**Lambda Expressions (Anonymous Functions)**

Lambda:

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. The expression lambda parameters: expression yields a function object. The unnamed object behaves like a function object defined with:

def <lambda>(parameters):

return expression

**Note that functions created with lambda expressions cannot contain statements or annotations.**

Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

**>>> def** make\_incrementor(n):

**...**  **return** **lambda** x: x + n

**...**

**>>>** f = make\_incrementor(42)

**>>>** f(0)

42

**>>>** f(1)

43

**>>>** pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]

**>>>** pairs.sort(key=**lambda** pair: pair[1])

**>>>** pairs

[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]

**list comprehension**

A compact way to process all or part of the elements in a sequence and return a list with theresults result= ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0] generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The [if](https://docs.python.org/3/reference/compound_stmts.html#if) clause is optional. If omitted, all elements in range(256) are processed.

**Iterators**

**Iteration:** It is a general term for taking each item of something, one after another. Any time you use a loop, explicit or implicit, to go over a group of items, that is iteration.

**Iterable:** An **iterable** is an object that has an \_\_iter\_\_ method which returns an **iterator**, or which defines a \_\_getitem\_\_ method that can take sequential indexes starting from zero (and raises an IndexErrorwhen the indexes are no longer valid). So an **iterable** is an object that you can get an **iterator** from.

**Iterator**: It is an object with a next method. Whenever you use a for loop, or map, or a list comprehension, etc. in Python, the next method is called automatically to get each item from the **iterator**, thus going through the process of **iteration**.

**Generator**

A generator is simply a function which returns an object on which you can call next, such that for every call it returns some value, until it raises a StopIteration exception, signaling that all values have been generated. Such an object is called an iterator.

Normal functions return a single value using return, just like in Java. In Python, however, there is an alternative, called yield. Using yield anywhere in a function makes it a generator.

>>> def myGen(n):

... yield n

... yield n + 1

...

>>> g = myGen(6)

>>> next(g)

6

>>> next(g)

7

>>> next(g)

Traceback (most recent call last):

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

StopIteration

**generator iterator**

An object created by a [generator](https://docs.python.org/3/glossary.html#term-generator) function.

Each [yield](https://docs.python.org/3/reference/simple_stmts.html#yield) temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

**generator expression**

An expression that returns an iterator. It looks like a normal expression followed by a for clause defining a loop variable, range, and an optional if clause. The combined expression generates values for an enclosing function:

**>>>** sum(i\*i **for** i **in** range(10)) *# sum of squares 0, 1, 4, ... 81*

285

**Generic Function**

A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

**Built-in Functions**

**enumerate**(*iterable*, *start=0*)

Return an enumerate object. *iterable* must be a sequence, an [iterator](https://docs.python.org/3/glossary.html#term-iterator), or some other object which supports iteration. The [\_\_next\_\_()](https://docs.python.org/3/library/stdtypes.html#iterator.__next__) method of the iterator returned by [enumerate()](https://docs.python.org/3/library/functions.html#enumerate) returns a tuple containing a count (from *start* which defaults to 0) and the values obtained from iterating over *iterable*.

**>>>** seasons = ['Spring', 'Summer', 'Fall', 'Winter']

**>>>** list(enumerate(seasons))

[(0, 'Spring'), (1, 'Summer'), (2, 'Fall'), (3, 'Winter')]

**>>>** list(enumerate(seasons, start=1))

[(1, 'Spring'), (2, 'Summer'), (3, 'Fall'), (4, 'Winter')]

**zip**(*\*iterables*)

Make an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the *i*-th tuple contains the *i*-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

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

**>>>** y = [4, 5, 6]

**>>>** zipped = zip(x, y)

**>>>** list(zipped)

[(1, 4), (2, 5), (3, 6)]

**>>>** x2, y2 = zip(\*zip(x, y))

**>>>** x == list(x2) **and** y == list(y2)

True

**map**(*function*, *iterable*, *...*)

Return an iterator that applies *function* to every item of *iterable*, yielding 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. With multiple iterables, the iterator stops when the shortest iterable is exhausted.

**filter**(*function*, *iterable*)

Construct an iterator 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 *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 the generator expression (item for item initerable if function(item)) if function is not None and (item for item in iterable if item) if function is None.

# [functools](https://docs.python.org/3/library/functools.html#module-functools) — Higher-order functions and operations on callable objects

**functools**.**reduce**(*function*, *iterable*[, *initializer*])[¶](https://docs.python.org/3/library/functools.html#functools.reduce)

Apply *function* of two arguments cumulatively to the items of *sequence*, from left to right, so as to reduce the sequence 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 *sequence*. If the optional *initializer* is present, it is placed before the items of the sequence in the calculation, and serves as a default when the sequence is empty. If *initializer* is not given and *sequence* contains only one item, the first item is returned.

# Input and Output

## 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).

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:**

1. close()

Flush and close the IO object.

This method has no effect if the file is already closed.

1. detach()

Separate the underlying buffer from the TextIOBase and return it.

After the underlying buffer has been detached, the TextIO is in an

unusable state.

1. fileno()

Returns underlying file descriptor if one exists.

OSError is raised if the IO object does not use a file descriptor.

1. flush()

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

1. read(size=-1)

Read from underlying buffer until we have n characters or we hit EOF.

If n is negative or omitted, read until EOF.

1. readable()

Return whether object was opened for reading. If False, read() will raise OSError.

1. readline(size=-1)

Read until newline or EOF.

Returns an empty string if EOF is hit immediately.

1. seek(cookie, whence=0, /)

Change stream position.

Change the stream position to the given byte offset. The offset is

interpreted relative to the position indicated by whence. Values

for whence are:

\* 0 -- start of stream (the default); offset should be zero or positive

\* 1 -- current stream position; offset may be negative

\* 2 -- end of stream; offset is usually negative

Return the new absolute position.

1. seekable()

Return whether object supports random access.

If False, seek(), tell() and truncate() will raise OSError.

This method may need to do a test seek().

1. tell()

Return current stream position.

1. truncate(pos=None)

Truncate file to size bytes.

File pointer is left unchanged. Size defaults to the current IO

position as reported by tell(). Returns the new size.

1. writable()

Return whether object was opened for writing.

If False, write() will raise OSError.

1. write(text)

Write string to stream.

Returns the number of characters written (which is always equal to the length of the string).

1. readlines(hint=-1)

Return a list of lines from the stream.

hint can be specified to control the number of lines read: no more

lines will be read if the total size (in bytes/characters) of all

lines so far exceeds hint.

1. writelines( lines)

**pickle-Python Object Serialization**

The [pickle](https://docs.python.org/3/library/pickle.html#module-pickle) module implements binary protocols for serializing and de-serializing a Python object structure.“Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a [binary file](https://docs.python.org/3/glossary.html#term-binary-file) or [bytes-like object](https://docs.python.org/3/glossary.html#term-bytes-like-object)) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” [[1]](https://docs.python.org/3/library/pickle.html#id6) or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

**Methods:**

1. pickle.**dump**(obj, file, protocol=None, \*, fix\_imports=True)

Write a pickled representation of obj to the open [file object](https://docs.python.org/3/glossary.html#term-file-object) file. This is equivalent to Pickler(file,protocol).dump(obj).

1. pickle.**dumps**(obj, protocol=None, \*, fix\_imports=True)

Return the pickled representation of the object as a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object, instead of writing it to a file.

1. pickle.**load**(file, \*, fix\_imports=True, encoding="ASCII", errors="strict")

Read a pickled object representation from the open [file object](https://docs.python.org/3/glossary.html#term-file-object) file and return the reconstituted object hierarchy specified therein. This is equivalent to Unpickler(file).load().

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

The argument file must have two methods, a read() method that takes an integer argument, and a readline() method that requires no arguments. Both methods should return bytes. Thus file can be an on-disk file opened for binary reading, an [io.BytesIO](https://docs.python.org/3/library/io.html#io.BytesIO) object, or any other custom object that meets this interface.

1. pickle.**loads**(*bytes\_object*, *\**, *fix\_imports=True*, *encoding="ASCII"*, *errors="strict"*)[¶](https://docs.python.org/3/library/pickle.html#pickle.loads)

Read a pickled object hierarchy from a [bytes](https://docs.python.org/3/library/stdtypes.html#bytes) object and return the reconstituted object hierarchy specified therein.

The protocol version of the pickle is detected automatically, so no protocol argument is needed. Bytes past the pickled object’s representation are ignored.

The following types can be pickled:

* None, True, and False
* integers, floating point numbers, complex numbers
* strings, bytes, bytearrays
* tuples, lists, sets, and dictionaries containing only picklable objects
* functions defined at the top level of a module (using [def](https://docs.python.org/3/reference/compound_stmts.html#def), not [lambda](https://docs.python.org/3/reference/expressions.html#lambda))
* built-in functions defined at the top level of a module
* classes that are defined at the top level of a module
* [lambda](https://docs.python.org/3/reference/expressions.html#lambda) functions **cannot be pickled:** all lambda functions share the same name: <lambda>.

# Decorators :