified further after creation.

In C++ terminology, normally class members (including the data members) are public (except see below [Private Variables](https://docs.python.org/3/tutorial/classes.html#tut-private)), and all member functions are virtual. As in Modula-3, there are no shorthands for referencing the object’s members from its methods: the method function is declared with an explicit first argument representing the object, which is provided implicitly by the call. As in Smalltalk, classes themselves are objects. This provides semantics for importing and renaming. Unlike C++ and Modula-3, built-in types can be used as base classes for extension by the user. Also, like in C++, most built-in operators with special syntax (arithmetic operators, subscripting etc.) can be redefined for class instances.

(Lacking universally accepted terminology to talk about classes, I will make occasional use of Smalltalk and C++ terms. I would use Modula-3 terms, since its object-oriented semantics are closer to those of Python than C++, but I expect that few readers have heard of it.)

A Word About Names and Objects

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

### Class Definition Syntax

The simplest form of class definition looks like this:

**class** **ClassName**:

<statement-1>

.

.

.

<statement-N>

Class definitions, like function definitions ([def](https://docs.python.org/3/reference/compound_stmts.html#def) statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an [if](https://docs.python.org/3/reference/compound_stmts.html#if) statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we’ll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the end), a class object is created. This is basically a wrapper around the contents of the namespace created by the class definition; we’ll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (ClassName in the example)

### Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: obj.name. Valid attribute names are all the names that were in the class’s namespace when the class object was created. So, if the class definition looked like this:

**class** **MyClass**:

*"""A simple example class"""*

i = 12345

**def** f(self):

**return** 'hello world'

then MyClass.i and MyClass.f are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of MyClass.i by assignment. \_\_doc\_\_ is also a valid attribute, returning the docstring belonging to the class: "A simple example class".

Class instantiation uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

x = MyClass()

creates a new instance of the class and assigns this object to the local variable x.

The instantiation operation (“calling” a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named [\_\_init\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__init__), like this:

**def** \_\_init\_\_(self):

self.data = []

When a class defines an [\_\_init\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__init__) method, class instantiation automatically invokes [\_\_init\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__init__) for the newly-created class instance. So in this example, a new, initialized instance can be obtained by:

x = MyClass()

Of course, the [\_\_init\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__init__) method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to [\_\_init\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__init__). For example,

>>>

**>>> class** **Complex**:

**...**  **def** \_\_init\_\_(self, realpart, imagpart):

**...**  self.r = realpart

**...**  self.i = imagpart

**...**

**>>>** x = Complex(3.0, -4.5)

**>>>** x.r, x.i

(3.0, -4.5)

### Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names, data attributes and methods.

data attributes correspond to “instance variables” in Smalltalk, and to “data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if x is the instance of MyClass created above, the following piece of code will print the value 16, without leaving a trace:

x.counter = 1

**while** x.counter < 10:

x.counter = x.counter \* 2

print(x.counter)

**del** x.counter

The other kind of instance attribute reference is a method. A method is a function that “belongs to” an object. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects have methods called append, insert, remove, sort, and so on. However, in the following discussion, we’ll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, x.f is a valid method reference, since MyClass.f is a function, but x.i is not, since MyClass.i is not. But x.f is not the same thing as MyClass.f — it is a method object, not a function object.

### Method Objects

Usually, a method is called right after it is bound:

x.f()

In the MyClass example, this will return the string 'hello world'. However, it is not necessary to call a method right away: x.f is a method object, and can be stored away and called at a later time. For example:

xf = x.f

**while** **True**:

print(xf())

will continue to print hello world until the end of time.

What exactly happens when a method is called? You may have noticed that x.f() was called without an argument above, even though the function definition for f() specified an argument. What happened to the argument? Surely Python raises an exception when a function that requires an argument is called without any — even if the argument isn’t actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call x.f() is exactly equivalent to MyClass.f(x). In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method’s instance object before the first argument.

If you still don’t understand how methods work, a look at the implementation can perhaps clarify matters. When a non-data attribute of an instance is referenced, the instance’s class is searched. If the name denotes a valid class attribute that is a function object, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

### Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

**class** **Dog**:

kind = 'canine' *# class variable shared by all instances*

**def** \_\_init\_\_(self, name):

self.name = name *# instance variable unique to each instance*

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.kind *# shared by all dogs*

'canine'

>>> e.kind *# shared by all dogs*

'canine'

>>> d.name *# unique to d*

'Fido'

>>> e.name *# unique to e*

'Buddy'

As discussed in [A Word About Names and Objects](https://docs.python.org/3/tutorial/classes.html#tut-object), shared data can have possibly surprising effects with involving [mutable](https://docs.python.org/3/glossary.html#term-mutable) objects such as lists and dictionaries. For example, the tricks list in the following code should not be used as a class variable because just a single list would be shared by all Dog instances:

**class** **Dog**:

tricks = [] *# mistaken use of a class variable*

**def** \_\_init\_\_(self, name):

self.name = name

**def** add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks *# unexpectedly shared by all dogs*

['roll over', 'play dead']

Correct design of the class should use an instance variable instead:

**class** **Dog**:

**def** \_\_init\_\_(self, name):

self.name = name

self.tricks = [] *# creates a new empty list for each dog*

**def** add\_trick(self, trick):

self.tricks.append(trick)

>>> d = Dog('Fido')

>>> e = Dog('Buddy')

>>> d.add\_trick('roll over')

>>> e.add\_trick('play dead')

>>> d.tricks

['roll over']

>>> e.tricks

['play dead']

Inheritance

Of course, a language feature would not be worthy of the name “class” without supporting inheritance. The syntax for a derived class definition looks like this:

**class** **DerivedClassName**(BaseClassName):

<statement-1>

.

.

.

<statement-N>

The name BaseClassName must be defined in a scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

**class** **DerivedClassName**(modname.BaseClassName):

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There’s nothing special about instantiation of derived classes: DerivedClassName() creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively virtual.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call BaseClassName.methodname(self, arguments). This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as BaseClassName in the global scope.)

Python has two built-in functions that work with inheritance:

* Use [isinstance()](https://docs.python.org/3/library/functions.html" \l "isinstance" \o "isinstance) to check an instance’s type: isinstance(obj, int) will be True only if obj.\_\_class\_\_ is [int](https://docs.python.org/3/library/functions.html" \l "int" \o "int) or some class derived from [int](https://docs.python.org/3/library/functions.html#int).
* Use [issubclass()](https://docs.python.org/3/library/functions.html" \l "issubclass" \o "issubclass) to check class inheritance: issubclass(bool, int) is True since [bool](https://docs.python.org/3/library/functions.html" \l "bool" \o "bool) is a subclass of [int](https://docs.python.org/3/library/functions.html#int). However, issubclass(float, int) is False since [float](https://docs.python.org/3/library/functions.html#float) is not a subclass of [int](https://docs.python.org/3/library/functions.html#int).

### Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

**class** **DerivedClassName**(Base1, Base2, Base3):

<statement-1>

.

.

.

<statement-N>

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in DerivedClassName, it is searched for in Base1, then (recursively) in the base classes of Base1, and if it was not found there, it was searched for in Base2, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to [super()](https://docs.python.org/3/library/functions.html#super). This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from [object](https://docs.python.org/3/library/functions.html#object), so any case of multiple inheritance provides more than one path to reach [object](https://docs.python.org/3/library/functions.html#object). To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see

Private Variables

“Private” instance variables that cannot be accessed except from inside an object don’t exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. \_spam) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form \_\_spam (at least two leading underscores, at most one trailing underscore) is textually replaced with \_classname\_\_spam, where classname is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

**class** **Mapping**:

**def** \_\_init\_\_(self, iterable):

self.items\_list = []

self.\_\_update(iterable)

**def** update(self, iterable):

**for** item **in** iterable:

self.items\_list.append(item)

\_\_update = update *# private copy of original update() method*

**class** **MappingSubclass**(Mapping):

**def** update(self, keys, values):

*# provides new signature for update()*

*# but does not break \_\_init\_\_()*

**for** item **in** zip(keys, values):

self.items\_list.append(item)

The above example would work even if MappingSubclass were to introduce a \_\_update identifier since it is replaced with \_Mapping\_\_update in the Mapping class and \_MappingSubclass\_\_update in the MappingSubclass class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to exec() or eval() does not consider the classname of the invoking class to be the current class; this is similar to the effect of the global statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to getattr(), setattr() and delattr(), as well as when referencing \_\_dict\_\_ directly.

Iterators

By now you have probably noticed that most container objects can be looped over using a [for](https://docs.python.org/3/reference/compound_stmts.html" \l "for) statement:

**for** element **in** [1, 2, 3]:

print(element)

**for** element **in** (1, 2, 3):

print(element)

**for** key **in** {'one':1, 'two':2}:

print(key)

**for** char **in** "123":

print(char)

**for** line **in** open("myfile.txt"):

print(line, end='')

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the [for](https://docs.python.org/3/reference/compound_stmts.html" \l "for) statement calls [iter()](https://docs.python.org/3/library/functions.html" \l "iter" \o "iter) on the container object. The function returns an iterator object that defines the method [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) which accesses elements in the container one at a time. When there are no more elements, [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) raises a [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration) exception which tells the for loop to terminate. You can call the [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) method using the [next()](https://docs.python.org/3/library/functions.html#next) built-in function; this example shows how it all works:

>>>

**>>>** s = 'abc'

**>>>** it = iter(s)

**>>>** it

<iterator object at 0x00A1DB50>

**>>>** next(it)

'a'

**>>>** next(it)

'b'

**>>>** next(it)

'c'

**>>>** next(it)

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

next(it)

StopIteration

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an [\_\_iter\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__iter__) method which returns an object with a [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) method. If the class defines \_\_next\_\_(), then [\_\_iter\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__iter__) can just return self:

**class** **Reverse**:

*"""Iterator for looping over a sequence backwards."""*

**def** \_\_init\_\_(self, data):

self.data = data

self.index = len(data)

**def** \_\_iter\_\_(self):

**return** self

**def** \_\_next\_\_(self):

**if** self.index == 0:

**raise** StopIteration

self.index = self.index - 1

**return** self.data[self.index]

>>>

**>>>** rev = Reverse('spam')

**>>>** iter(rev)

<\_\_main\_\_.Reverse object at 0x00A1DB50>

**>>> for** char **in** rev:

**...**  print(char)

**...**

m

a

p

s

9.9. Generators

[Generator](https://docs.python.org/3/glossary.html#term-generator)s are a simple and powerful tool for creating iterators. They are written like regular functions but use the [yield](https://docs.python.org/3/reference/simple_stmts.html#yield) statement whenever they want to return data. Each time [next()](https://docs.python.org/3/library/functions.html#next) is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

**def** reverse(data):

**for** index **in** range(len(data)-1, -1, -1):

**yield** data[index]

>>>

**>>> for** char **in** reverse('golf'):

**...**  print(char)

**...**

f

l

o

g

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the [\_\_iter\_\_()](https://docs.python.org/3/reference/datamodel.html#object.__iter__) and [\_\_next\_\_()](https://docs.python.org/3/reference/expressions.html#generator.__next__) methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like self.index and self.data.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise [StopIteration](https://docs.python.org/3/library/exceptions.html" \l "StopIteration" \o "StopIteration). In combination, these features make it easy to create iterators with no more effort than writing a regular function.

Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

>>>

**>>>** sum(i\*i **for** i **in** range(10)) *# sum of squares*

285

**>>>** xvec = [10, 20, 30]

**>>>** yvec = [7, 5, 3]

**>>>** sum(x\*y **for** x,y **in** zip(xvec, yvec)) *# dot product*

260

**>>> from** **math** **import** pi, sin

**>>>** sine\_table = {x: sin(x\*pi/180) **for** x **in** range(0, 91)}

**>>>** unique\_words = set(word **for** line **in** page **for** word **in** line.split())

**>>>** valedictorian = max((student.gpa, student.name) **for** student **in** graduates)

**>>>** data = 'golf'

**>>>** list(data[i] **for** i **in** range(len(data)-1, -1, -1))

['f', 'l', 'o', 'g']

Python Scopes

Function inside the function uses nonlocal variable. The variable should not belong to the inner function.

def f1():

x=”SICSR”

def f2():

nonlocal x

x=”Thanks”

f2()

return x

print(“Caling f1”,f1())

### Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how [global](https://docs.python.org/3/reference/simple_stmts.html#global) and [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) affect variable binding:

**def** scope\_test():

**def** do\_local():

spam = "local spam"

**def** do\_nonlocal():

**nonlocal** spam

spam = "nonlocal spam"

**def** do\_global():

**global** spam

spam = "global spam"

spam = "test spam"

do\_local()

print("After local assignment:", spam)

do\_nonlocal()

print("After nonlocal assignment:", spam)

do\_global()

print("After global assignment:", spam)

scope\_test()

print("In global scope:", spam)

The output of the example code is:

After local assignment: test spam

After nonlocal assignment: nonlocal spam

After global assignment: nonlocal spam

In global scope: global spam

Note how the local assignment (which is default) didn’t change scope\_test’s binding of spam. The [nonlocal](https://docs.python.org/3/reference/simple_stmts.html#nonlocal) assignment changed scope\_test’s binding of spam, and the [global](https://docs.python.org/3/reference/simple_stmts.html#global) assignment changed the module-level binding.

You can also see that there was no previous binding for spam before the [global](https://docs.python.org/3/reference/simple_stmts.html#global) assignment.

7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

Fancier Output Formatting

So far we’ve encountered two ways of writing values: *expression statements* and the [print()](https://docs.python.org/3/library/functions.html#print) function. (A third way is using the write() method of file objects; the standard output file can be referenced as sys.stdout. See the Library Reference for more information on this.)

Often you’ll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

* To use [formatted string literals](https://docs.python.org/3/tutorial/inputoutput.html#tut-f-strings), begin a string with f or F before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between { and } characters that can refer to variables or literal values.

>>>

**>>>** year = 2016

**>>>** event = 'Referendum'

**>>>** f'Results of the *{year}* *{event}*'

'Results of the 2016 Referendum'

* The [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format) method of strings requires more manual effort. You’ll still use { and } to mark where a variable will be substituted and can provide detailed formatting directives, but you’ll also need to provide the information to be formatted.

>>>

**>>>** yes\_votes = 42\_572\_654

**>>>** no\_votes = 43\_132\_495

**>>>** percentage = yes\_votes / (yes\_votes + no\_votes)

**>>>** '*{:-9}* YES votes *{:2.2%}*'.format(yes\_votes, percentage)

' 42572654 YES votes 49.67%'

* Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don’t need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the [repr()](https://docs.python.org/3/library/functions.html" \l "repr" \o "repr) or [str()](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "str) functions.

The [str()](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "str) function is meant to return representations of values which are fairly human-readable, while [repr()](https://docs.python.org/3/library/functions.html" \l "repr" \o "repr) is meant to generate representations which can be read by the interpreter (or will force a [SyntaxError](https://docs.python.org/3/library/exceptions.html#SyntaxError) if there is no equivalent syntax). For objects which don’t have a particular representation for human consumption, [str()](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "str) will return the same value as [repr()](https://docs.python.org/3/library/functions.html" \l "repr" \o "repr). Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

>>>

**>>>** s = 'Hello, world.'

**>>>** str(s)

'Hello, world.'

**>>>** repr(s)

"'Hello, world.'"

**>>>** str(1/7)

'0.14285714285714285'

**>>>** x = 10 \* 3.25

**>>>** y = 200 \* 200

**>>>** s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'

**>>>** print(s)

The value of x is 32.5, and y is 40000...

**>>>** *# The repr() of a string adds string quotes and backslashes:*

**...** hello = 'hello, world**\n**'

**>>>** hellos = repr(hello)

**>>>** print(hellos)

'hello, world\n'

**>>>** *# The argument to repr() may be any Python object:*

**...** repr((x, y, ('spam', 'eggs')))

"(32.5, 40000, ('spam', 'eggs'))"

The [string](https://docs.python.org/3/library/string.html#module-string) module contains a [Template](https://docs.python.org/3/library/string.html#string.Template) class that offers yet another way to substitute values into strings, using placeholders like $x and replacing them with values from a dictionary, but offers much less control of the formatting.

Formatted String Literals

[Formatted string literals](https://docs.python.org/3/reference/lexical_analysis.html#f-strings) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with f or F and writing expressions as {expression}.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

>>>

**>>> import** **math**

**>>>** print(f'The value of pi is approximately *{math.pi:.3f}*.')

The value of pi is approximately 3.142.

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

>>>

**>>>** table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}

**>>> for** name, phone **in** table.items():

**...**  print(f'*{name:10}* ==> *{phone:10d}*')

**...**

Sjoerd ==> 4127

Jack ==> 4098

Dcab ==> 7678

Other modifiers can be used to convert the value before it is formatted. '!a' applies [ascii()](https://docs.python.org/3/library/functions.html" \l "ascii" \o "ascii), '!s' applies [str()](https://docs.python.org/3/library/stdtypes.html" \l "str" \o "str), and '!r' applies [repr()](https://docs.python.org/3/library/functions.html" \l "repr" \o "repr):

>>>

**>>>** animals = 'eels'

**>>>** print(f'My hovercraft is full of *{animals}*.')

My hovercraft is full of eels.

**>>>** print(f'My hovercraft is full of *{animals!r}*.')

My hovercraft is full of 'eels'.

For a reference on these format specifications, see the reference guide for the [Format Specification Mini-Language](https://docs.python.org/3/library/string.html#formatspec).

The String format() Method

Basic usage of the [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format) method looks like this:

>>>

**>>>** print('We are the *{}* who say "*{}*!"'.format('knights', 'Ni'))

We are the knights who say "Ni!"

The brackets and characters within them (called format fields) are replaced with the objects passed into the [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format) method. A number in the brackets can be used to refer to the position of the object passed into the [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format) method.

>>>

**>>>** print('*{0}* and *{1}*'.format('spam', 'eggs'))

spam and eggs

**>>>** print('*{1}* and *{0}*'.format('spam', 'eggs'))

eggs and spam

If keyword arguments are used in the [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format) method, their values are referred to by using the name of the argument.

>>>

**>>>** print('This *{food}* is *{adjective}*.'.format(

**...**  food='spam', adjective='absolutely horrible'))

This spam is absolutely horrible.

Positional and keyword arguments can be arbitrarily combined:

>>>

**>>>** print('The story of *{0}*, *{1}*, and *{other}*.'.format('Bill', 'Manfred',

other='Georg'))

The story of Bill, Manfred, and Georg.

If you have a really long format string that you don’t want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys

>>>

**>>>** table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}

**>>>** print('Jack: *{0[Jack]:d}*; Sjoerd: *{0[Sjoerd]:d}*; '

**...**  'Dcab: *{0[Dcab]:d}*'.format(table))

Jack: 4098; Sjoerd: 4127; Dcab: 8637678

This could also be done by passing the table as keyword arguments with the ‘\*\*’ notation.

>>>

**>>>** table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}

**>>>** print('Jack: *{Jack:d}*; Sjoerd: *{Sjoerd:d}*; Dcab: *{Dcab:d}*'.format(\*\*table))

Jack: 4098; Sjoerd: 4127; Dcab: 8637678

This is particularly useful in combination with the built-in function [vars()](https://docs.python.org/3/library/functions.html" \l "vars" \o "vars), which returns a dictionary containing all local variables.

As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes:

>>>

**>>> for** x **in** range(1, 11):

**...**  print('*{0:2d}* *{1:3d}* *{2:4d}*'.format(x, x\*x, x\*x\*x))

**...**

1 1 1

2 4 8

3 9 27

4 16 64

5 25 125

6 36 216

7 49 343

8 64 512

9 81 729

10 100 1000

For a complete overview of string formatting with [str.format()](https://docs.python.org/3/library/stdtypes.html" \l "str.format" \o "str.format), see [Format String Syntax](https://docs.python.org/3/library/string.html#formatstrings).

Manual String Formatting

Here’s the same table of squares and cubes, formatted manually:

>>>

**>>> for** x **in** range(1, 11):

**...**  print(repr(x).rjust(2), repr(x\*x).rjust(3), end=' ')

**...**  *# Note use of 'end' on previous line*

**...**  print(repr(x\*x\*x).rjust(4))

**...**

1 1 1

2 4 8

3 9 27

4 16 64

5 25 125

6 36 216

7 49 343

8 64 512

9 81 729

10 100 1000

(Note that the one space between each column was added by the way [print()](https://docs.python.org/3/library/functions.html#print) works: it always adds spaces between its arguments.)

The [str.rjust()](https://docs.python.org/3/library/stdtypes.html" \l "str.rjust" \o "str.rjust) method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods [str.ljust()](https://docs.python.org/3/library/stdtypes.html" \l "str.ljust" \o "str.ljust) and [str.center()](https://docs.python.org/3/library/stdtypes.html" \l "str.center" \o "str.center). These methods do not write anything, they just return a new string. If the input string is too long, they don’t truncate it, but return it unchanged; this will mess up your column lay-out but that’s usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in x.ljust(n)[:n].)

There is another method, [str.zfill()](https://docs.python.org/3/library/stdtypes.html" \l "str.zfill" \o "str.zfill), which pads a numeric string on the left with zeros. It understands about plus and minus signs:

>>>

**>>>** '12'.zfill(5)

'00012'

**>>>** '-3.14'.zfill(7)

'-003.14'

**>>>** '3.14159265359'.zfill(5)

'3.14159265359'

Old string formatting

The % operator can also be used for string formatting. It interprets the left argument much like a sprintf()-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

>>>

**>>> import** **math**

**>>>** print('The value of pi is approximately *%5.3f*.' % math.pi)

The value of pi is approximately 3.142.

More information can be found in the [printf-style String Formatting](https://docs.python.org/3/library/stdtypes.html" \l "old-string-formatting) section.

Reading and Writing Files

[open()](https://docs.python.org/3/library/functions.html#open) returns a [file object](https://docs.python.org/3/glossary.html#term-file-object), and is most commonly used with two arguments: open(filename, mode).

>>>

**>>>** f = open('workfile', 'w')

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be 'r' when the file will only be read, 'w' for only writing (an existing file with the same name will be erased), and 'a' opens the file for appending; any data written to the file is automatically added to the end. 'r+' opens the file for both reading and writing. The *mode* argument is optional; 'r' will be assumed if it’s omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see [open()](https://docs.python.org/3/library/functions.html#open)). 'b' appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don’t contain text.

In text mode, the default when reading is to convert platform-specific line endings (\n on Unix, \r\n on Windows) to just \n. When writing in text mode, the default is to convert occurrences of \n back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the [with](https://docs.python.org/3/reference/compound_stmts.html#with) keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using with is also much shorter than writing equivalent [try](https://docs.python.org/3/reference/compound_stmts.html#try)-[finally](https://docs.python.org/3/reference/compound_stmts.html#finally) blocks:

>>>

**>>> with** open('workfile') **as** f:

**...**  read\_data = f.read()

**>>>** f.closed

True

If you’re not using the [with](https://docs.python.org/3/reference/compound_stmts.html#with) keyword, then you should call f.close() to close the file and immediately free up any system resources used by it. If you don’t explicitly close a file, Python’s garbage collector will eventually destroy the object and close the open file for you, but the file may stay open for a while. Another risk is that different Python implementations will do this clean-up at different times.

After a file object is closed, either by a [with](https://docs.python.org/3/reference/compound_stmts.html#with) statement or by calling f.close(), attempts to use the file object will automatically fail.

>>>

**>>>** f.close()

**>>>** f.read()

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

ValueError: I/O operation on closed file.

Methods of File Objects

The rest of the examples in this section will assume that a file object called f has already been created.

To read a file’s contents, call f.read(size), which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it’s your problem if the file is twice as large as your machine’s memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, f.read() will return an empty string ('').

>>>

**>>>** f.read()

'This is the entire file.\n'

**>>>** f.read()

''

f.readline() reads a single line from the file; a newline character (\n) is left at the end of the string, and is only omitted on the last line of the file if the file doesn’t end in a newline. This makes the return value unambiguous; if f.readline() returns an empty string, the end of the file has been reached, while a blank line is represented by '\n', a string containing only a single newline.

>>>

**>>>** f.readline()

'This is the first line of the file.\n'

**>>>** f.readline()

'Second line of the file\n'

**>>>** f.readline()

''

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

>>>

**>>> for** line **in** f:

**...**  print(line, end='')

**...**

This is the first line of the file.

Second line of the file

If you want to read all the lines of a file in a list you can also use list(f) or f.readlines().

f.write(string) writes the contents of *string* to the file, returning the number of characters written.

>>>

**>>>** f.write('This is a test**\n**')

15

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

>>>

**>>>** value = ('the answer', 42)

**>>>** s = str(value) *# convert the tuple to string*

**>>>** f.write(s)

18

f.tell() returns an integer giving the file object’s current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object’s position, use f.seek(offset, from\_what). The position is computed from adding *offset* to a reference point; the reference point is selected by the *from\_what* argument. A *from\_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from\_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

>>>

**>>>** f = open('workfile', 'rb+')

**>>>** f.write(b'0123456789abcdef')

16

**>>>** f.seek(5) *# Go to the 6th byte in the file*

5

**>>>** f.read(1)

b'5'

**>>>** f.seek(-3, 2) *# Go to the 3rd byte before the end*

13

**>>>** f.read(1)

b'd'

In text files (those opened without a b in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with seek(0, 2)) and the only valid *offset* values are those returned from the f.tell(), or zero. Any other *offset* value produces undefined behaviour.

File objects have some additional methods, such as isatty() and truncate() which are less frequently used; consult the Library Reference for a complete guide to file objects.