Argparse Tutorial

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

author Tshepang Lekhonkhobe

This tutorial is intended to be a gentle introduction to `argparse`, the recommended command-line parsing module in the Python standard library. This was written for argparse in Python 3. A few details are different in 2.x, especially some exception messages, which were improved in 3.x.

Note: There are two other modules that fulfill the same task, namely `getopt` (an equivalent for `getopt()` from the C language) and the deprecated `optparse`. Note also that `argparse` is based on `optparse`, and therefore very similar in terms of usage.

1 Concepts

Let's show the sort of functionality that we are going to explore in this introductory tutorial by making use of the `ls` command:

```
$ ls
cpython  devguide  prog.py  pypy  rm-unused-function.patch
$ ls pypy
ctypes_configure  demo  dotviewer  include  lib_pypy  lib-python ...
$ ls -l
total 20
drwxr-xr-x 19 wena wena 4096 Feb 18 18:51 cpython
drwxr-xr-x  4 wena wena 4096 Feb  8 12:04 devguide
-rwxr-xr-x  1 wena wena  535 Feb 19 00:05 prog.py
drwxr-xr-x 14 wena wena 4096 Feb  7 00:59 pypy
-rw-r--r--  1 wena wena  741 Feb 18 01:01 rm-unused-function.patch
$ ls --help
Usage: ls [OPTION]... [FILE]...
List information about the FILEs (the current directory by default).
Sort entries alphabetically if none of -cftuvSUX nor --sort is specified.
...
```

A few concepts we can learn from the four commands:

- The `ls` command is useful when run without any options at all. It defaults to displaying the contents of the current directory.
- If we want beyond what it provides by default, we tell it a bit more. In this case, we want it to display a different directory, `pypy`. What we did is specify what is known as a positional argument. It's named so because the program should know what to do with the value, solely based on where it appears on the command line. This concept is more relevant to a command like `cp`, whose most basic usage is `cp SRC DEST`. The first position is *what you want copied,* and the second position is *where you want it copied to.*
- Now, say we want to change behaviour of the program. In our example, we display more info for each file instead of just showing the file names. The `-l` in that case is known as an optional argument.
- That's a snippet of the help text. It's very useful in that you can come across a program you have never used before, and can figure out how it works simply by reading its help text.

2 The basics

Let us start with a very simple example which does (almost) nothing:

```
import argparse
parser = argparse.ArgumentParser()
parser.parse_args()
```

Following is a result of running the code:

```
$ python prog.py
$ python prog.py --help
usage: prog.py [-h]

optional arguments:
  -h, --help  show this help message and exit
$ python prog.py --verbose
```

```
usage: prog.py [-h]
prog.py: error: unrecognized arguments: --verbose
$ python prog.py foo
usage: prog.py [-h]
prog.py: error: unrecognized arguments: foo
```

Here is what is happening:

- Running the script without any options results in nothing displayed to stdout. Not so useful.
- The second one starts to display the usefulness of the `argparse` module. We have done almost nothing, but already we get a nice help message.
- The `--help` option, which can also be shortened to `-h`, is the only option we get for free (i.e. no need to specify it). Specifying anything else results in an error. But even then, we do get a useful usage message, also for free.

3 Introducing Positional arguments

An example:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo")
args = parser.parse_args()
print args.echo
```

And running the code:

```
$ python prog.py
usage: prog.py [-h] echo
prog.py: error: the following arguments are required: echo
$ python prog.py --help
usage: prog.py [-h] echo

positional arguments:
  echo

optional arguments:
  -h, --help  show this help message and exit
$ python prog.py foo
foo
```

Here is what's happening:

- We've added the `add_argument()` method, which is what we use to specify which command-line options the program is willing to accept. In this case, I've named it `echo` so that it's in line with its function.
- Calling our program now requires us to specify an option.
- The `parse_args()` method actually returns some data from the options specified, in this case, `echo`.
- The variable is some form of 'magic' that `argparse` performs for free (i.e. no need to specify which variable that value is stored in). You will also notice that its name matches the string argument given to the method, `echo`.

Note however that, although the help display looks nice and all, it currently is not as helpful as it can be. For example we see that we got `echo` as a positional argument, but we don't know what it does, other than by guessing or by reading the source code. So, let's make it a bit more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("echo", help="echo the string you use here")
args = parser.parse_args()
print args.echo
```

And we get:

```
$ python prog.py -h
usage: prog.py [-h] echo

positional arguments:
  echo        echo the string you use here

optional arguments:
  -h, --help  show this help message and exit
```

Now, how about doing something even more useful:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number")
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 5, in <module>
    print args.square**2
TypeError: unsupported operand type(s) for ** or pow(): 'str' and 'int'
```

That didn't go so well. That's because `argparse` treats the options we give it as strings, unless we tell it otherwise. So, let's tell `argparse` to treat that input as an integer:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", help="display a square of a given number",
                    type=int)
args = parser.parse_args()
print args.square**2
```

Following is a result of running the code:

```
$ python prog.py 4
16
$ python prog.py four
usage: prog.py [-h] square
prog.py: error: argument square: invalid int value: 'four'
```

That went well. The program now even helpfully quits on bad illegal input before proceeding.

4 Introducing Optional arguments

So far we have been playing with positional arguments. Let us have a look on how to add optional ones:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbosity", help="increase output verbosity")
args = parser.parse_args()
if args.verbosity:
    print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbosity 1
verbosity turned on
$ python prog.py
$ python prog.py --help
usage: prog.py [-h] [--verbosity VERBOSITY]

optional arguments:
  -h, --help            show this help message and exit
  --verbosity VERBOSITY
                        increase output verbosity
$ python prog.py --verbosity
usage: prog.py [-h] [--verbosity VERBOSITY]
prog.py: error: argument --verbosity: expected one argument
```

Here is what is happening:

- The program is written so as to display something when `--verbosity` is specified and display nothing when not.
- To show that the option is actually optional, there is no error when running the program without it. Note that by default, if an optional argument isn't used, the relevant variable, in this case `args.verbosity`, is given `None` as a value, which is the reason it fails the truth test of the `if` statement.
- The help message is a bit different.
- When using the `--verbosity` option, one must also specify some value, any value.

The above example accepts arbitrary integer values for `--verbosity`, but for our simple program, only two values are actually useful, `True` or `False`. Let's modify the code accordingly:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
   print "verbosity turned on"
```

And the output:

```
$ python prog.py --verbose
verbosity turned on
$ python prog.py --verbose 1
usage: prog.py [-h] [--verbose]
prog.py: error: unrecognized arguments: 1
$ python prog.py --help
usage: prog.py [-h] [--verbose]

optional arguments:
  -h, --help  show this help message and exit
  --verbose   increase output verbosity
```

Here is what is happening:

- The option is now more of a flag than something that requires a value. We even changed the name of the option to match that idea. Note that we now specify a new keyword, `action`, and give it the value `"store_true"`. This means that, if the option is specified, assign the value `True` to `args.verbose`. Not specifying it implies `False`.
- It complains when you specify a value, in true spirit of what flags actually are.
- Notice the different help text.

4.1 Short options

If you are familiar with command line usage, you will notice that I haven't yet touched on the topic of short versions of the options. It's quite simple:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("-v", "--verbose", help="increase output verbosity",
                    action="store_true")
args = parser.parse_args()
if args.verbose:
    print "verbosity turned on"
```

And here goes:

```
$ python prog.py -v
verbosity turned on
$ python prog.py --help
usage: prog.py [-h] [-v]

optional arguments:
  -h, --help     show this help message and exit
  -v, --verbose  increase output verbosity
```

Note that the new ability is also reflected in the help text.

5 Combining Positional and Optional arguments

Our program keeps growing in complexity:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbose", action="store_true",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbose:
    print "the square of {} equals {}".format(args.square, answer)
else:
    print answer
```

And now the output:

```
$ python prog.py
usage: prog.py [-h] [-v] square
prog.py: error: the following arguments are required: square
$ python prog.py 4
16
$ python prog.py 4 --verbose
the square of 4 equals 16
$ python prog.py --verbose 4
the square of 4 equals 16
```

- We've brought back a positional argument, hence the complaint.
- Note that the order does not matter.

How about we give this program of ours back the ability to have multiple verbosity values, and actually get to use them:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And the output:

```
$ python prog.py 4
16
$ python prog.py 4 -v
usage: prog.py [-h] [-v VERBOSITY] square
prog.py: error: argument -v/--verbosity: expected one argument
$ python prog.py 4 -v 1
4^2 == 16
$ python prog.py 4 -v 2
the square of 4 equals 16
$ python prog.py 4 -v 3
16
```

These all look good except the last one, which exposes a bug in our program. Let's fix it by restricting the values the `--verbosity` option can accept:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", type=int, choices=[0, 1, 2],
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
```

```
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And the output:

```
$ python prog.py 4 -v 3
usage: prog.py [-h] [-v {0,1,2}] square
prog.py: error: argument -v/--verbosity: invalid choice: 3 (choose from 0, 1, 2)
$ python prog.py 4 -h
usage: prog.py [-h] [-v {0,1,2}] square

positional arguments:
  square                display a square of a given number

optional arguments:
  -h, --help            show this help message and exit
  -v {0,1,2}, --verbosity {0,1,2}
                        increase output verbosity
```

Note that the change also reflects both in the error message as well as the help string.

Now, let's use a different approach of playing with verbosity, which is pretty common. It also matches the way the CPython executable handles its own verbosity argument (check the output of `python --help`):

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display the square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity == 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity == 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

We have introduced another action, "count", to count the number of occurrences of a specific optional arguments:

```
$ python prog.py 4
16
$ python prog.py 4 -v
4^2 == 16
$ python prog.py 4 -vv
the square of 4 equals 16
$ python prog.py 4 --verbosity --verbosity
the square of 4 equals 16
$ python prog.py 4 -v 1
usage: prog.py [-h] [-v] square
prog.py: error: unrecognized arguments: 1
$ python prog.py 4 -h
usage: prog.py [-h] [-v] square
```

```
positional arguments:
  square           display a square of a given number

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity  increase output verbosity
$ python prog.py 4 -vvv
16
```

- Yes, it's now more of a flag (similar to `action="store_true"`) in the previous version of our script. That should explain the complaint.
- It also behaves similar to "store_true" action.
- Now here's a demonstration of what the "count" action gives. You've probably seen this sort of usage before.
- And, just like the "store_true" action, if you don't specify the `-v` flag, that flag is considered to have `None` value.
- As should be expected, specifying the long form of the flag, we should get the same output.
- Sadly, our help output isn't very informative on the new ability our script has acquired, but that can always be fixed by improving the documentation for our script (e.g. via the `help` keyword argument).
- That last output exposes a bug in our program.

Let's fix:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count",
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2

# bugfix: replace == with >=
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

And this is what it gives:

```
$ python prog.py 4 -vvv
the square of 4 equals 16
$ python prog.py 4 -vvvv
the square of 4 equals 16
$ python prog.py 4
Traceback (most recent call last):
  File "prog.py", line 11, in <module>
    if args.verbosity >= 2:
TypeError: unorderable types: NoneType() >= int()
```

- First output went well, and fixes the bug we had before. That is, we want any value >= 2 to be as verbose as possible.

- Third output not so good.

Let's fix that bug:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("square", type=int,
                    help="display a square of a given number")
parser.add_argument("-v", "--verbosity", action="count", default=0,
                    help="increase output verbosity")
args = parser.parse_args()
answer = args.square**2
if args.verbosity >= 2:
    print "the square of {} equals {}".format(args.square, answer)
elif args.verbosity >= 1:
    print "{}^2 == {}".format(args.square, answer)
else:
    print answer
```

We've just introduced yet another keyword, `default`. We've set it to `0` in order to make it comparable to the other int values. Remember that by default, if an optional argument isn't specified, it gets the `None` value, and that cannot be compared to an int value (hence the `TypeError` exception).

And:

```
$ python prog.py 4
16
```

You can go quite far just with what we've learned so far, and we have only scratched the surface. The `argparse` module is very powerful, and we'll explore a bit more of it before we end this tutorial.

6 Getting a little more advanced

What if we wanted to expand our tiny program to perform other powers, not just squares:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
elif args.verbosity >= 1:
    print "{}^{} == {}".format(args.x, args.y, answer)
else:
    print answer
```

Output:

```
$ python prog.py
usage: prog.py [-h] [-v] x y
prog.py: error: the following arguments are required: x, y
$ python prog.py -h
usage: prog.py [-h] [-v] x y
```

```
positional arguments:
  x                the base
  y                the exponent

optional arguments:
  -h, --help       show this help message and exit
  -v, --verbosity
$ python prog.py 4 2 -v
4^2 == 16
```

Notice that so far we've been using verbosity level to *change* the text that gets displayed. The following example instead uses verbosity level to display *more* text instead:

```
import argparse
parser = argparse.ArgumentParser()
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
parser.add_argument("-v", "--verbosity", action="count", default=0)
args = parser.parse_args()
answer = args.x**args.y
if args.verbosity >= 2:
    print "Running '{}'".format(__file__)
if args.verbosity >= 1:
    print "{}^{} ==".format(args.x, args.y),
print answer
```

Output:

```
$ python prog.py 4 2
16
$ python prog.py 4 2 -v
4^2 == 16
$ python prog.py 4 2 -vv
Running 'prog.py'
4^2 == 16
```

6.1 Conflicting options

So far, we have been working with two methods of an `argparse.ArgumentParser` instance. Let's introduce a third one, `add_mutually_exclusive_group()`. It allows for us to specify options that conflict with each other. Let's also change the rest of the program so that the new functionality makes more sense: we'll introduce the `--quiet` option, which will be the opposite of the `--verbose` one:

```
import argparse

parser = argparse.ArgumentParser()
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print answer
```

```
elif args.verbose:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
else:
    print "{}^{} == {}".format(args.x, args.y, answer)
```

Our program is now simpler, and we've lost some functionality for the sake of demonstration. Anyways, here's the output:

```
$ python prog.py 4 2
4^2 == 16
$ python prog.py 4 2 -q
16
$ python prog.py 4 2 -v
4 to the power 2 equals 16
$ python prog.py 4 2 -vq
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
$ python prog.py 4 2 -v --quiet
usage: prog.py [-h] [-v | -q] x y
prog.py: error: argument -q/--quiet: not allowed with argument -v/--verbose
```

That should be easy to follow. I've added that last output so you can see the sort of flexibility you get, i.e. mixing long form options with short form ones.

Before we conclude, you probably want to tell your users the main purpose of your program, just in case they don't know:

```
import argparse

parser = argparse.ArgumentParser(description="calculate X to the power of Y")
group = parser.add_mutually_exclusive_group()
group.add_argument("-v", "--verbose", action="store_true")
group.add_argument("-q", "--quiet", action="store_true")
parser.add_argument("x", type=int, help="the base")
parser.add_argument("y", type=int, help="the exponent")
args = parser.parse_args()
answer = args.x**args.y

if args.quiet:
    print answer
elif args.verbose:
    print "{} to the power {} equals {}".format(args.x, args.y, answer)
else:
    print "{}^{} == {}".format(args.x, args.y, answer)
```

Note that slight difference in the usage text. Note the `[-v | -q]`, which tells us that we can either use `-v` or `-q`, but not both at the same time:

```
$ python prog.py --help
usage: prog.py [-h] [-v | -q] x y

calculate X to the power of Y

positional arguments:
  x              the base
  y              the exponent

optional arguments:
```

```
  -h, --help     show this help message and exit
  -v, --verbose
  -q, --quiet
```

7 Conclusion

The `argparse` module offers a lot more than shown here. Its docs are quite detailed and thorough, and full of examples. Having gone through this tutorial, you should easily digest them without feeling overwhelmed.

Porting Extension Modules to Python 3

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

author Benjamin Peterson

Abstract

Although changing the C-API was not one of Python 3's objectives, the many Python-level changes made leaving Python 2's API intact impossible. In fact, some changes such as `int()` and `long()` unification are more obvious on the C level. This document endeavors to document incompatibilities and how they can be worked around.

1 Conditional compilation

The easiest way to compile only some code for Python 3 is to check if `PY_MAJOR_VERSION` is greater than or equal to 3.

```
#if PY_MAJOR_VERSION >= 3
#define IS_PY3K
#endif
```

API functions that are not present can be aliased to their equivalents within conditional blocks.

2 Changes to Object APIs

Python 3 merged together some types with similar functions while cleanly separating others.

2.1 str/unicode Unification

Python 3's `str()` type is equivalent to Python 2's `unicode()`; the C functions are called `PyUnicode_*` for both. The old 8-bit string type has become `bytes()`, with C functions called `PyBytes_*`. Python 2.6 and later provide a compatibility header, `bytesobject.h`, mapping `PyBytes` names to `PyString` ones. For best compatibility with Python 3, `PyUnicode` should be used for textual data and `PyBytes` for binary data. It's also important to remember that `PyBytes` and `PyUnicode` in Python 3 are not interchangeable like `PyString` and `PyUnicode` are in Python 2. The following example shows best practices with regards to `PyUnicode`, `PyString`, and `PyBytes`.

```
#include "stdlib.h"
#include "Python.h"
#include "bytesobject.h"

/* text example */
static PyObject *
say_hello(PyObject *self, PyObject *args) {
    PyObject *name, *result;

    if (!PyArg_ParseTuple(args, "U:say_hello", &name))
        return NULL;

    result = PyUnicode_FromFormat("Hello, %S!", name);
    return result;
}

/* just a forward */
static char * do_encode(PyObject *);

/* bytes example */
static PyObject *
encode_object(PyObject *self, PyObject *args) {
    char *encoded;
    PyObject *result, *myobj;

    if (!PyArg_ParseTuple(args, "O:encode_object", &myobj))
        return NULL;

    encoded = do_encode(myobj);
    if (encoded == NULL)
        return NULL;
    result = PyBytes_FromString(encoded);
    free(encoded);
```

```
    return result;
}
```

2.2 long/int Unification

Python 3 has only one integer type, `int()`. But it actually corresponds to Python 2's `long()` type—the `int()` type used in Python 2 was removed. In the C-API, `PyInt_*` functions are replaced by their `PyLong_*` equivalents.

3 Module initialization and state

Python 3 has a revamped extension module initialization system. (See **PEP 3121**.) Instead of storing module state in globals, they should be stored in an interpreter specific structure. Creating modules that act correctly in both Python 2 and Python 3 is tricky. The following simple example demonstrates how.

```
#include "Python.h"

struct module_state {
    PyObject *error;
};

#if PY_MAJOR_VERSION >= 3
#define GETSTATE(m) ((struct module_state*)PyModule_GetState(m))
#else
#define GETSTATE(m) (&_state)
static struct module_state _state;
#endif

static PyObject *
error_out(PyObject *m) {
    struct module_state *st = GETSTATE(m);
    PyErr_SetString(st->error, "something bad happened");
    return NULL;
}

static PyMethodDef myextension_methods[] = {
    {"error_out", (PyCFunction)error_out, METH_NOARGS, NULL},
    {NULL, NULL}
};

#if PY_MAJOR_VERSION >= 3

static int myextension_traverse(PyObject *m, visitproc visit, void *arg) {
    Py_VISIT(GETSTATE(m)->error);
    return 0;
}

static int myextension_clear(PyObject *m) {
    Py_CLEAR(GETSTATE(m)->error);
    return 0;
}


static struct PyModuleDef moduledef = {
```

```
        PyModuleDef_HEAD_INIT,
        "myextension",
        NULL,
        sizeof(struct module_state),
        myextension_methods,
        NULL,
        myextension_traverse,
        myextension_clear,
        NULL
};

#define INITERROR return NULL

PyMODINIT_FUNC
PyInit_myextension(void)

#else
#define INITERROR return

void
initmyextension(void)
#endif
{
#if PY_MAJOR_VERSION >= 3
    PyObject *module = PyModule_Create(&moduledef);
#else
    PyObject *module = Py_InitModule("myextension", myextension_methods);
#endif

    if (module == NULL)
        INITERROR;
    struct module_state *st = GETSTATE(module);

    st->error = PyErr_NewException("myextension.Error", NULL, NULL);
    if (st->error == NULL) {
        Py_DECREF(module);
        INITERROR;
    }

#if PY_MAJOR_VERSION >= 3
    return module;
#endif
}
```

4 CObject replaced with Capsule

The `Capsule` object was introduced in Python 3.1 and 2.7 to replace `CObject`. CObjects were useful, but the `CObject` API was problematic: it didn't permit distinguishing between valid CObjects, which allowed mismatched CObjects to crash the interpreter, and some of its APIs relied on undefined behavior in C. (For further reading on the rationale behind Capsules, please see bpo-5630.)

If you're currently using CObjects, and you want to migrate to 3.1 or newer, you'll need to switch to Capsules. `CObject` was deprecated in 3.1 and 2.7 and completely removed in Python 3.2. If you only support 2.7, or 3.1 and above, you can simply switch to `Capsule`. If you need to support Python 3.0, or versions of Python earlier than 2.7, you'll have to support both CObjects and Capsules. (Note that Python 3.0 is no longer supported, and it is not recommended for production use.)

The following example header file `capsulethunk.h` may solve the problem for you. Simply write your code against the `Capsule` API and include this header file after `Python.h`. Your code will automatically use Capsules in versions of Python with Capsules, and switch to COjects when Capsules are unavailable.

`capsulethunk.h` simulates Capsules using COjects. However, `CObject` provides no place to store the capsule's "name". As a result the simulated `Capsule` objects created by `capsulethunk.h` behave slightly differently from real Capsules. Specifically:

- The name parameter passed in to `PyCapsule_New()` is ignored.
- The name parameter passed in to `PyCapsule_IsValid()` and `PyCapsule_GetPointer()` is ignored, and no error checking of the name is performed.
- `PyCapsule_GetName()` always returns NULL.
- `PyCapsule_SetName()` always raises an exception and returns failure. (Since there's no way to store a name in a CObject, noisy failure of `PyCapsule_SetName()` was deemed preferable to silent failure here. If this is inconvenient, feel free to modify your local copy as you see fit.)

You can find `capsulethunk.h` in the Python source distribution as Doc/includes/capsulethunk.h. We also include it here for your convenience:

```
#ifndef __CAPSULETHUNK_H
#define __CAPSULETHUNK_H

#if (    (PY_VERSION_HEX <  0x02070000) \
     || ((PY_VERSION_HEX >= 0x03000000) \
      && (PY_VERSION_HEX <  0x03010000)) )

#define __PyCapsule_GetField(capsule, field, default_value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field) \
        : (default_value) \
    ) \

#define __PyCapsule_SetField(capsule, field, value) \
    ( PyCapsule_CheckExact(capsule) \
        ? (((PyCObject *)capsule)->field = value), 1 \
        : 0 \
    ) \


#define PyCapsule_Type PyCObject_Type

#define PyCapsule_CheckExact(capsule) (PyCObject_Check(capsule))
#define PyCapsule_IsValid(capsule, name) (PyCObject_Check(capsule))


#define PyCapsule_New(pointer, name, destructor) \
    (PyCObject_FromVoidPtr(pointer, destructor))


#define PyCapsule_GetPointer(capsule, name) \
    (PyCObject_AsVoidPtr(capsule))

/* Don't call PyCObject_SetPointer here, it fails if there's a destructor */
#define PyCapsule_SetPointer(capsule, pointer) \
    __PyCapsule_SetField(capsule, cobject, pointer)


#define PyCapsule_GetDestructor(capsule) \
```

```
    __PyCapsule_GetField(capsule, destructor)

#define PyCapsule_SetDestructor(capsule, dtor) \
    __PyCapsule_SetField(capsule, destructor, dtor)


/*
 * Sorry, there's simply no place
 * to store a Capsule "name" in a CObject.
 */
#define PyCapsule_GetName(capsule) NULL

static int
PyCapsule_SetName(PyObject *capsule, const char *unused)
{
    unused = unused;
    PyErr_SetString(PyExc_NotImplementedError,
        "can't use PyCapsule_SetName with CObjects");
    return 1;
}



#define PyCapsule_GetContext(capsule) \
    __PyCapsule_GetField(capsule, descr)

#define PyCapsule_SetContext(capsule, context) \
    __PyCapsule_SetField(capsule, descr, context)


static void *
PyCapsule_Import(const char *name, int no_block)
{
    PyObject *object = NULL;
    void *return_value = NULL;
    char *trace;
    size_t name_length = (strlen(name) + 1) * sizeof(char);
    char *name_dup = (char *)PyMem_MALLOC(name_length);

    if (!name_dup) {
        return NULL;
    }

    memcpy(name_dup, name, name_length);

    trace = name_dup;
    while (trace) {
        char *dot = strchr(trace, '.');
        if (dot) {
            *dot++ = '\0';
        }

        if (object == NULL) {
            if (no_block) {
                object = PyImport_ImportModuleNoBlock(trace);
            } else {
                object = PyImport_ImportModule(trace);
                if (!object) {
```

```
                    PyErr_Format(PyExc_ImportError,
                        "PyCapsule_Import could not "
                        "import module \"%s\"", trace);
                }
            }
        } else {
            PyObject *object2 = PyObject_GetAttrString(object, trace);
            Py_DECREF(object);
            object = object2;
        }
        if (!object) {
            goto EXIT;
        }

        trace = dot;
    }

    if (PyCObject_Check(object)) {
        PyCObject *cobject = (PyCObject *)object;
        return_value = cobject->cobject;
    } else {
        PyErr_Format(PyExc_AttributeError,
            "PyCapsule_Import \"%s\" is not valid",
            name);
    }

EXIT:
    Py_XDECREF(object);
    if (name_dup) {
        PyMem_FREE(name_dup);
    }
    return return_value;
}

#endif /* #if PY_VERSION_HEX < 0x02070000 */

#endif /* __CAPSULETHUNK_H */
```

5 Other options

If you are writing a new extension module, you might consider Cython. It translates a Python-like language to C. The extension modules it creates are compatible with Python 3 and Python 2.

Index

P

Curses Programming with Python

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author A.M. Kuchling, Eric S. Raymond

Release 2.03

Abstract

This document describes how to write text-mode programs with Python 2.x, using the `curses` extension module to control the display.

1 What is curses?

The curses library supplies a terminal-independent screen-painting and keyboard-handling facility for text-based terminals; such terminals include VT100s, the Linux console, and the simulated terminal provided by X11 programs such as xterm and rxvt. Display terminals support various control codes to perform common

operations such as moving the cursor, scrolling the screen, and erasing areas. Different terminals use widely differing codes, and often have their own minor quirks.

In a world of X displays, one might ask "why bother"? It's true that character-cell display terminals are an obsolete technology, but there are niches in which being able to do fancy things with them are still valuable. One is on small-footprint or embedded Unixes that don't carry an X server. Another is for tools like OS installers and kernel configurators that may have to run before X is available.

The curses library hides all the details of different terminals, and provides the programmer with an abstraction of a display, containing multiple non-overlapping windows. The contents of a window can be changed in various ways—adding text, erasing it, changing its appearance—and the curses library will automagically figure out what control codes need to be sent to the terminal to produce the right output.

The curses library was originally written for BSD Unix; the later System V versions of Unix from AT&T added many enhancements and new functions. BSD curses is no longer maintained, having been replaced by ncurses, which is an open-source implementation of the AT&T interface. If you're using an open-source Unix such as Linux or FreeBSD, your system almost certainly uses ncurses. Since most current commercial Unix versions are based on System V code, all the functions described here will probably be available. The older versions of curses carried by some proprietary Unixes may not support everything, though.

No one has made a Windows port of the curses module. On a Windows platform, try the Console module written by Fredrik Lundh. The Console module provides cursor-addressable text output, plus full support for mouse and keyboard input, and is available from http://effbot.org/zone/console-index.htm.

1.1 The Python curses module

Thy Python module is a fairly simple wrapper over the C functions provided by curses; if you're already familiar with curses programming in C, it's really easy to transfer that knowledge to Python. The biggest difference is that the Python interface makes things simpler, by merging different C functions such as `addstr()`, `mvaddstr()`, `mvwaddstr()`, into a single `addstr()` method. You'll see this covered in more detail later.

This HOWTO is simply an introduction to writing text-mode programs with curses and Python. It doesn't attempt to be a complete guide to the curses API; for that, see the Python library guide's section on ncurses, and the C manual pages for ncurses. It will, however, give you the basic ideas.

2 Starting and ending a curses application

Before doing anything, curses must be initialized. This is done by calling the `initscr()` function, which will determine the terminal type, send any required setup codes to the terminal, and create various internal data structures. If successful, `initscr()` returns a window object representing the entire screen; this is usually called `stdscr`, after the name of the corresponding C variable.

```
import curses
stdscr = curses.initscr()
```

Usually curses applications turn off automatic echoing of keys to the screen, in order to be able to read keys and only display them under certain circumstances. This requires calling the `noecho()` function.

```
curses.noecho()
```

Applications will also commonly need to react to keys instantly, without requiring the Enter key to be pressed; this is called cbreak mode, as opposed to the usual buffered input mode.

```
curses.cbreak()
```

Terminals usually return special keys, such as the cursor keys or navigation keys such as Page Up and Home, as a multibyte escape sequence. While you could write your application to expect such sequences and process them accordingly, curses can do it for you, returning a special value such as `curses.KEY_LEFT`. To get curses to do the job, you'll have to enable keypad mode.

```
stdscr.keypad(1)
```

Terminating a curses application is much easier than starting one. You'll need to call

```
curses.nocbreak(); stdscr.keypad(0); curses.echo()
```

to reverse the curses-friendly terminal settings. Then call the `endwin()` function to restore the terminal to its original operating mode.

```
curses.endwin()
```

A common problem when debugging a curses application is to get your terminal messed up when the application dies without restoring the terminal to its previous state. In Python this commonly happens when your code is buggy and raises an uncaught exception. Keys are no longer echoed to the screen when you type them, for example, which makes using the shell difficult.

In Python you can avoid these complications and make debugging much easier by importing the `curses.wrapper()` function. It takes a callable and does the initializations described above, also initializing colors if color support is present. It then runs your provided callable and finally deinitializes appropriately. The callable is called inside a try-catch clause which catches exceptions, performs curses deinitialization, and then passes the exception upwards. Thus, your terminal won't be left in a funny state on exception.

3 Windows and Pads

Windows are the basic abstraction in curses. A window object represents a rectangular area of the screen, and supports various methods to display text, erase it, allow the user to input strings, and so forth.

The `stdscr` object returned by the `initscr()` function is a window object that covers the entire screen. Many programs may need only this single window, but you might wish to divide the screen into smaller windows, in order to redraw or clear them separately. The `newwin()` function creates a new window of a given size, returning the new window object.

```
begin_x = 20; begin_y = 7
height = 5; width = 40
win = curses.newwin(height, width, begin_y, begin_x)
```

A word about the coordinate system used in curses: coordinates are always passed in the order *y,x*, and the top-left corner of a window is coordinate (0,0). This breaks a common convention for handling coordinates, where the *x* coordinate usually comes first. This is an unfortunate difference from most other computer applications, but it's been part of curses since it was first written, and it's too late to change things now.

When you call a method to display or erase text, the effect doesn't immediately show up on the display. This is because curses was originally written with slow 300-baud terminal connections in mind; with these terminals, minimizing the time required to redraw the screen is very important. This lets curses accumulate changes to the screen, and display them in the most efficient manner. For example, if your program displays some characters in a window, and then clears the window, there's no need to send the original characters because they'd never be visible.

Accordingly, curses requires that you explicitly tell it to redraw windows, using the `refresh()` method of window objects. In practice, this doesn't really complicate programming with curses much. Most programs go into a flurry of activity, and then pause waiting for a keypress or some other action on the part of the

user. All you have to do is to be sure that the screen has been redrawn before pausing to wait for user input, by simply calling `stdscr.refresh()` or the `refresh()` method of some other relevant window.

A pad is a special case of a window; it can be larger than the actual display screen, and only a portion of it displayed at a time. Creating a pad simply requires the pad's height and width, while refreshing a pad requires giving the coordinates of the on-screen area where a subsection of the pad will be displayed.

```
pad = curses.newpad(100, 100)
#  These loops fill the pad with letters; this is
# explained in the next section
for y in range(0, 100):
    for x in range(0, 100):
        try:
            pad.addch(y,x, ord('a') + (x*x+y*y) % 26)
        except curses.error:
            pass

#  Displays a section of the pad in the middle of the screen
pad.refresh(0,0, 5,5, 20,75)
```

The `refresh()` call displays a section of the pad in the rectangle extending from coordinate (5,5) to coordinate (20,75) on the screen; the upper left corner of the displayed section is coordinate (0,0) on the pad. Beyond that difference, pads are exactly like ordinary windows and support the same methods.

If you have multiple windows and pads on screen there is a more efficient way to go, which will prevent annoying screen flicker at refresh time. Use the `noutrefresh()` method of each window to update the data structure representing the desired state of the screen; then change the physical screen to match the desired state in one go with the function `doupdate()`. The normal `refresh()` method calls `doupdate()` as its last act.

4 Displaying Text

From a C programmer's point of view, curses may sometimes look like a twisty maze of functions, all subtly different. For example, `addstr()` displays a string at the current cursor location in the `stdscr` window, while `mvaddstr()` moves to a given y,x coordinate first before displaying the string. `waddstr()` is just like `addstr()`, but allows specifying a window to use, instead of using `stdscr` by default. `mvwaddstr()` follows similarly.

Fortunately the Python interface hides all these details; `stdscr` is a window object like any other, and methods like `addstr()` accept multiple argument forms. Usually there are four different forms.

Form	Description
str or *ch*	Display the string *str* or character *ch* at the current position
str or *ch*, *attr*	Display the string *str* or character *ch*, using attribute *attr* at the current position
y, *x*, *str* or *ch*	Move to position *y,x* within the window, and display *str* or *ch*
y, *x*, *str* or *ch*, *attr*	Move to position *y,x* within the window, and display *str* or *ch*, using attribute *attr*

Attributes allow displaying text in highlighted forms, such as in boldface, underline, reverse code, or in color. They'll be explained in more detail in the next subsection.

The `addstr()` function takes a Python string as the value to be displayed, while the `addch()` functions take a character, which can be either a Python string of length 1 or an integer. If it's a string, you're limited to displaying characters between 0 and 255. SVr4 curses provides constants for extension characters; these constants are integers greater than 255. For example, `ACS_PLMINUS` is a +/- symbol, and `ACS_ULCORNER` is the upper left corner of a box (handy for drawing borders).

Windows remember where the cursor was left after the last operation, so if you leave out the *y,x* coordinates, the string or character will be displayed wherever the last operation left off. You can also move the cursor with the `move(y,x)` method. Because some terminals always display a flashing cursor, you may want to ensure that the cursor is positioned in some location where it won't be distracting; it can be confusing to have the cursor blinking at some apparently random location.

If your application doesn't need a blinking cursor at all, you can call `curs_set(0)` to make it invisible. Equivalently, and for compatibility with older curses versions, there's a `leaveok(bool)` function. When *bool* is true, the curses library will attempt to suppress the flashing cursor, and you won't need to worry about leaving it in odd locations.

4.1 Attributes and Color

Characters can be displayed in different ways. Status lines in a text-based application are commonly shown in reverse video; a text viewer may need to highlight certain words. curses supports this by allowing you to specify an attribute for each cell on the screen.

An attribute is an integer, each bit representing a different attribute. You can try to display text with multiple attribute bits set, but curses doesn't guarantee that all the possible combinations are available, or that they're all visually distinct. That depends on the ability of the terminal being used, so it's safest to stick to the most commonly available attributes, listed here.

Attribute	Description
`A_BLINK`	Blinking text
`A_BOLD`	Extra bright or bold text
`A_DIM`	Half bright text
`A_REVERSE`	Reverse-video text
`A_STANDOUT`	The best highlighting mode available
`A_UNDERLINE`	Underlined text

So, to display a reverse-video status line on the top line of the screen, you could code:

```
stdscr.addstr(0, 0, "Current mode: Typing mode",
              curses.A_REVERSE)
stdscr.refresh()
```

The curses library also supports color on those terminals that provide it. The most common such terminal is probably the Linux console, followed by color xterms.

To use color, you must call the `start_color()` function soon after calling `initscr()`, to initialize the default color set (the `curses.wrapper.wrapper()` function does this automatically). Once that's done, the `has_colors()` function returns TRUE if the terminal in use can actually display color. (Note: curses uses the American spelling 'color', instead of the Canadian/British spelling 'colour'. If you're used to the British spelling, you'll have to resign yourself to misspelling it for the sake of these functions.)

The curses library maintains a finite number of color pairs, containing a foreground (or text) color and a background color. You can get the attribute value corresponding to a color pair with the `color_pair()` function; this can be bitwise-OR'ed with other attributes such as `A_REVERSE`, but again, such combinations are not guaranteed to work on all terminals.

An example, which displays a line of text using color pair 1:

```
stdscr.addstr("Pretty text", curses.color_pair(1))
stdscr.refresh()
```

As I said before, a color pair consists of a foreground and background color. `start_color()` initializes 8 basic colors when it activates color mode. They are: 0:black, 1:red, 2:green, 3:yellow, 4:blue, 5:magenta, 6:cyan,

and 7:white. The curses module defines named constants for each of these colors: `curses.COLOR_BLACK`, `curses.COLOR_RED`, and so forth.

The `init_pair(n, f, b)` function changes the definition of color pair n, to foreground color f and background color b. Color pair 0 is hard-wired to white on black, and cannot be changed.

Let's put all this together. To change color 1 to red text on a white background, you would call:

```
curses.init_pair(1, curses.COLOR_RED, curses.COLOR_WHITE)
```

When you change a color pair, any text already displayed using that color pair will change to the new colors. You can also display new text in this color with:

```
stdscr.addstr(0,0, "RED ALERT!", curses.color_pair(1))
```

Very fancy terminals can change the definitions of the actual colors to a given RGB value. This lets you change color 1, which is usually red, to purple or blue or any other color you like. Unfortunately, the Linux console doesn't support this, so I'm unable to try it out, and can't provide any examples. You can check if your terminal can do this by calling `can_change_color()`, which returns TRUE if the capability is there. If you're lucky enough to have such a talented terminal, consult your system's man pages for more information.

5 User Input

The curses library itself offers only very simple input mechanisms. Python's support adds a text-input widget that makes up some of the lack.

The most common way to get input to a window is to use its `getch()` method. `getch()` pauses and waits for the user to hit a key, displaying it if `echo()` has been called earlier. You can optionally specify a coordinate to which the cursor should be moved before pausing.

It's possible to change this behavior with the method `nodelay()`. After `nodelay(1)`, `getch()` for the window becomes non-blocking and returns `curses.ERR` (a value of -1) when no input is ready. There's also a `halfdelay()` function, which can be used to (in effect) set a timer on each `getch()`; if no input becomes available within a specified delay (measured in tenths of a second), curses raises an exception.

The `getch()` method returns an integer; if it's between 0 and 255, it represents the ASCII code of the key pressed. Values greater than 255 are special keys such as Page Up, Home, or the cursor keys. You can compare the value returned to constants such as `curses.KEY_PPAGE`, `curses.KEY_HOME`, or `curses.KEY_LEFT`. Usually the main loop of your program will look something like this:

```
while 1:
    c = stdscr.getch()
    if c == ord('p'):
        PrintDocument()
    elif c == ord('q'):
        break  # Exit the while()
    elif c == curses.KEY_HOME:
        x = y = 0
```

The `curses.ascii` module supplies ASCII class membership functions that take either integer or 1-character-string arguments; these may be useful in writing more readable tests for your command interpreters. It also supplies conversion functions that take either integer or 1-character-string arguments and return the same type. For example, `curses.ascii.ctrl()` returns the control character corresponding to its argument.

There's also a method to retrieve an entire string, `getstr()`. It isn't used very often, because its functionality is quite limited; the only editing keys available are the backspace key and the Enter key, which terminates the string. It can optionally be limited to a fixed number of characters.

```
curses.echo()             # Enable echoing of characters

# Get a 15-character string, with the cursor on the top line
s = stdscr.getstr(0,0, 15)
```

The Python `curses.textpad` module supplies something better. With it, you can turn a window into a text box that supports an Emacs-like set of keybindings. Various methods of `Textbox` class support editing with input validation and gathering the edit results either with or without trailing spaces. See the library documentation on `curses.textpad` for the details.

6 For More Information

This HOWTO didn't cover some advanced topics, such as screen-scraping or capturing mouse events from an xterm instance. But the Python library page for the curses modules is now pretty complete. You should browse it next.

If you're in doubt about the detailed behavior of any of the ncurses entry points, consult the manual pages for your curses implementation, whether it's ncurses or a proprietary Unix vendor's. The manual pages will document any quirks, and provide complete lists of all the functions, attributes, and `ACS_*` characters available to you.

Because the curses API is so large, some functions aren't supported in the Python interface, not because they're difficult to implement, but because no one has needed them yet. Feel free to add them and then submit a patch. Also, we don't yet have support for the menu library associated with ncurses; feel free to add that.

If you write an interesting little program, feel free to contribute it as another demo. We can always use more of them!

The ncurses FAQ: http://invisible-island.net/ncurses/ncurses.faq.html

Descriptor HowTo Guide

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Raymond Hettinger

Contact <python at rcn dot com>

Contents

1 Abstract

Defines descriptors, summarizes the protocol, and shows how descriptors are called. Examines a custom descriptor and several built-in python descriptors including functions, properties, static methods, and class methods. Shows how each works by giving a pure Python equivalent and a sample application.

Learning about descriptors not only provides access to a larger toolset, it creates a deeper understanding of how Python works and an appreciation for the elegance of its design.

2 Definition and Introduction

In general, a descriptor is an object attribute with "binding behavior", one whose attribute access has been overridden by methods in the descriptor protocol. Those methods are `__get__()`, `__set__()`, and `__delete__()`. If any of those methods are defined for an object, it is said to be a descriptor.

The default behavior for attribute access is to get, set, or delete the attribute from an object's dictionary. For instance, `a.x` has a lookup chain starting with `a.__dict__['x']`, then `type(a).__dict__['x']`, and continuing through the base classes of `type(a)` excluding metaclasses. If the looked-up value is an object defining one of the descriptor methods, then Python may override the default behavior and invoke the descriptor method instead. Where this occurs in the precedence chain depends on which descriptor methods were defined. Note that descriptors are only invoked for new style objects or classes (a class is new style if it inherits from `object` or `type`).

Descriptors are a powerful, general purpose protocol. They are the mechanism behind properties, methods, static methods, class methods, and `super()`. They are used throughout Python itself to implement the new style classes introduced in version 2.2. Descriptors simplify the underlying C-code and offer a flexible set of new tools for everyday Python programs.

3 Descriptor Protocol

```
descr.__get__(self, obj, type=None) --> value

descr.__set__(self, obj, value) --> None

descr.__delete__(self, obj) --> None
```

That is all there is to it. Define any of these methods and an object is considered a descriptor and can override default behavior upon being looked up as an attribute.

If an object defines both `__get__()` and `__set__()`, it is considered a data descriptor. Descriptors that only define `__get__()` are called non-data descriptors (they are typically used for methods but other uses are possible).

Data and non-data descriptors differ in how overrides are calculated with respect to entries in an instance's dictionary. If an instance's dictionary has an entry with the same name as a data descriptor, the data descriptor takes precedence. If an instance's dictionary has an entry with the same name as a non-data descriptor, the dictionary entry takes precedence.

To make a read-only data descriptor, define both `__get__()` and `__set__()` with the `__set__()` raising an `AttributeError` when called. Defining the `__set__()` method with an exception raising placeholder is enough to make it a data descriptor.

4 Invoking Descriptors

A descriptor can be called directly by its method name. For example, `d.__get__(obj)`.

Alternatively, it is more common for a descriptor to be invoked automatically upon attribute access. For example, `obj.d` looks up `d` in the dictionary of `obj`. If `d` defines the method `__get__()`, then `d.__get__(obj)` is invoked according to the precedence rules listed below.

The details of invocation depend on whether `obj` is an object or a class. Either way, descriptors only work for new style objects and classes. A class is new style if it is a subclass of `object`.

For objects, the machinery is in `object.__getattribute__()` which transforms `b.x` into `type(b).__dict__['x'].__get__(b, type(b))`. The implementation works through a precedence chain that gives data descriptors priority over instance variables, instance variables priority over non-data descriptors, and assigns lowest priority to `__getattr__()` if provided. The full C implementation can be found in `PyObject_GenericGetAttr()` in Objects/object.c.

For classes, the machinery is in `type.__getattribute__()` which transforms `B.x` into `B.__dict__['x'].__get__(None, B)`. In pure Python, it looks like:

```
def __getattribute__(self, key):
    "Emulate type_getattro() in Objects/typeobject.c"
    v = object.__getattribute__(self, key)
    if hasattr(v, '__get__'):
        return v.__get__(None, self)
    return v
```

The important points to remember are:

- descriptors are invoked by the `__getattribute__()` method
- overriding `__getattribute__()` prevents automatic descriptor calls
- `__getattribute__()` is only available with new style classes and objects
- `object.__getattribute__()` and `type.__getattribute__()` make different calls to `__get__()`.
- data descriptors always override instance dictionaries.
- non-data descriptors may be overridden by instance dictionaries.

The object returned by `super()` also has a custom `__getattribute__()` method for invoking descriptors. The call `super(B, obj).m()` searches `obj.__class__.__mro__` for the base class `A` immediately following `B` and then returns `A.__dict__['m'].__get__(obj, B)`. If not a descriptor, `m` is returned unchanged. If not in the dictionary, `m` reverts to a search using `object.__getattribute__()`.

Note, in Python 2.2, `super(B, obj).m()` would only invoke `__get__()` if `m` was a data descriptor. In Python 2.3, non-data descriptors also get invoked unless an old-style class is involved. The implementation details are in `super_getattro()` in Objects/typeobject.c.

The details above show that the mechanism for descriptors is embedded in the `__getattribute__()` methods for `object`, `type`, and `super()`. Classes inherit this machinery when they derive from `object` or if they have a meta-class providing similar functionality. Likewise, classes can turn-off descriptor invocation by overriding `__getattribute__()`.

5 Descriptor Example

The following code creates a class whose objects are data descriptors which print a message for each get or set. Overriding `__getattribute__()` is alternate approach that could do this for every attribute. However, this descriptor is useful for monitoring just a few chosen attributes:

```
class RevealAccess(object):
    """A data descriptor that sets and returns values
       normally and prints a message logging their access.
    """

    def __init__(self, initval=None, name='var'):
        self.val = initval
        self.name = name

    def __get__(self, obj, objtype):
        print 'Retrieving', self.name
        return self.val

    def __set__(self, obj, val):
        print 'Updating', self.name
        self.val = val

>>> class MyClass(object):
...     x = RevealAccess(10, 'var "x"')
...     y = 5
...
>>> m = MyClass()
>>> m.x
Retrieving var "x"
10
>>> m.x = 20
Updating var "x"
>>> m.x
Retrieving var "x"
20
>>> m.y
5
```

The protocol is simple and offers exciting possibilities. Several use cases are so common that they have been packaged into individual function calls. Properties, bound and unbound methods, static methods, and class methods are all based on the descriptor protocol.

6 Properties

Calling `property()` is a succinct way of building a data descriptor that triggers function calls upon access to an attribute. Its signature is:

```
property(fget=None, fset=None, fdel=None, doc=None) -> property attribute
```

The documentation shows a typical use to define a managed attribute `x`:

```
class C(object):
    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.")
```

To see how `property()` is implemented in terms of the descriptor protocol, here is a pure Python equivalent:

```
class Property(object):
    "Emulate PyProperty_Type() in Objects/descrobject.c"

    def __init__(self, fget=None, fset=None, fdel=None, doc=None):
        self.fget = fget
        self.fset = fset
        self.fdel = fdel
        if doc is None and fget is not None:
            doc = fget.__doc__
        self.__doc__ = doc

    def __get__(self, obj, objtype=None):
        if obj is None:
            return self
        if self.fget is None:
            raise AttributeError("unreadable attribute")
        return self.fget(obj)

    def __set__(self, obj, value):
        if self.fset is None:
            raise AttributeError("can't set attribute")
        self.fset(obj, value)

    def __delete__(self, obj):
        if self.fdel is None:
            raise AttributeError("can't delete attribute")
        self.fdel(obj)

    def getter(self, fget):
        return type(self)(fget, self.fset, self.fdel, self.__doc__)

    def setter(self, fset):
        return type(self)(self.fget, fset, self.fdel, self.__doc__)

    def deleter(self, fdel):
        return type(self)(self.fget, self.fset, fdel, self.__doc__)
```

The `property()` builtin helps whenever a user interface has granted attribute access and then subsequent changes require the intervention of a method.

For instance, a spreadsheet class may grant access to a cell value through `Cell('b10').value`. Subsequent improvements to the program require the cell to be recalculated on every access; however, the programmer does not want to affect existing client code accessing the attribute directly. The solution is to wrap access to the value attribute in a property data descriptor:

```
class Cell(object):
    . . .
    def getvalue(self):
        "Recalculate the cell before returning value"
        self.recalc()
        return self._value
    value = property(getvalue)
```

7 Functions and Methods

Python's object oriented features are built upon a function based environment. Using non-data descriptors, the two are merged seamlessly.

Class dictionaries store methods as functions. In a class definition, methods are written using `def` and `lambda`, the usual tools for creating functions. The only difference from regular functions is that the first argument is reserved for the object instance. By Python convention, the instance reference is called *self* but may be called *this* or any other variable name.

To support method calls, functions include the `__get__()` method for binding methods during attribute access. This means that all functions are non-data descriptors which return bound or unbound methods depending whether they are invoked from an object or a class. In pure python, it works like this:

```
class Function(object):
    . . .
    def __get__(self, obj, objtype=None):
        "Simulate func_descr_get() in Objects/funcobject.c"
        return types.MethodType(self, obj, objtype)
```

Running the interpreter shows how the function descriptor works in practice:

```
>>> class D(object):
...     def f(self, x):
...         return x
...
>>> d = D()
>>> D.__dict__['f']  # Stored internally as a function
<function f at 0x00C45070>
>>> D.f               # Get from a class becomes an unbound method
<unbound method D.f>
>>> d.f               # Get from an instance becomes a bound method
<bound method D.f of <__main__.D object at 0x00B18C90>>
```

The output suggests that bound and unbound methods are two different types. While they could have been implemented that way, the actual C implementation of `PyMethod_Type` in Objects/classobject.c is a single object with two different representations depending on whether the `im_self` field is set or is *NULL* (the C equivalent of `None`).

Likewise, the effects of calling a method object depend on the `im_self` field. If set (meaning bound), the original function (stored in the `im_func` field) is called as expected with the first argument set to the instance. If unbound, all of the arguments are passed unchanged to the original function. The actual C implementation of `instancemethod_call()` is only slightly more complex in that it includes some type checking.

8 Static Methods and Class Methods

Non-data descriptors provide a simple mechanism for variations on the usual patterns of binding functions into methods.

To recap, functions have a `__get__()` method so that they can be converted to a method when accessed as attributes. The non-data descriptor transforms an `obj.f(*args)` call into `f(obj, *args)`. Calling `klass.f(*args)` becomes `f(*args)`.

This chart summarizes the binding and its two most useful variants:

Transformation	Called from an Object	Called from a Class
function	f(obj, *args)	f(*args)
staticmethod	f(*args)	f(*args)
classmethod	f(type(obj), *args)	f(klass, *args)

Static methods return the underlying function without changes. Calling either `c.f` or `C.f` is the equivalent of a direct lookup into `object.__getattribute__(c, "f")` or `object.__getattribute__(C, "f")`. As a result, the function becomes identically accessible from either an object or a class.

Good candidates for static methods are methods that do not reference the `self` variable.

For instance, a statistics package may include a container class for experimental data. The class provides normal methods for computing the average, mean, median, and other descriptive statistics that depend on the data. However, there may be useful functions which are conceptually related but do not depend on the data. For instance, `erf(x)` is handy conversion routine that comes up in statistical work but does not directly depend on a particular dataset. It can be called either from an object or the class: `s.erf(1.5) --> .9332` or `Sample.erf(1.5) --> .9332`.

Since staticmethods return the underlying function with no changes, the example calls are unexciting:

```
>>> class E(object):
...     def f(x):
...         print x
...     f = staticmethod(f)
...
>>> print E.f(3)
3
>>> print E().f(3)
3
```

Using the non-data descriptor protocol, a pure Python version of `staticmethod()` would look like this:

```
class StaticMethod(object):
    "Emulate PyStaticMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, objtype=None):
        return self.f
```

Unlike static methods, class methods prepend the class reference to the argument list before calling the function. This format is the same for whether the caller is an object or a class:

```
>>> class E(object):
...     def f(klass, x):
...         return klass.__name__, x
...     f = classmethod(f)
...
>>> print E.f(3)
('E', 3)
>>> print E().f(3)
('E', 3)
```

This behavior is useful whenever the function only needs to have a class reference and does not care about any underlying data. One use for classmethods is to create alternate class constructors. In Python 2.3, the classmethod `dict.fromkeys()` creates a new dictionary from a list of keys. The pure Python equivalent is:

```
class Dict(object):
    . . .
    def fromkeys(klass, iterable, value=None):
        "Emulate dict_fromkeys() in Objects/dictobject.c"
        d = klass()
        for key in iterable:
            d[key] = value
        return d
    fromkeys = classmethod(fromkeys)
```

Now a new dictionary of unique keys can be constructed like this:

```
>>> Dict.fromkeys('abracadabra')
{'a': None, 'r': None, 'b': None, 'c': None, 'd': None}
```

Using the non-data descriptor protocol, a pure Python version of `classmethod()` would look like this:

```
class ClassMethod(object):
    "Emulate PyClassMethod_Type() in Objects/funcobject.c"

    def __init__(self, f):
        self.f = f

    def __get__(self, obj, klass=None):
        if klass is None:
            klass = type(obj)
        def newfunc(*args):
            return self.f(klass, *args)
        return newfunc
```

Idioms and Anti-Idioms in Python

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Moshe Zadka

Abstract

This document can be considered a companion to the tutorial. It shows how to use Python, and even more importantly, how *not* to use Python.

1 Language Constructs You Should Not Use

While Python has relatively few gotchas compared to other languages, it still has some constructs which are only useful in corner cases, or are plain dangerous.

1.1 from module import *

Inside Function Definitions

`from module import *` is *invalid* inside function definitions. While many versions of Python do not check for the invalidity, it does not make it more valid, no more than having a smart lawyer makes a man innocent. Do not use it like that ever. Even in versions where it was accepted, it made the function execution slower, because the compiler could not be certain which names were local and which were global. In Python 2.1 this construct causes warnings, and sometimes even errors.

At Module Level

While it is valid to use `from module import *` at module level it is usually a bad idea. For one, this loses an important property Python otherwise has — you can know where each toplevel name is defined by a simple "search" function in your favourite editor. You also open yourself to trouble in the future, if some module grows additional functions or classes.

One of the most awful questions asked on the newsgroup is why this code:

```
f = open("www")
f.read()
```

does not work. Of course, it works just fine (assuming you have a file called "www".) But it does not work if somewhere in the module, the statement `from os import *` is present. The `os` module has a function called `open()` which returns an integer. While it is very useful, shadowing a builtin is one of its least useful properties.

Remember, you can never know for sure what names a module exports, so either take what you need — `from module import name1, name2`, or keep them in the module and access on a per-need basis — `import module;print module.name`.

When It Is Just Fine

There are situations in which `from module import *` is just fine:

- The interactive prompt. For example, `from math import *` makes Python an amazing scientific calculator.
- When extending a module in C with a module in Python.
- When the module advertises itself as `from import *` safe.

1.2 Unadorned `exec`, `execfile()` and friends

The word "unadorned" refers to the use without an explicit dictionary, in which case those constructs evaluate code in the *current* environment. This is dangerous for the same reasons `from import *` is dangerous — it might step over variables you are counting on and mess up things for the rest of your code. Simply do not do that.

Bad examples:

```
>>> for name in sys.argv[1:]:
>>>     exec "%s=1" % name
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         exec "s.%s=val" % var  # invalid!
>>> execfile("handler.py")
>>> handle()
```

Good examples:

```
>>> d = {}
>>> for name in sys.argv[1:]:
>>>     d[name] = 1
>>> def func(s, **kw):
>>>     for var, val in kw.items():
>>>         setattr(s, var, val)
>>> d={}
>>> execfile("handle.py", d, d)
>>> handle = d['handle']
>>> handle()
```

1.3 from module import name1, name2

This is a "don't" which is much weaker than the previous "don't"s but is still something you should not do if you don't have good reasons to do that. The reason it is usually a bad idea is because you suddenly have an object which lives in two separate namespaces. When the binding in one namespace changes, the binding in the other will not, so there will be a discrepancy between them. This happens when, for example, one module is reloaded, or changes the definition of a function at runtime.

Bad example:

```
# foo.py
a = 1

# bar.py
from foo import a
if something():
    a = 2 # danger: foo.a != a
```

Good example:

```
# foo.py
a = 1

# bar.py
import foo
if something():
    foo.a = 2
```

1.4 except:

Python has the `except:` clause, which catches all exceptions. Since *every* error in Python raises an exception, using `except:` can make many programming errors look like runtime problems, which hinders the debugging process.

The following code shows a great example of why this is bad:

```
try:
    foo = opne("file") # misspelled "open"
except:
    sys.exit("could not open file!")
```

The second line triggers a `NameError`, which is caught by the except clause. The program will exit, and the error message the program prints will make you think the problem is the readability of `"file"` when in fact the real error has nothing to do with `"file"`.

A better way to write the above is

```
try:
    foo = opne("file")
except IOError:
    sys.exit("could not open file")
```

When this is run, Python will produce a traceback showing the `NameError`, and it will be immediately apparent what needs to be fixed.

Because `except:` catches *all* exceptions, including `SystemExit`, `KeyboardInterrupt`, and `GeneratorExit` (which is not an error and should not normally be caught by user code), using a bare `except:` is almost never a good idea. In situations where you need to catch all "normal" errors, such as in a framework that runs callbacks, you can catch the base class for all normal exceptions, `Exception`. Unfortunately in Python 2.x it is possible for third-party code to raise exceptions that do not inherit from `Exception`, so in Python 2.x there are some cases where you may have to use a bare `except:` and manually re-raise the exceptions you don't want to catch.

2 Exceptions

Exceptions are a useful feature of Python. You should learn to raise them whenever something unexpected occurs, and catch them only where you can do something about them.

The following is a very popular anti-idiom

```
def get_status(file):
    if not os.path.exists(file):
        print "file not found"
        sys.exit(1)
    return open(file).readline()
```

Consider the case where the file gets deleted between the time the call to `os.path.exists()` is made and the time `open()` is called. In that case the last line will raise an `IOError`. The same thing would happen if *file* exists but has no read permission. Since testing this on a normal machine on existent and non-existent files makes it seem bugless, the test results will seem fine, and the code will get shipped. Later an unhandled `IOError` (or perhaps some other `EnvironmentError`) escapes to the user, who gets to watch the ugly traceback.

Here is a somewhat better way to do it.

```
def get_status(file):
    try:
        return open(file).readline()
    except EnvironmentError as err:
        print "Unable to open file: {}".format(err)
        sys.exit(1)
```

In this version, *either* the file gets opened and the line is read (so it works even on flaky NFS or SMB connections), or an error message is printed that provides all the available information on why the open failed, and the application is aborted.

However, even this version of `get_status()` makes too many assumptions — that it will only be used in a short running script, and not, say, in a long running server. Sure, the caller could do something like

```
try:
    status = get_status(log)
except SystemExit:
    status = None
```

But there is a better way. You should try to use as few `except` clauses in your code as you can — the ones you do use will usually be inside calls which should always succeed, or a catch-all in a main function.

So, an even better version of `get_status()` is probably

```
def get_status(file):
    return open(file).readline()
```

The caller can deal with the exception if it wants (for example, if it tries several files in a loop), or just let the exception filter upwards to *its* caller.

But the last version still has a serious problem — due to implementation details in CPython, the file would not be closed when an exception is raised until the exception handler finishes; and, worse, in other implementations (e.g., Jython) it might not be closed at all regardless of whether or not an exception is raised.

The best version of this function uses the `open()` call as a context manager, which will ensure that the file gets closed as soon as the function returns:

```
def get_status(file):
    with open(file) as fp:
        return fp.readline()
```

3 Using the Batteries

Every so often, people seem to be writing stuff in the Python library again, usually poorly. While the occasional module has a poor interface, it is usually much better to use the rich standard library and data types that come with Python than inventing your own.

A useful module very few people know about is `os.path`. It always has the correct path arithmetic for your operating system, and will usually be much better than whatever you come up with yourself.

Compare:

```
# ugh!
return dir+"/"+file
# better
return os.path.join(dir, file)
```

More useful functions in `os.path`: `basename()`, `dirname()` and `splitext()`.

There are also many useful built-in functions people seem not to be aware of for some reason: `min()` and `max()` can find the minimum/maximum of any sequence with comparable semantics, for example, yet many people write their own `max()`/`min()`. Another highly useful function is `reduce()` which can be used to repeatly apply a binary operation to a sequence, reducing it to a single value. For example, compute a factorial with a series of multiply operations:

```
>>> n = 4
>>> import operator
>>> reduce(operator.mul, range(1, n+1))
24
```

When it comes to parsing numbers, note that `float()`, `int()` and `long()` all accept string arguments and will reject ill-formed strings by raising an `ValueError`.

4 Using Backslash to Continue Statements

Since Python treats a newline as a statement terminator, and since statements are often more than is comfortable to put in one line, many people do:

```
if foo.bar()['first'][0] == baz.quux(1, 2)[5:9] and \
   calculate_number(10, 20) != forbulate(500, 360):
      pass
```

You should realize that this is dangerous: a stray space after the `\` would make this line wrong, and stray spaces are notoriously hard to see in editors. In this case, at least it would be a syntax error, but if the code was:

```
value = foo.bar()['first'][0]*baz.quux(1, 2)[5:9] \
        + calculate_number(10, 20)*forbulate(500, 360)
```

then it would just be subtly wrong.

It is usually much better to use the implicit continuation inside parenthesis:

This version is bulletproof:

```
value = (foo.bar()['first'][0]*baz.quux(1, 2)[5:9]
        + calculate_number(10, 20)*forbulate(500, 360))
```

Index

Functional Programming HOWTO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author A. M. Kuchling

Release 0.31

In this document, we'll take a tour of Python's features suitable for implementing programs in a functional style. After an introduction to the concepts of functional programming, we'll look at language features such as iterators and generators and relevant library modules such as `itertools` and `functools`.

1 Introduction

This section explains the basic concept of functional programming; if you're just interested in learning about Python language features, skip to the next section.

Programming languages support decomposing problems in several different ways:

- Most programming languages are **procedural**: programs are lists of instructions that tell the computer what to do with the program's input. C, Pascal, and even Unix shells are procedural languages.
- In **declarative** languages, you write a specification that describes the problem to be solved, and the language implementation figures out how to perform the computation efficiently. SQL is the declarative language you're most likely to be familiar with; a SQL query describes the data set you want to retrieve, and the SQL engine decides whether to scan tables or use indexes, which subclauses should be performed first, etc.
- **Object-oriented** programs manipulate collections of objects. Objects have internal state and support methods that query or modify this internal state in some way. Smalltalk and Java are object-oriented languages. C++ and Python are languages that support object-oriented programming, but don't force the use of object-oriented features.
- **Functional** programming decomposes a problem into a set of functions. Ideally, functions only take inputs and produce outputs, and don't have any internal state that affects the output produced for a given input. Well-known functional languages include the ML family (Standard ML, OCaml, and other variants) and Haskell.

The designers of some computer languages choose to emphasize one particular approach to programming. This often makes it difficult to write programs that use a different approach. Other languages are multi-paradigm languages that support several different approaches. Lisp, C++, and Python are multi-paradigm; you can write programs or libraries that are largely procedural, object-oriented, or functional in all of these languages. In a large program, different sections might be written using different approaches; the GUI might be object-oriented while the processing logic is procedural or functional, for example.

In a functional program, input flows through a set of functions. Each function operates on its input and produces some output. Functional style discourages functions with side effects that modify internal state or make other changes that aren't visible in the function's return value. Functions that have no side effects at all are called **purely functional**. Avoiding side effects means not using data structures that get updated as a program runs; every function's output must only depend on its input.

Some languages are very strict about purity and don't even have assignment statements such as `a=3` or `c = a + b`, but it's difficult to avoid all side effects. Printing to the screen or writing to a disk file are side effects, for example. For example, in Python a `print` statement or a `time.sleep(1)` both return no useful value; they're only called for their side effects of sending some text to the screen or pausing execution for a second.

Python programs written in functional style usually won't go to the extreme of avoiding all I/O or all assignments; instead, they'll provide a functional-appearing interface but will use non-functional features internally. For example, the implementation of a function will still use assignments to local variables, but won't modify global variables or have other side effects.

Functional programming can be considered the opposite of object-oriented programming. Objects are little capsules containing some internal state along with a collection of method calls that let you modify this state, and programs consist of making the right set of state changes. Functional programming wants to avoid state changes as much as possible and works with data flowing between functions. In Python you might combine the two approaches by writing functions that take and return instances representing objects in your application (e-mail messages, transactions, etc.).

Functional design may seem like an odd constraint to work under. Why should you avoid objects and side effects? There are theoretical and practical advantages to the functional style:

- Formal provability.
- Modularity.
- Composability.
- Ease of debugging and testing.

1.1 Formal provability

A theoretical benefit is that it's easier to construct a mathematical proof that a functional program is correct.

For a long time researchers have been interested in finding ways to mathematically prove programs correct. This is different from testing a program on numerous inputs and concluding that its output is usually correct, or reading a program's source code and concluding that the code looks right; the goal is instead a rigorous proof that a program produces the right result for all possible inputs.

The technique used to prove programs correct is to write down **invariants**, properties of the input data and of the program's variables that are always true. For each line of code, you then show that if invariants X and Y are true **before** the line is executed, the slightly different invariants X' and Y' are true **after** the line is executed. This continues until you reach the end of the program, at which point the invariants should match the desired conditions on the program's output.

Functional programming's avoidance of assignments arose because assignments are difficult to handle with this technique; assignments can break invariants that were true before the assignment without producing any new invariants that can be propagated onward.

Unfortunately, proving programs correct is largely impractical and not relevant to Python software. Even trivial programs require proofs that are several pages long; the proof of correctness for a moderately complicated program would be enormous, and few or none of the programs you use daily (the Python interpreter, your XML parser, your web browser) could be proven correct. Even if you wrote down or generated a proof, there would then be the question of verifying the proof; maybe there's an error in it, and you wrongly believe you've proved the program correct.

1.2 Modularity

A more practical benefit of functional programming is that it forces you to break apart your problem into small pieces. Programs are more modular as a result. It's easier to specify and write a small function that does one thing than a large function that performs a complicated transformation. Small functions are also easier to read and to check for errors.

1.3 Ease of debugging and testing

Testing and debugging a functional-style program is easier.

Debugging is simplified because functions are generally small and clearly specified. When a program doesn't work, each function is an interface point where you can check that the data are correct. You can look at the intermediate inputs and outputs to quickly isolate the function that's responsible for a bug.

Testing is easier because each function is a potential subject for a unit test. Functions don't depend on system state that needs to be replicated before running a test; instead you only have to synthesize the right input and then check that the output matches expectations.

1.4 Composability

As you work on a functional-style program, you'll write a number of functions with varying inputs and outputs. Some of these functions will be unavoidably specialized to a particular application, but others will be useful in a wide variety of programs. For example, a function that takes a directory path and returns all the XML files in the directory, or a function that takes a filename and returns its contents, can be applied to many different situations.

Over time you'll form a personal library of utilities. Often you'll assemble new programs by arranging existing functions in a new configuration and writing a few functions specialized for the current task.

2 Iterators

I'll start by looking at a Python language feature that's an important foundation for writing functional-style programs: iterators.

An iterator is an object representing a stream of data; this object returns the data one element at a time. A Python iterator must support a method called `next()` that takes no arguments and always returns the next element of the stream. If there are no more elements in the stream, `next()` must raise the `StopIteration` exception. Iterators don't have to be finite, though; it's perfectly reasonable to write an iterator that produces an infinite stream of data.

The built-in `iter()` function takes an arbitrary object and tries to return an iterator that will return the object's contents or elements, raising `TypeError` if the object doesn't support iteration. Several of Python's built-in data types support iteration, the most common being lists and dictionaries. An object is called an **iterable** object if you can get an iterator for it.

You can experiment with the iteration interface manually:

```
>>> L = [1,2,3]
>>> it = iter(L)
>>> print it
<...iterator object at ...>
>>> it.next()
1
>>> it.next()
2
>>> it.next()
3
>>> it.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Python expects iterable objects in several different contexts, the most important being the `for` statement. In the statement `for X in Y`, Y must be an iterator or some object for which `iter()` can create an iterator. These two statements are equivalent:

```
for i in iter(obj):
    print i

for i in obj:
    print i
```

Iterators can be materialized as lists or tuples by using the `list()` or `tuple()` constructor functions:

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> t = tuple(iterator)
>>> t
(1, 2, 3)
```

Sequence unpacking also supports iterators: if you know an iterator will return N elements, you can unpack them into an N-tuple:

```
>>> L = [1,2,3]
>>> iterator = iter(L)
>>> a,b,c = iterator
>>> a,b,c
(1, 2, 3)
```

Built-in functions such as `max()` and `min()` can take a single iterator argument and will return the largest or smallest element. The `"in"` and `"not in"` operators also support iterators: `X in iterator` is true if X is found in the stream returned by the iterator. You'll run into obvious problems if the iterator is infinite; `max()`, `min()` will never return, and if the element X never appears in the stream, the `"in"` and `"not in"` operators won't return either.

Note that you can only go forward in an iterator; there's no way to get the previous element, reset the iterator, or make a copy of it. Iterator objects can optionally provide these additional capabilities, but the iterator protocol only specifies the `next()` method. Functions may therefore consume all of the iterator's output, and if you need to do something different with the same stream, you'll have to create a new iterator.

2.1 Data Types That Support Iterators

We've already seen how lists and tuples support iterators. In fact, any Python sequence type, such as strings, will automatically support creation of an iterator.

Calling `iter()` on a dictionary returns an iterator that will loop over the dictionary's keys:

```
>>> m = {'Jan': 1, 'Feb': 2, 'Mar': 3, 'Apr': 4, 'May': 5, 'Jun': 6,
...      'Jul': 7, 'Aug': 8, 'Sep': 9, 'Oct': 10, 'Nov': 11, 'Dec': 12}
>>> for key in m:
...     print key, m[key]
Mar 3
Feb 2
Aug 8
Sep 9
Apr 4
Jun 6
Jul 7
Jan 1
May 5
```

```
Nov 11
Dec 12
Oct 10
```

Note that the order is essentially random, because it's based on the hash ordering of the objects in the dictionary.

Applying `iter()` to a dictionary always loops over the keys, but dictionaries have methods that return other iterators. If you want to iterate over keys, values, or key/value pairs, you can explicitly call the `iterkeys()`, `itervalues()`, or `iteritems()` methods to get an appropriate iterator.

The `dict()` constructor can accept an iterator that returns a finite stream of `(key, value)` tuples:

```
>>> L = [('Italy', 'Rome'), ('France', 'Paris'), ('US', 'Washington DC')]
>>> dict(iter(L))
{'Italy': 'Rome', 'US': 'Washington DC', 'France': 'Paris'}
```

Files also support iteration by calling the `readline()` method until there are no more lines in the file. This means you can read each line of a file like this:

```
for line in file:
    # do something for each line
    ...
```

Sets can take their contents from an iterable and let you iterate over the set's elements:

```
S = set((2, 3, 5, 7, 11, 13))
for i in S:
    print i
```

3 Generator expressions and list comprehensions

Two common operations on an iterator's output are 1) performing some operation for every element, 2) selecting a subset of elements that meet some condition. For example, given a list of strings, you might want to strip off trailing whitespace from each line or extract all the strings containing a given substring.

List comprehensions and generator expressions (short form: "listcomps" and "genexps") are a concise notation for such operations, borrowed from the functional programming language Haskell (https://www.haskell.org/). You can strip all the whitespace from a stream of strings with the following code:

```
line_list = ['  line 1\n', 'line 2  \n', ...]

# Generator expression -- returns iterator
stripped_iter = (line.strip() for line in line_list)

# List comprehension -- returns list
stripped_list = [line.strip() for line in line_list]
```

You can select only certain elements by adding an `"if"` condition:

```
stripped_list = [line.strip() for line in line_list
                 if line != ""]
```

With a list comprehension, you get back a Python list; `stripped_list` is a list containing the resulting lines, not an iterator. Generator expressions return an iterator that computes the values as necessary, not needing to materialize all the values at once. This means that list comprehensions aren't useful if you're

working with iterators that return an infinite stream or a very large amount of data. Generator expressions are preferable in these situations.

Generator expressions are surrounded by parentheses ("()") and list comprehensions are surrounded by square brackets ("[]"). Generator expressions have the form:

```
( expression for expr in sequence1
             if condition1
             for expr2 in sequence2
             if condition2
             for expr3 in sequence3 ...
             if condition3
             for exprN in sequenceN
             if conditionN )
```

Again, for a list comprehension only the outside brackets are different (square brackets instead of parentheses).

The elements of the generated output will be the successive values of `expression`. The `if` clauses are all optional; if present, `expression` is only evaluated and added to the result when `condition` is true.

Generator expressions always have to be written inside parentheses, but the parentheses signalling a function call also count. If you want to create an iterator that will be immediately passed to a function you can write:

```
obj_total = sum(obj.count for obj in list_all_objects())
```

The `for...in` clauses contain the sequences to be iterated over. The sequences do not have to be the same length, because they are iterated over from left to right, **not** in parallel. For each element in `sequence1`, `sequence2` is looped over from the beginning. `sequence3` is then looped over for each resulting pair of elements from `sequence1` and `sequence2`.

To put it another way, a list comprehension or generator expression is equivalent to the following Python code:

```
for expr1 in sequence1:
    if not (condition1):
        continue   # Skip this element
    for expr2 in sequence2:
        if not (condition2):
            continue   # Skip this element
        ...
        for exprN in sequenceN:
            if not (conditionN):
                continue   # Skip this element

            # Output the value of
            # the expression.
```

This means that when there are multiple `for...in` clauses but no `if` clauses, the length of the resulting output will be equal to the product of the lengths of all the sequences. If you have two lists of length 3, the output list is 9 elements long:

```
>>> seq1 = 'abc'
>>> seq2 = (1,2,3)
>>> [(x,y) for x in seq1 for y in seq2]
[('a', 1), ('a', 2), ('a', 3),
 ('b', 1), ('b', 2), ('b', 3),
 ('c', 1), ('c', 2), ('c', 3)]
```

To avoid introducing an ambiguity into Python's grammar, if `expression` is creating a tuple, it must be surrounded with parentheses. The first list comprehension below is a syntax error, while the second one is correct:

```
# Syntax error
[ x,y for x in seq1 for y in seq2]
# Correct
[ (x,y) for x in seq1 for y in seq2]
```

4 Generators

Generators are a special class of functions that simplify the task of writing iterators. Regular functions compute a value and return it, but generators return an iterator that returns a stream of values.

You're doubtless familiar with how regular function calls work in Python or C. When you call a function, it gets a private namespace where its local variables are created. When the function reaches a `return` statement, the local variables are destroyed and the value is returned to the caller. A later call to the same function creates a new private namespace and a fresh set of local variables. But, what if the local variables weren't thrown away on exiting a function? What if you could later resume the function where it left off? This is what generators provide; they can be thought of as resumable functions.

Here's the simplest example of a generator function:

```
def generate_ints(N):
    for i in range(N):
        yield i
```

Any function containing a `yield` keyword is a generator function; this is detected by Python's bytecode compiler which compiles the function specially as a result.

When you call a generator function, it doesn't return a single value; instead it returns a generator object that supports the iterator protocol. On executing the `yield` expression, the generator outputs the value of `i`, similar to a `return` statement. The big difference between `yield` and a `return` statement is that on reaching a `yield` the generator's state of execution is suspended and local variables are preserved. On the next call to the generator's `.next()` method, the function will resume executing.

Here's a sample usage of the `generate_ints()` generator:

```
>>> gen = generate_ints(3)
>>> gen
<generator object generate_ints at ...>
>>> gen.next()
0
>>> gen.next()
1
>>> gen.next()
2
>>> gen.next()
Traceback (most recent call last):
  File "stdin", line 1, in <module>
  File "stdin", line 2, in generate_ints
StopIteration
```

You could equally write `for i in generate_ints(5)`, or `a,b,c = generate_ints(3)`.

Inside a generator function, the `return` statement can only be used without a value, and signals the end of the procession of values; after executing a `return` the generator cannot return any further values. `return`

with a value, such as `return 5`, is a syntax error inside a generator function. The end of the generator's results can also be indicated by raising `StopIteration` manually, or by just letting the flow of execution fall off the bottom of the function.

You could achieve the effect of generators manually by writing your own class and storing all the local variables of the generator as instance variables. For example, returning a list of integers could be done by setting `self.count` to 0, and having the `next()` method increment `self.count` and return it. However, for a moderately complicated generator, writing a corresponding class can be much messier.

The test suite included with Python's library, `test_generators.py`, contains a number of more interesting examples. Here's one generator that implements an in-order traversal of a tree using generators recursively.

```
# A recursive generator that generates Tree leaves in in-order.
def inorder(t):
    if t:
        for x in inorder(t.left):
            yield x

        yield t.label

        for x in inorder(t.right):
            yield x
```

Two other examples in `test_generators.py` produce solutions for the N-Queens problem (placing N queens on an NxN chess board so that no queen threatens another) and the Knight's Tour (finding a route that takes a knight to every square of an NxN chessboard without visiting any square twice).

4.1 Passing values into a generator

In Python 2.4 and earlier, generators only produced output. Once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. You could hack together this ability by making the generator look at a global variable or by passing in some mutable object that callers then modify, but these approaches are messy.

In Python 2.5 there's a simple way to pass values into a generator. `yield` became an expression, returning a value that can be assigned to a variable or otherwise operated on:

```
val = (yield i)
```

I recommend that you **always** put parentheses around a `yield` expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(value)` method. This method resumes the generator's code and the `yield` expression returns the specified value. If the regular `next()` method is called, the `yield` returns `None`.

Here's a simple counter that increments by 1 and allows changing the value of the internal counter.

```
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
```

```
            i = val
        else:
            i += 1
```

And here's an example of changing the counter:

```
>>> it = counter(10)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File "t.py", line 15, in <module>
    print it.next()
StopIteration
```

Because `yield` will often be returning `None`, you should always check for this case. Don't just use its value in expressions unless you're sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other new methods on generators:

- `throw(type, value=None, traceback=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.
- `close()` raises a `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`; catching the exception and doing anything else is illegal and will trigger a `RuntimeError`. `close()` will also be called by Python's garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become **coroutines**, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements).

5 Built-in functions

Let's look in more detail at built-in functions often used with iterators.

Two of Python's built-in functions, `map()` and `filter()`, are somewhat obsolete; they duplicate the features of list comprehensions but return actual lists instead of iterators.

`map(f, iterA, iterB, ...)` returns a list containing `f(iterA[0], iterB[0]), f(iterA[1], iterB[1]), f(iterA[2], iterB[2]), ....`

```
>>> def upper(s):
...     return s.upper()
```

```
>>> map(upper, ['sentence', 'fragment'])
['SENTENCE', 'FRAGMENT']
```

```
>>> [upper(s) for s in ['sentence', 'fragment']]
['SENTENCE', 'FRAGMENT']
```

As shown above, you can achieve the same effect with a list comprehension. The `itertools.imap()` function does the same thing but can handle infinite iterators; it'll be discussed later, in the section on the `itertools` module.

`filter(predicate, iter)` returns a list that contains all the sequence elements that meet a certain condition, and is similarly duplicated by list comprehensions. A **predicate** is a function that returns the truth value of some condition; for use with `filter()`, the predicate must take a single value.

```
>>> def is_even(x):
...     return (x % 2) == 0
```

```
>>> filter(is_even, range(10))
[0, 2, 4, 6, 8]
```

This can also be written as a list comprehension:

```
>>> [x for x in range(10) if is_even(x)]
[0, 2, 4, 6, 8]
```

`filter()` also has a counterpart in the `itertools` module, `itertools.ifilter()`, that returns an iterator and can therefore handle infinite sequences just as `itertools.imap()` can.

`reduce(func, iter, [initial_value])` doesn't have a counterpart in the `itertools` module because it cumulatively performs an operation on all the iterable's elements and therefore can't be applied to infinite iterables. `func` must be a function that takes two elements and returns a single value. `reduce()` takes the first two elements A and B returned by the iterator and calculates `func(A, B)`. It then requests the third element, C, calculates `func(func(A, B), C)`, combines this result with the fourth element returned, and continues until the iterable is exhausted. If the iterable returns no values at all, a `TypeError` exception is raised. If the initial value is supplied, it's used as a starting point and `func(initial_value, A)` is the first calculation.

```
>>> import operator
>>> reduce(operator.concat, ['A', 'BB', 'C'])
'ABBC'
>>> reduce(operator.concat, [])
Traceback (most recent call last):
  ...
TypeError: reduce() of empty sequence with no initial value
>>> reduce(operator.mul, [1,2,3], 1)
6
>>> reduce(operator.mul, [], 1)
1
```

If you use `operator.add()` with `reduce()`, you'll add up all the elements of the iterable. This case is so common that there's a special built-in called `sum()` to compute it:

```
>>> reduce(operator.add, [1,2,3,4], 0)
10
>>> sum([1,2,3,4])
10
```

```
>>> sum([])
0
```

For many uses of `reduce()`, though, it can be clearer to just write the obvious `for` loop:

```
# Instead of:
product = reduce(operator.mul, [1,2,3], 1)

# You can write:
product = 1
for i in [1,2,3]:
    product *= i
```

`enumerate(iter)` counts off the elements in the iterable, returning 2-tuples containing the count and each element.

```
>>> for item in enumerate(['subject', 'verb', 'object']):
...     print item
(0, 'subject')
(1, 'verb')
(2, 'object')
```

`enumerate()` is often used when looping through a list and recording the indexes at which certain conditions are met:

```
f = open('data.txt', 'r')
for i, line in enumerate(f):
    if line.strip() == '':
        print 'Blank line at line #%i' % i
```

`sorted(iterable, [cmp=None], [key=None], [reverse=False])` collects all the elements of the iterable into a list, sorts the list, and returns the sorted result. The `cmp`, `key`, and `reverse` arguments are passed through to the constructed list's `.sort()` method.

```
>>> import random
>>> # Generate 8 random numbers between [0, 10000)
>>> rand_list = random.sample(range(10000), 8)
>>> rand_list
[769, 7953, 9828, 6431, 8442, 9878, 6213, 2207]
>>> sorted(rand_list)
[769, 2207, 6213, 6431, 7953, 8442, 9828, 9878]
>>> sorted(rand_list, reverse=True)
[9878, 9828, 8442, 7953, 6431, 6213, 2207, 769]
```

(For a more detailed discussion of sorting, see the Sorting mini-HOWTO in the Python wiki at https://wiki.python.org/moin/HowTo/Sorting.)

The `any(iter)` and `all(iter)` built-ins look at the truth values of an iterable's contents. `any()` returns `True` if any element in the iterable is a true value, and `all()` returns `True` if all of the elements are true values:

```
>>> any([0,1,0])
True
>>> any([0,0,0])
False
>>> any([1,1,1])
True
>>> all([0,1,0])
```

```
False
>>> all([0,0,0])
False
>>> all([1,1,1])
True
```

6 Small functions and the lambda expression

When writing functional-style programs, you'll often need little functions that act as predicates or that combine elements in some way.

If there's a Python built-in or a module function that's suitable, you don't need to define a new function at all:

```
stripped_lines = [line.strip() for line in lines]
existing_files = filter(os.path.exists, file_list)
```

If the function you need doesn't exist, you need to write it. One way to write small functions is to use the `lambda` statement. `lambda` takes a number of parameters and an expression combining these parameters, and creates a small function that returns the value of the expression:

```
lowercase = lambda x: x.lower()

print_assign = lambda name, value: name + '=' + str(value)

adder = lambda x, y: x+y
```

An alternative is to just use the `def` statement and define a function in the usual way:

```
def lowercase(x):
    return x.lower()

def print_assign(name, value):
    return name + '=' + str(value)

def adder(x,y):
    return x + y
```

Which alternative is preferable? That's a style question; my usual course is to avoid using `lambda`.

One reason for my preference is that `lambda` is quite limited in the functions it can define. The result has to be computable as a single expression, which means you can't have multiway `if... elif... else` comparisons or `try... except` statements. If you try to do too much in a `lambda` statement, you'll end up with an overly complicated expression that's hard to read. Quick, what's the following code doing?

```
total = reduce(lambda a, b: (0, a[1] + b[1]), items)[1]
```

You can figure it out, but it takes time to disentangle the expression to figure out what's going on. Using a short nested `def` statements makes things a little bit better:

```
def combine (a, b):
    return 0, a[1] + b[1]

total = reduce(combine, items)[1]
```

But it would be best of all if I had simply used a `for` loop:

```
total = 0
for a, b in items:
    total += b
```

Or the `sum()` built-in and a generator expression:

```
total = sum(b for a,b in items)
```

Many uses of `reduce()` are clearer when written as `for` loops.

Fredrik Lundh once suggested the following set of rules for refactoring uses of `lambda`:

1. Write a lambda function.
2. Write a comment explaining what the heck that lambda does.
3. Study the comment for a while, and think of a name that captures the essence of the comment.
4. Convert the lambda to a def statement, using that name.
5. Remove the comment.

I really like these rules, but you're free to disagree about whether this lambda-free style is better.

7 The itertools module

The `itertools` module contains a number of commonly-used iterators as well as functions for combining several iterators. This section will introduce the module's contents by showing small examples.

The module's functions fall into a few broad classes:

- Functions that create a new iterator based on an existing iterator.
- Functions for treating an iterator's elements as function arguments.
- Functions for selecting portions of an iterator's output.
- A function for grouping an iterator's output.

7.1 Creating new iterators

`itertools.count(n)` returns an infinite stream of integers, increasing by 1 each time. You can optionally supply the starting number, which defaults to 0:

```
itertools.count() =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
itertools.count(10) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
```

`itertools.cycle(iter)` saves a copy of the contents of a provided iterable and returns a new iterator that returns its elements from first to last. The new iterator will repeat these elements infinitely.

```
itertools.cycle([1,2,3,4,5]) =>
  1, 2, 3, 4, 5, 1, 2, 3, 4, 5, ...
```

`itertools.repeat(elem, [n])` returns the provided element `n` times, or returns the element endlessly if `n` is not provided.

```
itertools.repeat('abc') =>
  abc, abc, abc, abc, abc, abc, abc, abc, abc, abc, ...
itertools.repeat('abc', 5) =>
  abc, abc, abc, abc, abc
```

`itertools.chain(iterA, iterB, ...)` takes an arbitrary number of iterables as input, and returns all the elements of the first iterator, then all the elements of the second, and so on, until all of the iterables have been exhausted.

```
itertools.chain(['a', 'b', 'c'], (1, 2, 3)) =>
  a, b, c, 1, 2, 3
```

`itertools.izip(iterA, iterB, ...)` takes one element from each iterable and returns them in a tuple:

```
itertools.izip(['a', 'b', 'c'], (1, 2, 3)) =>
  ('a', 1), ('b', 2), ('c', 3)
```

It's similar to the built-in `zip()` function, but doesn't construct an in-memory list and exhaust all the input iterators before returning; instead tuples are constructed and returned only if they're requested. (The technical term for this behaviour is lazy evaluation.)

This iterator is intended to be used with iterables that are all of the same length. If the iterables are of different lengths, the resulting stream will be the same length as the shortest iterable.

```
itertools.izip(['a', 'b'], (1, 2, 3)) =>
  ('a', 1), ('b', 2)
```

You should avoid doing this, though, because an element may be taken from the longer iterators and discarded. This means you can't go on to use the iterators further because you risk skipping a discarded element.

`itertools.islice(iter, [start], stop, [step])` returns a stream that's a slice of the iterator. With a single `stop` argument, it will return the first `stop` elements. If you supply a starting index, you'll get `stop-start` elements, and if you supply a value for `step`, elements will be skipped accordingly. Unlike Python's string and list slicing, you can't use negative values for `start`, `stop`, or `step`.

```
itertools.islice(range(10), 8) =>
  0, 1, 2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8) =>
  2, 3, 4, 5, 6, 7
itertools.islice(range(10), 2, 8, 2) =>
  2, 4, 6
```

`itertools.tee(iter, [n])` replicates an iterator; it returns `n` independent iterators that will all return the contents of the source iterator. If you don't supply a value for `n`, the default is 2. Replicating iterators requires saving some of the contents of the source iterator, so this can consume significant memory if the iterator is large and one of the new iterators is consumed more than the others.

```
itertools.tee( itertools.count() ) =>
   iterA, iterB

where iterA ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...

and   iterB ->
   0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
```

7.2 Calling functions on elements

Two functions are used for calling other functions on the contents of an iterable.

`itertools.imap(f, iterA, iterB, ...)` returns a stream containing `f(iterA[0], iterB[0])`, `f(iterA[1], iterB[1])`, `f(iterA[2], iterB[2])`, ...:

```
itertools.imap(operator.add, [5, 6, 5], [1, 2, 3]) =>
  6, 8, 8
```

The `operator` module contains a set of functions corresponding to Python's operators. Some examples are `operator.add(a, b)` (adds two values), `operator.ne(a, b)` (same as `a!=b`), and `operator.attrgetter('id')` (returns a callable that fetches the `"id"` attribute).

`itertools.starmap(func, iter)` assumes that the iterable will return a stream of tuples, and calls `f()` using these tuples as the arguments:

```
itertools.starmap(os.path.join,
                  [('/usr', 'bin', 'java'), ('/bin', 'python'),
                   ('/usr', 'bin', 'perl'),('/usr', 'bin', 'ruby')])
=>
  /usr/bin/java, /bin/python, /usr/bin/perl, /usr/bin/ruby
```

7.3 Selecting elements

Another group of functions chooses a subset of an iterator's elements based on a predicate.

`itertools.ifilter(predicate, iter)` returns all the elements for which the predicate returns true:

```
def is_even(x):
    return (x % 2) == 0

itertools.ifilter(is_even, itertools.count()) =>
  0, 2, 4, 6, 8, 10, 12, 14, ...
```

`itertools.ifilterfalse(predicate, iter)` is the opposite, returning all elements for which the predicate returns false:

```
itertools.ifilterfalse(is_even, itertools.count()) =>
  1, 3, 5, 7, 9, 11, 13, 15, ...
```

`itertools.takewhile(predicate, iter)` returns elements for as long as the predicate returns true. Once the predicate returns false, the iterator will signal the end of its results.

```
def less_than_10(x):
    return (x < 10)

itertools.takewhile(less_than_10, itertools.count()) =>
  0, 1, 2, 3, 4, 5, 6, 7, 8, 9

itertools.takewhile(is_even, itertools.count()) =>
  0
```

`itertools.dropwhile(predicate, iter)` discards elements while the predicate returns true, and then returns the rest of the iterable's results.

```
itertools.dropwhile(less_than_10, itertools.count()) =>
  10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...

itertools.dropwhile(is_even, itertools.count()) =>
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
```

7.4 Grouping elements

The last function I'll discuss, `itertools.groupby(iter, key_func=None)`, is the most complicated. `key_func(elem)` is a function that can compute a key value for each element returned by the iterable. If you don't supply a key function, the key is simply each element itself.

`groupby()` collects all the consecutive elements from the underlying iterable that have the same key value, and returns a stream of 2-tuples containing a key value and an iterator for the elements with that key.

```
city_list = [('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL'),
             ('Anchorage', 'AK'), ('Nome', 'AK'),
             ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ'),
             ...
            ]

def get_state ((city, state)):
    return state

itertools.groupby(city_list, get_state) =>
  ('AL', iterator-1),
  ('AK', iterator-2),
  ('AZ', iterator-3), ...

where
iterator-1 =>
  ('Decatur', 'AL'), ('Huntsville', 'AL'), ('Selma', 'AL')
iterator-2 =>
  ('Anchorage', 'AK'), ('Nome', 'AK')
iterator-3 =>
  ('Flagstaff', 'AZ'), ('Phoenix', 'AZ'), ('Tucson', 'AZ')
```

`groupby()` assumes that the underlying iterable's contents will already be sorted based on the key. Note that the returned iterators also use the underlying iterable, so you have to consume the results of iterator-1 before requesting iterator-2 and its corresponding key.

8 The functools module

The `functools` module in Python 2.5 contains some higher-order functions. A **higher-order function** takes one or more functions as input and returns a new function. The most useful tool in this module is the `functools.partial()` function.

For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c)`; you may wish to create a new function `g(b, c)` that's equivalent to `f(1, b, c)`; you're filling in a value for one of `f()`'s parameters. This is called "partial function application".

The constructor for `partial` takes the arguments `(function, arg1, arg2, ... kwarg1=value1, kwarg2=value2)`. The resulting object is callable, so you can just call it to invoke `function` with the filled-in arguments.

Here's a small but realistic example:

```
import functools

def log (message, subsystem):
    "Write the contents of 'message' to the specified subsystem."
    print '%s: %s' % (subsystem, message)
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

8.1 The operator module

The `operator` module was mentioned earlier. It contains a set of functions corresponding to Python's operators. These functions are often useful in functional-style code because they save you from writing trivial functions that perform a single operation.

Some of the functions in this module are:

- Math operations: `add()`, `sub()`, `mul()`, `div()`, `floordiv()`, `abs()`, ...
- Logical operations: `not_()`, `truth()`.
- Bitwise operations: `and_()`, `or_()`, `invert()`.
- Comparisons: `eq()`, `ne()`, `lt()`, `le()`, `gt()`, and `ge()`.
- Object identity: `is_()`, `is_not()`.

Consult the operator module's documentation for a complete list.

9 Revision History and Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Ian Bicking, Nick Coghlan, Nick Efford, Raymond Hettinger, Jim Jewett, Mike Krell, Leandro Lameiro, Jussi Salmela, Collin Winter, Blake Winton.

Version 0.1: posted June 30 2006.

Version 0.11: posted July 1 2006. Typo fixes.

Version 0.2: posted July 10 2006. Merged genexp and listcomp sections into one. Typo fixes.

Version 0.21: Added more references suggested on the tutor mailing list.

Version 0.30: Adds a section on the `functional` module written by Collin Winter; adds short section on the operator module; a few other edits.

10 References

10.1 General

Structure and Interpretation of Computer Programs, by Harold Abelson and Gerald Jay Sussman with Julie Sussman. Full text at https://mitpress.mit.edu/sicp/. In this classic textbook of computer science, chapters 2 and 3 discuss the use of sequences and streams to organize the data flow inside a program. The

book uses Scheme for its examples, but many of the design approaches described in these chapters are applicable to functional-style Python code.

http://www.defmacro.org/ramblings/fp.html: A general introduction to functional programming that uses Java examples and has a lengthy historical introduction.

https://en.wikipedia.org/wiki/Functional_programming: General Wikipedia entry describing functional programming.

https://en.wikipedia.org/wiki/Coroutine: Entry for coroutines.

https://en.wikipedia.org/wiki/Currying: Entry for the concept of currying.

10.2 Python-specific

http://gnosis.cx/TPiP/: The first chapter of David Mertz's book *Text Processing in Python* discusses functional programming for text processing, in the section titled "Utilizing Higher-Order Functions in Text Processing".

Mertz also wrote a 3-part series of articles on functional programming for IBM's DeveloperWorks site; see

part 1, part 2, and part 3,

10.3 Python documentation

Documentation for the `itertools` module.

Documentation for the `operator` module.

PEP 289: "Generator Expressions"

PEP 342: "Coroutines via Enhanced Generators" describes the new generator features in Python 2.5.

Index

P

Logging HOWTO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Vinay Sajip <vinay_sajip at red-dove dot com>

1 Basic Logging Tutorial

Logging is a means of tracking events that happen when some software runs. The software's developer adds logging calls to their code to indicate that certain events have occurred. An event is described by a descriptive message which can optionally contain variable data (i.e. data that is potentially different for each occurrence of the event). Events also have an importance which the developer ascribes to the event; the importance can also be called the *level* or *severity*.

1.1 When to use logging

Logging provides a set of convenience functions for simple logging usage. These are `debug()`, `info()`, `warning()`, `error()` and `critical()`. To determine when to use logging, see the table below, which states, for each of a set of common tasks, the best tool to use for it.

Task you want to perform	The best tool for the task
Display console output for ordinary usage of a command line script or program	`print()`
Report events that occur during normal operation of a program (e.g. for status monitoring or fault investigation)	`logging.info()` (or `logging.debug()` for very detailed output for diagnostic purposes)
Issue a warning regarding a particular runtime event	`warnings.warn()` in library code if the issue is avoidable and the client application should be modified to eliminate the warning `logging.warning()` if there is nothing the client application can do about the situation, but the event should still be noted
Report an error regarding a particular runtime event	Raise an exception
Report suppression of an error without raising an exception (e.g. error handler in a long-running server process)	`logging.error()`, `logging.exception()` or `logging.critical()` as appropriate for the specific error and application domain

The logging functions are named after the level or severity of the events they are used to track. The standard levels and their applicability are described below (in increasing order of severity):

Level	When it's used
`DEBUG`	Detailed information, typically of interest only when diagnosing problems.
`INFO`	Confirmation that things are working as expected.
`WARNING`	An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
`ERROR`	Due to a more serious problem, the software has not been able to perform some function.
`CRITICAL`	A serious error, indicating that the program itself may be unable to continue running.

The default level is `WARNING`, which means that only events of this level and above will be tracked, unless the logging package is configured to do otherwise.

Events that are tracked can be handled in different ways. The simplest way of handling tracked events is to print them to the console. Another common way is to write them to a disk file.

1.2 A simple example

A very simple example is:

```
import logging
logging.warning('Watch out!')  # will print a message to the console
logging.info('I told you so')  # will not print anything
```

If you type these lines into a script and run it, you'll see:

```
WARNING:root:Watch out!
```

printed out on the console. The `INFO` message doesn't appear because the default level is `WARNING`. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. 'Watch out!'. Don't worry about the 'root' part for now: it will be explained later. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

1.3 Logging to a file

A very common situation is that of recording logging events in a file, so let's look at that next. Be sure to try the following in a newly-started Python interpreter, and don't just continue from the session described above:

```
import logging
logging.basicConfig(filename='example.log',level=logging.DEBUG)
logging.debug('This message should go to the log file')
logging.info('So should this')
logging.warning('And this, too')
```

And now if we open the file and look at what we have, we should find the log messages:

```
DEBUG:root:This message should go to the log file
INFO:root:So should this
WARNING:root:And this, too
```

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to `DEBUG`, all of the messages were printed.

If you want to set the logging level from a command-line option such as:

```
--log=INFO
```

and you have the value of the parameter passed for `--log` in some variable *loglevel*, you can use:

```
getattr(logging, loglevel.upper())
```

to get the value which you'll pass to `basicConfig()` via the *level* argument. You may want to error check any user input value, perhaps as in the following example:

```
# assuming loglevel is bound to the string value obtained from the
# command line argument. Convert to upper case to allow the user to
# specify --log=DEBUG or --log=debug
numeric_level = getattr(logging, loglevel.upper(), None)
if not isinstance(numeric_level, int):
    raise ValueError('Invalid log level: %s' % loglevel)
logging.basicConfig(level=numeric_level, ...)
```

The call to `basicConfig()` should come *before* any calls to `debug()`, `info()` etc. As it's intended as a one-off simple configuration facility, only the first call will actually do anything: subsequent calls are effectively no-ops.

If you run the above script several times, the messages from successive runs are appended to the file *example.log*. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* argument, by changing the call in the above example to:

```
logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

1.4 Logging from multiple modules

If your program consists of multiple modules, here's an example of how you could organize logging in it:

```
# myapp.py
import logging
import mylib

def main():
    logging.basicConfig(filename='myapp.log', level=logging.INFO)
    logging.info('Started')
    mylib.do_something()
    logging.info('Finished')

if __name__ == '__main__':
    main()
```

```
# mylib.py
import logging

def do_something():
    logging.info('Doing something')
```

If you run *myapp.py*, you should see this in *myapp.log*:

```
INFO:root:Started
INFO:root:Doing something
INFO:root:Finished
```

which is hopefully what you were expecting to see. You can generalize this to multiple modules, using the pattern in *mylib.py*. Note that for this simple usage pattern, you won't know, by looking in the log file, *where* in your application your messages came from, apart from looking at the event description. If you want to track the location of your messages, you'll need to refer to the documentation beyond the tutorial level – see *Advanced Logging Tutorial*.

1.5 Logging variable data

To log variable data, use a format string for the event description message and append the variable data as arguments. For example:

```
import logging
logging.warning('%s before you %s', 'Look', 'leap!')
```

will display:

```
WARNING:root:Look before you leap!
```

As you can see, merging of variable data into the event description message uses the old, %-style of string formatting. This is for backwards compatibility: the logging package pre-dates newer formatting options such as `str.format()` and `string.Template`. These newer formatting options *are* supported, but exploring them is outside the scope of this tutorial.

1.6 Changing the format of displayed messages

To change the format which is used to display messages, you need to specify the format you want to use:

```
import logging
logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.DEBUG)
logging.debug('This message should appear on the console')
logging.info('So should this')
logging.warning('And this, too')
```

which would print:

```
DEBUG:This message should appear on the console
INFO:So should this
WARNING:And this, too
```

Notice that the 'root' which appeared in earlier examples has disappeared. For a full set of things that can appear in format strings, you can refer to the documentation for logrecord-attributes, but for simple usage, you just need the *levelname* (severity), *message* (event description, including variable data) and perhaps to display when the event occurred. This is described in the next section.

1.7 Displaying the date/time in messages

To display the date and time of an event, you would place '%(asctime)s' in your format string:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s')
logging.warning('is when this event was logged.')
```

which should print something like this:

```
2010-12-12 11:41:42,612 is when this event was logged.
```

The default format for date/time display (shown above) is ISO8601. If you need more control over the formatting of the date/time, provide a *datefmt* argument to `basicConfig`, as in this example:

```
import logging
logging.basicConfig(format='%(asctime)s %(message)s', datefmt='%m/%d/%Y %I:%M:%S %p')
logging.warning('is when this event was logged.')
```

which would display something like this:

```
12/12/2010 11:46:36 AM is when this event was logged.
```

The format of the *datefmt* argument is the same as supported by `time.strftime()`.

1.8 Next Steps

That concludes the basic tutorial. It should be enough to get you up and running with logging. There's a lot more that the logging package offers, but to get the best out of it, you'll need to invest a little more of your time in reading the following sections. If you're ready for that, grab some of your favourite beverage and carry on.

If your logging needs are simple, then use the above examples to incorporate logging into your own scripts, and if you run into problems or don't understand something, please post a question on the comp.lang.python Usenet group (available at https://groups.google.com/group/comp.lang.python) and you should receive help before too long.

Still here? You can carry on reading the next few sections, which provide a slightly more advanced/in-depth tutorial than the basic one above. After that, you can take a look at the logging-cookbook.

2 Advanced Logging Tutorial

The logging library takes a modular approach and offers several categories of components: loggers, handlers, filters, and formatters.

- Loggers expose the interface that application code directly uses.
- Handlers send the log records (created by loggers) to the appropriate destination.
- Filters provide a finer grained facility for determining which log records to output.
- Formatters specify the layout of log records in the final output.

Log event information is passed between loggers, handlers, filters and formatters in a `LogRecord` instance.

Logging is performed by calling methods on instances of the `Logger` class (hereafter called *loggers*). Each instance has a name, and they are conceptually arranged in a namespace hierarchy using dots (periods) as separators. For example, a logger named 'scan' is the parent of loggers 'scan.text', 'scan.html' and 'scan.pdf'. Logger names can be anything you want, and indicate the area of an application in which a logged message originates.

A good convention to use when naming loggers is to use a module-level logger, in each module which uses logging, named as follows:

```
logger = logging.getLogger(__name__)
```

This means that logger names track the package/module hierarchy, and it's intuitively obvious where events are logged just from the logger name.

The root of the hierarchy of loggers is called the root logger. That's the logger used by the functions `debug()`, `info()`, `warning()`, `error()` and `critical()`, which just call the same-named method of the root logger. The functions and the methods have the same signatures. The root logger's name is printed as 'root' in the logged output.

It is, of course, possible to log messages to different destinations. Support is included in the package for writing log messages to files, HTTP GET/POST locations, email via SMTP, generic sockets, or OS-specific logging mechanisms such as syslog or the Windows NT event log. Destinations are served by *handler* classes. You can create your own log destination class if you have special requirements not met by any of the built-in handler classes.

By default, no destination is set for any logging messages. You can specify a destination (such as console or file) by using `basicConfig()` as in the tutorial examples. If you call the functions `debug()`, `info()`, `warning()`, `error()` and `critical()`, they will check to see if no destination is set; and if one is not set, they will set a destination of the console (`sys.stderr`) and a default format for the displayed message before delegating to the root logger to do the actual message output.

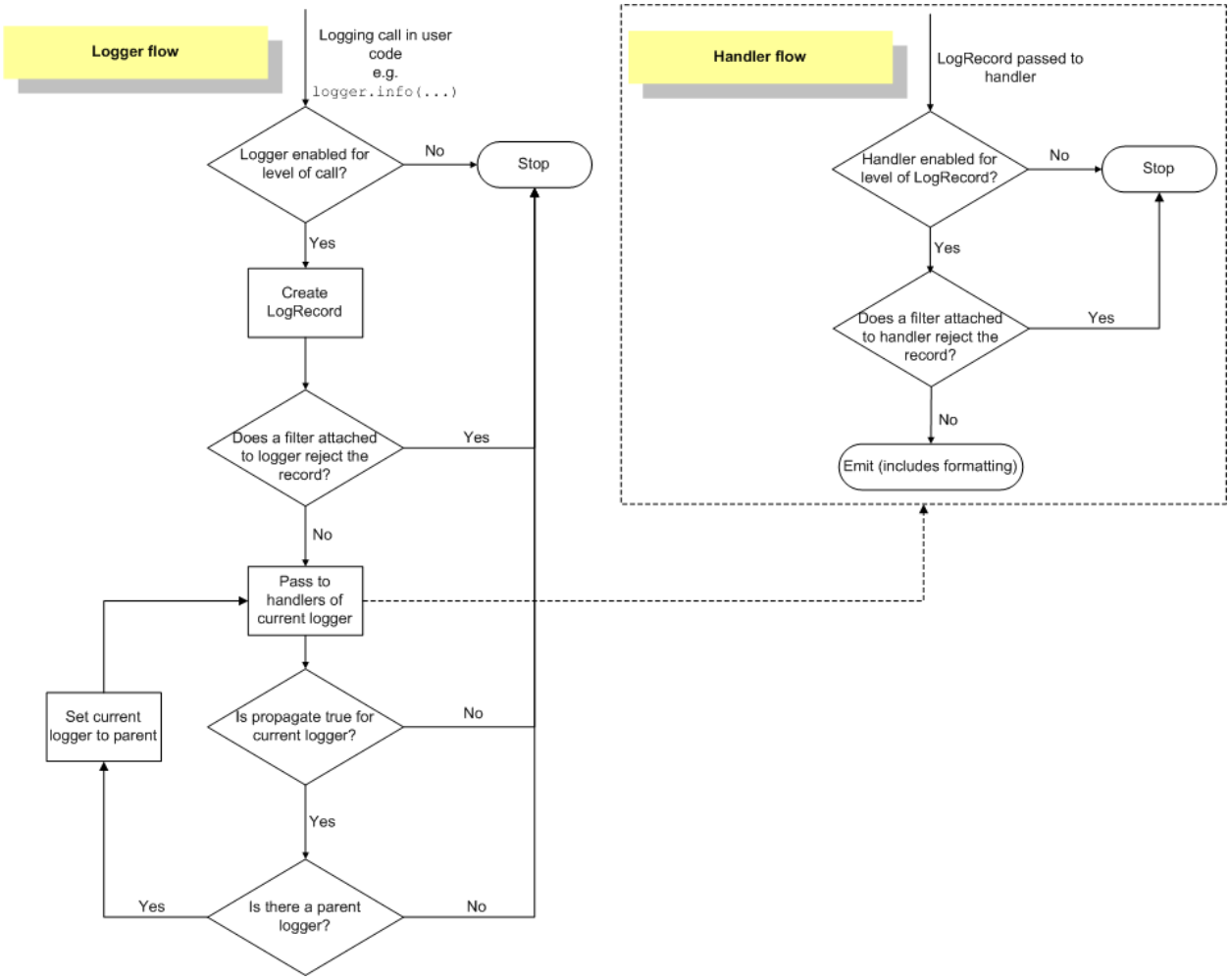
The default format set by `basicConfig()` for messages is:

```
severity:logger name:message
```

You can change this by passing a format string to `basicConfig()` with the *format* keyword argument. For all options regarding how a format string is constructed, see formatter-objects.

2.1 Logging Flow

The flow of log event information in loggers and handlers is illustrated in the following diagram.



2.2 Loggers

`Logger` objects have a threefold job. First, they expose several methods to application code so that applications can log messages at runtime. Second, logger objects determine which log messages to act upon based upon severity (the default filtering facility) or filter objects. Third, logger objects pass along relevant log messages to all interested log handlers.

The most widely used methods on logger objects fall into two categories: configuration and message sending.

These are the most common configuration methods:

- `Logger.setLevel()` specifies the lowest-severity log message a logger will handle, where debug is the lowest built-in severity level and critical is the highest built-in severity. For example, if the severity level is INFO, the logger will handle only INFO, WARNING, ERROR, and CRITICAL messages and will ignore DEBUG messages.
- `Logger.addHandler()` and `Logger.removeHandler()` add and remove handler objects from the logger object. Handlers are covered in more detail in *Handlers*.
- `Logger.addFilter()` and `Logger.removeFilter()` add and remove filter objects from the logger object. Filters are covered in more detail in filter.

You don't need to always call these methods on every logger you create. See the last two paragraphs in this section.

With the logger object configured, the following methods create log messages:

- `Logger.debug()`, `Logger.info()`, `Logger.warning()`, `Logger.error()`, and `Logger.critical()` all create log records with a message and a level that corresponds to their respective method names. The message is actually a format string, which may contain the standard string substitution syntax of `%s`, `%d`, `%f`, and so on. The rest of their arguments is a list of objects that correspond with the substitution fields in the message. With regard to `**kwargs`, the logging methods care only about a keyword of `exc_info` and use it to determine whether to log exception information.
- `Logger.exception()` creates a log message similar to `Logger.error()`. The difference is that `Logger.exception()` dumps a stack trace along with it. Call this method only from an exception handler.
- `Logger.log()` takes a log level as an explicit argument. This is a little more verbose for logging messages than using the log level convenience methods listed above, but this is how to log at custom log levels.

`getLogger()` returns a reference to a logger instance with the specified name if it is provided, or `root` if not. The names are period-separated hierarchical structures. Multiple calls to `getLogger()` with the same name will return a reference to the same logger object. Loggers that are further down in the hierarchical list are children of loggers higher up in the list. For example, given a logger with a name of `foo`, loggers with names of `foo.bar`, `foo.bar.baz`, and `foo.bam` are all descendants of `foo`.

Loggers have a concept of *effective level*. If a level is not explicitly set on a logger, the level of its parent is used instead as its effective level. If the parent has no explicit level set, *its* parent is examined, and so on - all ancestors are searched until an explicitly set level is found. The root logger always has an explicit level set (`WARNING` by default). When deciding whether to process an event, the effective level of the logger is used to determine whether the event is passed to the logger's handlers.

Child loggers propagate messages up to the handlers associated with their ancestor loggers. Because of this, it is unnecessary to define and configure handlers for all the loggers an application uses. It is sufficient to configure handlers for a top-level logger and create child loggers as needed. (You can, however, turn off propagation by setting the *propagate* attribute of a logger to `False`.)

2.3 Handlers

`Handler` objects are responsible for dispatching the appropriate log messages (based on the log messages' severity) to the handler's specified destination. `Logger` objects can add zero or more handler objects to themselves with an `addHandler()` method. As an example scenario, an application may want to send all log messages to a log file, all log messages of error or higher to stdout, and all messages of critical to an email address. This scenario requires three individual handlers where each handler is responsible for sending messages of a specific severity to a specific location.

The standard library includes quite a few handler types (see *Useful Handlers*); the tutorials use mainly `StreamHandler` and `FileHandler` in its examples.

There are very few methods in a handler for application developers to concern themselves with. The only handler methods that seem relevant for application developers who are using the built-in handler objects (that is, not creating custom handlers) are the following configuration methods:

- The `setLevel()` method, just as in logger objects, specifies the lowest severity that will be dispatched to the appropriate destination. Why are there two `setLevel()` methods? The level set in the logger determines which severity of messages it will pass to its handlers. The level set in each handler determines which messages that handler will send on.
- `setFormatter()` selects a Formatter object for this handler to use.
- `addFilter()` and `removeFilter()` respectively configure and deconfigure filter objects on handlers.

Application code should not directly instantiate and use instances of `Handler`. Instead, the `Handler` class is a base class that defines the interface that all handlers should have and establishes some default behavior that child classes can use (or override).

2.4 Formatters

Formatter objects configure the final order, structure, and contents of the log message. Unlike the base `logging.Handler` class, application code may instantiate formatter classes, although you could likely subclass the formatter if your application needs special behavior. The constructor takes two optional arguments – a message format string and a date format string.

`logging.Formatter.__init__`(*fmt=None*, *datefmt=None*)

If there is no message format string, the default is to use the raw message. If there is no date format string, the default date format is:

```
%Y-%m-%d %H:%M:%S
```

with the milliseconds tacked on at the end.

The message format string uses `%(<dictionary key>)s` styled string substitution; the possible keys are documented in logrecord-attributes.

The following message format string will log the time in a human-readable format, the severity of the message, and the contents of the message, in that order:

```
'%(asctime)s - %(levelname)s - %(message)s'
```

Formatters use a user-configurable function to convert the creation time of a record to a tuple. By default, `time.localtime()` is used; to change this for a particular formatter instance, set the `converter` attribute of the instance to a function with the same signature as `time.localtime()` or `time.gmtime()`. To change it for all formatters, for example if you want all logging times to be shown in GMT, set the `converter` attribute in the Formatter class (to `time.gmtime` for GMT display).

2.5 Configuring Logging

Programmers can configure logging in three ways:

1. Creating loggers, handlers, and formatters explicitly using Python code that calls the configuration methods listed above.
2. Creating a logging config file and reading it using the `fileConfig()` function.
3. Creating a dictionary of configuration information and passing it to the `dictConfig()` function.

For the reference documentation on the last two options, see logging-config-api. The following example configures a very simple logger, a console handler, and a simple formatter using Python code:

```
import logging

# create logger
logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)

# create console handler and set level to debug
ch = logging.StreamHandler()
ch.setLevel(logging.DEBUG)

# create formatter
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')

# add formatter to ch
ch.setFormatter(formatter)

# add ch to logger
logger.addHandler(ch)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Running this module from the command line produces the following output:

```
$ python simple_logging_module.py
2005-03-19 15:10:26,618 - simple_example - DEBUG - debug message
2005-03-19 15:10:26,620 - simple_example - INFO - info message
2005-03-19 15:10:26,695 - simple_example - WARNING - warn message
2005-03-19 15:10:26,697 - simple_example - ERROR - error message
2005-03-19 15:10:26,773 - simple_example - CRITICAL - critical message
```

The following Python module creates a logger, handler, and formatter nearly identical to those in the example listed above, with the only difference being the names of the objects:

```
import logging
import logging.config

logging.config.fileConfig('logging.conf')

# create logger
logger = logging.getLogger('simpleExample')

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Here is the logging.conf file:

```
[loggers]
keys=root,simpleExample
```

```
[handlers]
keys=consoleHandler

[formatters]
keys=simpleFormatter

[logger_root]
level=DEBUG
handlers=consoleHandler

[logger_simpleExample]
level=DEBUG
handlers=consoleHandler
qualname=simpleExample
propagate=0

[handler_consoleHandler]
class=StreamHandler
level=DEBUG
formatter=simpleFormatter
args=(sys.stdout,)

[formatter_simpleFormatter]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
datefmt=
```

The output is nearly identical to that of the non-config-file-based example:

```
$ python simple_logging_config.py
2005-03-19 15:38:55,977 - simpleExample - DEBUG - debug message
2005-03-19 15:38:55,979 - simpleExample - INFO - info message
2005-03-19 15:38:56,054 - simpleExample - WARNING - warn message
2005-03-19 15:38:56,055 - simpleExample - ERROR - error message
2005-03-19 15:38:56,130 - simpleExample - CRITICAL - critical message
```

You can see that the config file approach has a few advantages over the Python code approach, mainly separation of configuration and code and the ability of noncoders to easily modify the logging properties.

Warning: The `fileConfig()` function takes a default parameter, `disable_existing_loggers`, which defaults to `True` for reasons of backward compatibility. This may or may not be what you want, since it will cause any loggers existing before the `fileConfig()` call to be disabled unless they (or an ancestor) are explicitly named in the configuration. Please refer to the reference documentation for more information, and specify `False` for this parameter if you wish.

The dictionary passed to `dictConfig()` can also specify a Boolean value with key `disable_existing_loggers`, which if not specified explicitly in the dictionary also defaults to being interpreted as `True`. This leads to the logger-disabling behaviour described above, which may not be what you want - in which case, provide the key explicitly with a value of `False`.

Note that the class names referenced in config files need to be either relative to the logging module, or absolute values which can be resolved using normal import mechanisms. Thus, you could use either `WatchedFileHandler` (relative to the logging module) or `mypackage.mymodule.MyHandler` (for a class defined in package `mypackage` and module `mymodule`, where `mypackage` is available on the Python import path).

In Python 2.7, a new means of configuring logging has been introduced, using dictionaries to hold configuration information. This provides a superset of the functionality of the config-file-based approach outlined above, and is the recommended configuration method for new applications and deployments. Because a Python dictionary is used to hold configuration information, and since you can populate that dictionary using different means, you have more options for configuration. For example, you can use a configuration file in JSON format, or, if you have access to YAML processing functionality, a file in YAML format, to populate the configuration dictionary. Or, of course, you can construct the dictionary in Python code, receive it in pickled form over a socket, or use whatever approach makes sense for your application.

Here's an example of the same configuration as above, in YAML format for the new dictionary-based approach:

```
version: 1
formatters:
  simple:
    format: '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
handlers:
  console:
    class: logging.StreamHandler
    level: DEBUG
    formatter: simple
    stream: ext://sys.stdout
loggers:
  simpleExample:
    level: DEBUG
    handlers: [console]
    propagate: no
root:
  level: DEBUG
  handlers: [console]
```

For more information about logging using a dictionary, see logging-config-api.

2.6 What happens if no configuration is provided

If no logging configuration is provided, it is possible to have a situation where a logging event needs to be output, but no handlers can be found to output the event. The behaviour of the logging package in these circumstances is dependent on the Python version.

For Python 2.x, the behaviour is as follows:

- If *logging.raiseExceptions* is `False` (production mode), the event is silently dropped.
- If *logging.raiseExceptions* is `True` (development mode), a message 'No handlers could be found for logger X.Y.Z' is printed once.

2.7 Configuring Logging for a Library

When developing a library which uses logging, you should take care to document how the library uses logging - for example, the names of loggers used. Some consideration also needs to be given to its logging configuration. If the using application does not configure logging, and library code makes logging calls, then (as described in the previous section) an error message will be printed to `sys.stderr`.

If for some reason you *don't* want this message printed in the absence of any logging configuration, you can attach a do-nothing handler to the top-level logger for your library. This avoids the message being printed, since a handler will be always be found for the library's events: it just doesn't produce any output. If the library user configures logging for application use, presumably that configuration will add some handlers, and

if levels are suitably configured then logging calls made in library code will send output to those handlers, as normal.

A do-nothing handler is included in the logging package: `NullHandler` (since Python 2.7). An instance of this handler could be added to the top-level logger of the logging namespace used by the library (*if* you want to prevent an error message being output to `sys.stderr` in the absence of logging configuration). If all logging by a library *foo* is done using loggers with names matching 'foo.x', 'foo.x.y', etc. then the code:

```
import logging
logging.getLogger('foo').addHandler(logging.NullHandler())
```

should have the desired effect. If an organisation produces a number of libraries, then the logger name specified can be 'orgname.foo' rather than just 'foo'.

Note: It is strongly advised that you *do not add any handlers other than* `NullHandler` *to your library's loggers.* This is because the configuration of handlers is the prerogative of the application developer who uses your library. The application developer knows their target audience and what handlers are most appropriate for their application: if you add handlers 'under the hood', you might well interfere with their ability to carry out unit tests and deliver logs which suit their requirements.

---

3 Logging Levels

The numeric values of logging levels are given in the following table. These are primarily of interest if you want to define your own levels, and need them to have specific values relative to the predefined levels. If you define a level with the same numeric value, it overwrites the predefined value; the predefined name is lost.

Level	Numeric value
`CRITICAL`	50
`ERROR`	40
`WARNING`	30
`INFO`	20
`DEBUG`	10
`NOTSET`	0

Levels can also be associated with loggers, being set either by the developer or through loading a saved logging configuration. When a logging method is called on a logger, the logger compares its own level with the level associated with the method call. If the logger's level is higher than the method call's, no logging message is actually generated. This is the basic mechanism controlling the verbosity of logging output.

Logging messages are encoded as instances of the `LogRecord` class. When a logger decides to actually log an event, a `LogRecord` instance is created from the logging message.

Logging messages are subjected to a dispatch mechanism through the use of *handlers*, which are instances of subclasses of the `Handler` class. Handlers are responsible for ensuring that a logged message (in the form of a `LogRecord`) ends up in a particular location (or set of locations) which is useful for the target audience for that message (such as end users, support desk staff, system administrators, developers). Handlers are passed `LogRecord` instances intended for particular destinations. Each logger can have zero, one or more handlers associated with it (via the `addHandler()` method of `Logger`). In addition to any handlers directly associated with a logger, *all handlers associated with all ancestors of the logger* are called to dispatch the message (unless the *propagate* flag for a logger is set to a false value, at which point the passing to ancestor handlers stops).

Just as for loggers, handlers can have levels associated with them. A handler's level acts as a filter in the same way as a logger's level does. If a handler decides to actually dispatch an event, the `emit()` method is used to send the message to its destination. Most user-defined subclasses of `Handler` will need to override this `emit()`.

3.1 Custom Levels

Defining your own levels is possible, but should not be necessary, as the existing levels have been chosen on the basis of practical experience. However, if you are convinced that you need custom levels, great care should be exercised when doing this, and it is possibly *a very bad idea to define custom levels if you are developing a library*. That's because if multiple library authors all define their own custom levels, there is a chance that the logging output from such multiple libraries used together will be difficult for the using developer to control and/or interpret, because a given numeric value might mean different things for different libraries.

4 Useful Handlers

In addition to the base `Handler` class, many useful subclasses are provided:

1. `StreamHandler` instances send messages to streams (file-like objects).
2. `FileHandler` instances send messages to disk files.
3. `BaseRotatingHandler` is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use `RotatingFileHandler` or `TimedRotatingFileHandler`.
4. `RotatingFileHandler` instances send messages to disk files, with support for maximum log file sizes and log file rotation.
5. `TimedRotatingFileHandler` instances send messages to disk files, rotating the log file at certain timed intervals.
6. `SocketHandler` instances send messages to TCP/IP sockets.
7. `DatagramHandler` instances send messages to UDP sockets.
8. `SMTPHandler` instances send messages to a designated email address.
9. `SysLogHandler` instances send messages to a Unix syslog daemon, possibly on a remote machine.
10. `NTEventLogHandler` instances send messages to a Windows NT/2000/XP event log.
11. `MemoryHandler` instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
12. `HTTPHandler` instances send messages to an HTTP server using either `GET` or `POST` semantics.
13. `WatchedFileHandler` instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
14. `NullHandler` instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the 'No handlers could be found for logger XXX' message which can be displayed if the library user has not configured logging. See *Configuring Logging for a Library* for more information.

New in version 2.7: The `NullHandler` class.

The `NullHandler`, `StreamHandler` and `FileHandler` classes are defined in the core logging package. The other handlers are defined in a sub- module, `logging.handlers`. (There is also another sub-module, `logging.config`, for configuration functionality.)

Logged messages are formatted for presentation through instances of the `Formatter` class. They are initialized with a format string suitable for use with the % operator and a dictionary.

For formatting multiple messages in a batch, instances of `BufferingFormatter` can be used. In addition to the format string (which is applied to each message in the batch), there is provision for header and trailer format strings.

When filtering based on logger level and/or handler level is not enough, instances of `Filter` can be added to both `Logger` and `Handler` instances (through their `addFilter()` method). Before deciding to process a message further, both loggers and handlers consult all their filters for permission. If any filter returns a false value, the message is not processed further.

The basic `Filter` functionality allows filtering by specific logger name. If this feature is used, messages sent to the named logger and its children are allowed through the filter, and all others dropped.

5 Exceptions raised during logging

The logging package is designed to swallow exceptions which occur while logging in production. This is so that errors which occur while handling logging events - such as logging misconfiguration, network or other similar errors - do not cause the application using logging to terminate prematurely.

`SystemExit` and `KeyboardInterrupt` exceptions are never swallowed. Other exceptions which occur during the `emit()` method of a `Handler` subclass are passed to its `handleError()` method.

The default implementation of `handleError()` in `Handler` checks to see if a module-level variable, `raiseExceptions`, is set. If set, a traceback is printed to `sys.stderr`. If not set, the exception is swallowed.

Note: The default value of `raiseExceptions` is `True`. This is because during development, you typically want to be notified of any exceptions that occur. It's advised that you set `raiseExceptions` to `False` for production usage.

---

6 Using arbitrary objects as messages

In the preceding sections and examples, it has been assumed that the message passed when logging the event is a string. However, this is not the only possibility. You can pass an arbitrary object as a message, and its `__str__()` method will be called when the logging system needs to convert it to a string representation. In fact, if you want to, you can avoid computing a string representation altogether - for example, the `SocketHandler` emits an event by pickling it and sending it over the wire.

7 Optimization

Formatting of message arguments is deferred until it cannot be avoided. However, computing the arguments passed to the logging method can also be expensive, and you may want to avoid doing it if the logger will just throw away your event. To decide what to do, you can call the `isEnabledFor()` method which takes a level argument and returns true if the event would be created by the Logger for that level of call. You can write code like this:

```
if logger.isEnabledFor(logging.DEBUG):
    logger.debug('Message with %s, %s', expensive_func1(),
                                        expensive_func2())
```

so that if the logger's threshold is set above `DEBUG`, the calls to `expensive_func1()` and `expensive_func2()` are never made.

Note: In some cases, `isEnabledFor()` can itself be more expensive than you'd like (e.g. for deeply nested loggers where an explicit level is only set high up in the logger hierarchy). In such cases (or if you want to avoid calling a method in tight loops), you can cache the result of a call to `isEnabledFor()` in a local or instance variable, and use that instead of calling the method each time. Such a cached value would only need to be recomputed when the logging configuration changes dynamically while the application is running (which is not all that common).

There are other optimizations which can be made for specific applications which need more precise control over what logging information is collected. Here's a list of things you can do to avoid processing during logging which you don't need:

What you don't want to collect	How to avoid collecting it
Information about where calls were made from.	Set `logging._srcfile` to `None`. This avoids calling `sys._getframe()`, which may help to speed up your code in environments like PyPy (which can't speed up code that uses `sys._getframe()`).
Threading information.	Set `logging.logThreads` to `0`.
Process information.	Set `logging.logProcesses` to `0`.

Also note that the core logging module only includes the basic handlers. If you don't import `logging.handlers` and `logging.config`, they won't take up any memory.

See also:

Module `logging` API reference for the logging module.

Module `logging.config` Configuration API for the logging module.

Module `logging.handlers` Useful handlers included with the logging module.

A logging cookbook

Index

Symbols

Logging Cookbook

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Vinay Sajip <vinay_sajip at red-dove dot com>

This page contains a number of recipes related to logging, which have been found useful in the past.

1 Using logging in multiple modules

Multiple calls to `logging.getLogger('someLogger')` return a reference to the same logger object. This is true not only within the same module, but also across modules as long as it is in the same Python interpreter process. It is true for references to the same object; additionally, application code can define and configure a parent logger in one module and create (but not configure) a child logger in a separate module, and all logger calls to the child will pass up to the parent. Here is a main module:

```
import logging
import auxiliary_module

# create logger with 'spam_application'
logger = logging.getLogger('spam_application')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
fh.setFormatter(formatter)
ch.setFormatter(formatter)
# add the handlers to the logger
logger.addHandler(fh)
logger.addHandler(ch)

logger.info('creating an instance of auxiliary_module.Auxiliary')
a = auxiliary_module.Auxiliary()
logger.info('created an instance of auxiliary_module.Auxiliary')
logger.info('calling auxiliary_module.Auxiliary.do_something')
a.do_something()
logger.info('finished auxiliary_module.Auxiliary.do_something')
logger.info('calling auxiliary_module.some_function()')
auxiliary_module.some_function()
logger.info('done with auxiliary_module.some_function()')
```

Here is the auxiliary module:

```
import logging

# create logger
```

```
module_logger = logging.getLogger('spam_application.auxiliary')

class Auxiliary:
    def __init__(self):
        self.logger = logging.getLogger('spam_application.auxiliary.Auxiliary')
        self.logger.info('creating an instance of Auxiliary')

    def do_something(self):
        self.logger.info('doing something')
        a = 1 + 1
        self.logger.info('done doing something')

def some_function():
    module_logger.info('received a call to "some_function"')
```

The output looks like this:

```
2005-03-23 23:47:11,663 - spam_application - INFO -
   creating an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,665 - spam_application.auxiliary.Auxiliary - INFO -
   creating an instance of Auxiliary
2005-03-23 23:47:11,665 - spam_application - INFO -
   created an instance of auxiliary_module.Auxiliary
2005-03-23 23:47:11,668 - spam_application - INFO -
   calling auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,668 - spam_application.auxiliary.Auxiliary - INFO -
   doing something
2005-03-23 23:47:11,669 - spam_application.auxiliary.Auxiliary - INFO -
   done doing something
2005-03-23 23:47:11,670 - spam_application - INFO -
   finished auxiliary_module.Auxiliary.do_something
2005-03-23 23:47:11,671 - spam_application - INFO -
   calling auxiliary_module.some_function()
2005-03-23 23:47:11,672 - spam_application.auxiliary - INFO -
   received a call to 'some_function'
2005-03-23 23:47:11,673 - spam_application - INFO -
   done with auxiliary_module.some_function()
```

2 Logging from multiple threads

Logging from multiple threads requires no special effort. The following example shows logging from the main (initIal) thread and another thread:

```
import logging
import threading
import time

def worker(arg):
    while not arg['stop']:
        logging.debug('Hi from myfunc')
        time.sleep(0.5)

def main():
    logging.basicConfig(level=logging.DEBUG, format='%(relativeCreated)6d %(threadName)s
↪%(message)s')
    info = {'stop': False}
```

```
    thread = threading.Thread(target=worker, args=(info,))
    thread.start()
    while True:
        try:
            logging.debug('Hello from main')
            time.sleep(0.75)
        except KeyboardInterrupt:
            info['stop'] = True
            break
    thread.join()

if __name__ == '__main__':
    main()
```

When run, the script should print something like the following:

```
   0 Thread-1 Hi from myfunc
   3 MainThread Hello from main
 505 Thread-1 Hi from myfunc
 755 MainThread Hello from main
1007 Thread-1 Hi from myfunc
1507 MainThread Hello from main
1508 Thread-1 Hi from myfunc
2010 Thread-1 Hi from myfunc
2258 MainThread Hello from main
2512 Thread-1 Hi from myfunc
3009 MainThread Hello from main
3013 Thread-1 Hi from myfunc
3515 Thread-1 Hi from myfunc
3761 MainThread Hello from main
4017 Thread-1 Hi from myfunc
4513 MainThread Hello from main
4518 Thread-1 Hi from myfunc
```

This shows the logging output interspersed as one might expect. This approach works for more threads than shown here, of course.

3 Multiple handlers and formatters

Loggers are plain Python objects. The `addHandler()` method has no minimum or maximum quota for the number of handlers you may add. Sometimes it will be beneficial for an application to log all messages of all severities to a text file while simultaneously logging errors or above to the console. To set this up, simply configure the appropriate handlers. The logging calls in the application code will remain unchanged. Here is a slight modification to the previous simple module-based configuration example:

```
import logging

logger = logging.getLogger('simple_example')
logger.setLevel(logging.DEBUG)
# create file handler which logs even debug messages
fh = logging.FileHandler('spam.log')
fh.setLevel(logging.DEBUG)
# create console handler with a higher log level
ch = logging.StreamHandler()
ch.setLevel(logging.ERROR)
# create formatter and add it to the handlers
```

```
formatter = logging.Formatter('%(asctime)s - %(name)s - %(levelname)s - %(message)s')
ch.setFormatter(formatter)
fh.setFormatter(formatter)
# add the handlers to logger
logger.addHandler(ch)
logger.addHandler(fh)

# 'application' code
logger.debug('debug message')
logger.info('info message')
logger.warn('warn message')
logger.error('error message')
logger.critical('critical message')
```

Notice that the 'application' code does not care about multiple handlers. All that changed was the addition and configuration of a new handler named *fh*.

The ability to create new handlers with higher- or lower-severity filters can be very helpful when writing and testing an application. Instead of using many `print` statements for debugging, use `logger.debug`: Unlike the print statements, which you will have to delete or comment out later, the logger.debug statements can remain intact in the source code and remain dormant until you need them again. At that time, the only change that needs to happen is to modify the severity level of the logger and/or handler to debug.

4 Logging to multiple destinations

Let's say you want to log to console and file with different message formats and in differing circumstances. Say you want to log messages with levels of DEBUG and higher to file, and those messages at level INFO and higher to the console. Let's also assume that the file should contain timestamps, but the console messages should not. Here's how you can achieve this:

```
import logging

# set up logging to file - see previous section for more details
logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s %(name)-12s %(levelname)-8s %(message)s',
                    datefmt='%m-%d %H:%M',
                    filename='/temp/myapp.log',
                    filemode='w')
# define a Handler which writes INFO messages or higher to the sys.stderr
console = logging.StreamHandler()
console.setLevel(logging.INFO)
# set a format which is simpler for console use
formatter = logging.Formatter('%(name)-12s: %(levelname)-8s %(message)s')
# tell the handler to use this format
console.setFormatter(formatter)
# add the handler to the root logger
logging.getLogger('').addHandler(console)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')
```

```
logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

When you run this, on the console you will see

```
root        : INFO     Jackdaws love my big sphinx of quartz.
myapp.area1 : INFO     How quickly daft jumping zebras vex.
myapp.area2 : WARNING  Jail zesty vixen who grabbed pay from quack.
myapp.area2 : ERROR    The five boxing wizards jump quickly.
```

and in the file you will see something like

```
10-22 22:19 root         INFO     Jackdaws love my big sphinx of quartz.
10-22 22:19 myapp.area1  DEBUG    Quick zephyrs blow, vexing daft Jim.
10-22 22:19 myapp.area1  INFO     How quickly daft jumping zebras vex.
10-22 22:19 myapp.area2  WARNING  Jail zesty vixen who grabbed pay from quack.
10-22 22:19 myapp.area2  ERROR    The five boxing wizards jump quickly.
```

As you can see, the DEBUG message only shows up in the file. The other messages are sent to both destinations.

This example uses console and file handlers, but you can use any number and combination of handlers you choose.

5 Configuration server example

Here is an example of a module using the logging configuration server:

```
import logging
import logging.config
import time
import os

# read initial config file
logging.config.fileConfig('logging.conf')

# create and start listener on port 9999
t = logging.config.listen(9999)
t.start()

logger = logging.getLogger('simpleExample')

try:
    # loop through logging calls to see the difference
    # new configurations make, until Ctrl+C is pressed
    while True:
        logger.debug('debug message')
        logger.info('info message')
        logger.warn('warn message')
        logger.error('error message')
        logger.critical('critical message')
        time.sleep(5)
except KeyboardInterrupt:
```

```
    # cleanup
    logging.config.stopListening()
    t.join()
```

And here is a script that takes a filename and sends that file to the server, properly preceded with the binary-encoded length, as the new logging configuration:

```
#!/usr/bin/env python
import socket, sys, struct

with open(sys.argv[1], 'rb') as f:
    data_to_send = f.read()

HOST = 'localhost'
PORT = 9999
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
print('connecting...')
s.connect((HOST, PORT))
print('sending config...')
s.send(struct.pack('>L', len(data_to_send)))
s.send(data_to_send)
s.close()
print('complete')
```

6 Sending and receiving logging events across a network

Let's say you want to send logging events across a network, and handle them at the receiving end. A simple way of doing this is attaching a `SocketHandler` instance to the root logger at the sending end:

```
import logging, logging.handlers

rootLogger = logging.getLogger('')
rootLogger.setLevel(logging.DEBUG)
socketHandler = logging.handlers.SocketHandler('localhost',
                    logging.handlers.DEFAULT_TCP_LOGGING_PORT)
# don't bother with a formatter, since a socket handler sends the event as
# an unformatted pickle
rootLogger.addHandler(socketHandler)

# Now, we can log to the root logger, or any other logger. First the root...
logging.info('Jackdaws love my big sphinx of quartz.')

# Now, define a couple of other loggers which might represent areas in your
# application:

logger1 = logging.getLogger('myapp.area1')
logger2 = logging.getLogger('myapp.area2')

logger1.debug('Quick zephyrs blow, vexing daft Jim.')
logger1.info('How quickly daft jumping zebras vex.')
logger2.warning('Jail zesty vixen who grabbed pay from quack.')
logger2.error('The five boxing wizards jump quickly.')
```

At the receiving end, you can set up a receiver using the `SocketServer` module. Here is a basic working example:

```
import pickle
import logging
import logging.handlers
import SocketServer
import struct


class LogRecordStreamHandler(SocketServer.StreamRequestHandler):
    """Handler for a streaming logging request.

    This basically logs the record using whatever logging policy is
    configured locally.
    """

    def handle(self):
        """
        Handle multiple requests - each expected to be a 4-byte length,
        followed by the LogRecord in pickle format. Logs the record
        according to whatever policy is configured locally.
        """
        while True:
            chunk = self.connection.recv(4)
            if len(chunk) < 4:
                break
            slen = struct.unpack('>L', chunk)[0]
            chunk = self.connection.recv(slen)
            while len(chunk) < slen:
                chunk = chunk + self.connection.recv(slen - len(chunk))
            obj = self.unPickle(chunk)
            record = logging.makeLogRecord(obj)
            self.handleLogRecord(record)

    def unPickle(self, data):
        return pickle.loads(data)

    def handleLogRecord(self, record):
        # if a name is specified, we use the named logger rather than the one
        # implied by the record.
        if self.server.logname is not None:
            name = self.server.logname
        else:
            name = record.name
        logger = logging.getLogger(name)
        # N.B. EVERY record gets logged. This is because Logger.handle
        # is normally called AFTER logger-level filtering. If you want
        # to do filtering, do it at the client end to save wasting
        # cycles and network bandwidth!
        logger.handle(record)

class LogRecordSocketReceiver(SocketServer.ThreadingTCPServer):
    """
    Simple TCP socket-based logging receiver suitable for testing.
    """

    allow_reuse_address = 1

    def __init__(self, host='localhost',
                 port=logging.handlers.DEFAULT_TCP_LOGGING_PORT,
```

```
                    handler=LogRecordStreamHandler):
        SocketServer.ThreadingTCPServer.__init__(self, (host, port), handler)
        self.abort = 0
        self.timeout = 1
        self.logname = None

    def serve_until_stopped(self):
        import select
        abort = 0
        while not abort:
            rd, wr, ex = select.select([self.socket.fileno()],
                                       [], [],
                                       self.timeout)
            if rd:
                self.handle_request()
            abort = self.abort

def main():
    logging.basicConfig(
        format='%(relativeCreated)5d %(name)-15s %(levelname)-8s %(message)s')
    tcpserver = LogRecordSocketReceiver()
    print('About to start TCP server...')
    tcpserver.serve_until_stopped()

if __name__ == '__main__':
    main()
```

First run the server, and then the client. On the client side, nothing is printed on the console; on the server side, you should see something like:

```
About to start TCP server...
   59 root            INFO     Jackdaws love my big sphinx of quartz.
   59 myapp.area1     DEBUG    Quick zephyrs blow, vexing daft Jim.
   69 myapp.area1     INFO     How quickly daft jumping zebras vex.
   69 myapp.area2     WARNING  Jail zesty vixen who grabbed pay from quack.
   69 myapp.area2     ERROR    The five boxing wizards jump quickly.
```

Note that there are some security issues with pickle in some scenarios. If these affect you, you can use an alternative serialization scheme by overriding the `makePickle()` method and implementing your alternative there, as well as adapting the above script to use your alternative serialization.

7 Adding contextual information to your logging output

Sometimes you want logging output to contain contextual information in addition to the parameters passed to the logging call. For example, in a networked application, it may be desirable to log client-specific information in the log (e.g. remote client's username, or IP address). Although you could use the *extra* parameter to achieve this, it's not always convenient to pass the information in this way. While it might be tempting to create `Logger` instances on a per-connection basis, this is not a good idea because these instances are not garbage collected. While this is not a problem in practice, when the number of `Logger` instances is dependent on the level of granularity you want to use in logging an application, it could be hard to manage if the number of `Logger` instances becomes effectively unbounded.

7.1 Using LoggerAdapters to impart contextual information

An easy way in which you can pass contextual information to be output along with logging event information is to use the `LoggerAdapter` class. This class is designed to look like a `Logger`, so that you can call `debug()`, `info()`, `warning()`, `error()`, `exception()`, `critical()` and `log()`. These methods have the same signatures as their counterparts in `Logger`, so you can use the two types of instances interchangeably.

When you create an instance of `LoggerAdapter`, you pass it a `Logger` instance and a dict-like object which contains your contextual information. When you call one of the logging methods on an instance of `LoggerAdapter`, it delegates the call to the underlying instance of `Logger` passed to its constructor, and arranges to pass the contextual information in the delegated call. Here's a snippet from the code of `LoggerAdapter`:

```
def debug(self, msg, *args, **kwargs):
    """
    Delegate a debug call to the underlying logger, after adding
    contextual information from this adapter instance.
    """
    msg, kwargs = self.process(msg, kwargs)
    self.logger.debug(msg, *args, **kwargs)
```

The `process()` method of `LoggerAdapter` is where the contextual information is added to the logging output. It's passed the message and keyword arguments of the logging call, and it passes back (potentially) modified versions of these to use in the call to the underlying logger. The default implementation of this method leaves the message alone, but inserts an 'extra' key in the keyword argument whose value is the dict-like object passed to the constructor. Of course, if you had passed an 'extra' keyword argument in the call to the adapter, it will be silently overwritten.

The advantage of using 'extra' is that the values in the dict-like object are merged into the `LogRecord` instance's __dict__, allowing you to use customized strings with your `Formatter` instances which know about the keys of the dict-like object. If you need a different method, e.g. if you want to prepend or append the contextual information to the message string, you just need to subclass `LoggerAdapter` and override `process()` to do what you need. Here is a simple example:

```
class CustomAdapter(logging.LoggerAdapter):
    """
    This example adapter expects the passed in dict-like object to have a
    'connid' key, whose value in brackets is prepended to the log message.
    """
    def process(self, msg, kwargs):
        return '[%s] %s' % (self.extra['connid'], msg), kwargs
```

which you can use like this:

```
logger = logging.getLogger(__name__)
adapter = CustomAdapter(logger, {'connid': some_conn_id})
```

Then any events that you log to the adapter will have the value of `some_conn_id` prepended to the log messages.

Using objects other than dicts to pass contextual information

You don't need to pass an actual dict to a `LoggerAdapter` - you could pass an instance of a class which implements `__getitem__` and `__iter__` so that it looks like a dict to logging. This would be useful if you want to generate values dynamically (whereas the values in a dict would be constant).

7.2 Using Filters to impart contextual information

You can also add contextual information to log output using a user-defined `Filter`. `Filter` instances are allowed to modify the `LogRecords` passed to them, including adding additional attributes which can then be output using a suitable format string, or if needed a custom `Formatter`.

For example in a web application, the request being processed (or at least, the interesting parts of it) can be stored in a threadlocal (`threading.local`) variable, and then accessed from a `Filter` to add, say, information from the request - say, the remote IP address and remote user's username - to the `LogRecord`, using the attribute names 'ip' and 'user' as in the `LoggerAdapter` example above. In that case, the same format string can be used to get similar output to that shown above. Here's an example script:

```
import logging
from random import choice

class ContextFilter(logging.Filter):
    """
    This is a filter which injects contextual information into the log.

    Rather than use actual contextual information, we just use random
    data in this demo.
    """

    USERS = ['jim', 'fred', 'sheila']
    IPS = ['123.231.231.123', '127.0.0.1', '192.168.0.1']

    def filter(self, record):

        record.ip = choice(ContextFilter.IPS)
        record.user = choice(ContextFilter.USERS)
        return True

if __name__ == '__main__':
    levels = (logging.DEBUG, logging.INFO, logging.WARNING, logging.ERROR, logging.CRITICAL)
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)-15s %(name)-5s %(levelname)-8s IP: %(ip)-15s User:
↪%(user)-8s %(message)s')
    a1 = logging.getLogger('a.b.c')
    a2 = logging.getLogger('d.e.f')

    f = ContextFilter()
    a1.addFilter(f)
    a2.addFilter(f)
    a1.debug('A debug message')
    a1.info('An info message with %s', 'some parameters')
    for x in range(10):
        lvl = choice(levels)
        lvlname = logging.getLevelName(lvl)
        a2.log(lvl, 'A message at %s level with %d %s', lvlname, 2, 'parameters')
```

which, when run, produces something like:

```
2010-09-06 22:38:15,292 a.b.c DEBUG    IP: 123.231.231.123 User: fred     A debug message
2010-09-06 22:38:15,300 a.b.c INFO     IP: 192.168.0.1     User: sheila   An info message with␣
↪some parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A message at CRITICAL␣
↪level with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 127.0.0.1       User: jim      A message at ERROR level␣
↪with 2 parameters
```

```
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 127.0.0.1       User: sheila   A message at DEBUG level␣
↪with 2 parameters
2010-09-06 22:38:15,300 d.e.f ERROR    IP: 123.231.231.123 User: fred     A message at ERROR level␣
↪with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 192.168.0.1     User: jim      A message at CRITICAL␣
↪level with 2 parameters
2010-09-06 22:38:15,300 d.e.f CRITICAL IP: 127.0.0.1       User: sheila   A message at CRITICAL␣
↪level with 2 parameters
2010-09-06 22:38:15,300 d.e.f DEBUG    IP: 192.168.0.1     User: jim      A message at DEBUG level␣
↪with 2 parameters
2010-09-06 22:38:15,301 d.e.f ERROR    IP: 127.0.0.1       User: sheila   A message at ERROR level␣
↪with 2 parameters
2010-09-06 22:38:15,301 d.e.f DEBUG    IP: 123.231.231.123 User: fred     A message at DEBUG level␣
↪with 2 parameters
2010-09-06 22:38:15,301 d.e.f INFO     IP: 123.231.231.123 User: fred     A message at INFO level␣
↪with 2 parameters
```

8 Logging to a single file from multiple processes

Although logging is thread-safe, and logging to a single file from multiple threads in a single process *is* supported, logging to a single file from *multiple processes* is *not* supported, because there is no standard way to serialize access to a single file across multiple processes in Python. If you need to log to a single file from multiple processes, one way of doing this is to have all the processes log to a `SocketHandler`, and have a separate process which implements a socket server which reads from the socket and logs to file. (If you prefer, you can dedicate one thread in one of the existing processes to perform this function.) *This section* documents this approach in more detail and includes a working socket receiver which can be used as a starting point for you to adapt in your own applications.

If you are using a recent version of Python which includes the `multiprocessing` module, you could write your own handler which uses the `Lock` class from this module to serialize access to the file from your processes. The existing `FileHandler` and subclasses do not make use of `multiprocessing` at present, though they may do so in the future. Note that at present, the `multiprocessing` module does not provide working lock functionality on all platforms (see https://bugs.python.org/issue3770).

9 Using file rotation

Sometimes you want to let a log file grow to a certain size, then open a new file and log to that. You may want to keep a certain number of these files, and when that many files have been created, rotate the files so that the number of files and the size of the files both remain bounded. For this usage pattern, the logging package provides a `RotatingFileHandler`:

```
import glob
import logging
import logging.handlers

LOG_FILENAME = 'logging_rotatingfile_example.out'

# Set up a specific logger with our desired output level
my_logger = logging.getLogger('MyLogger')
my_logger.setLevel(logging.DEBUG)

# Add the log message handler to the logger
handler = logging.handlers.RotatingFileHandler(
```

```
                LOG_FILENAME, maxBytes=20, backupCount=5)

my_logger.addHandler(handler)

# Log some messages
for i in range(20):
    my_logger.debug('i = %d' % i)

# See what files are created
logfiles = glob.glob('%s*' % LOG_FILENAME)

for filename in logfiles:
    print(filename)
```

The result should be 6 separate files, each with part of the log history for the application:

```
logging_rotatingfile_example.out
logging_rotatingfile_example.out.1
logging_rotatingfile_example.out.2
logging_rotatingfile_example.out.3
logging_rotatingfile_example.out.4
logging_rotatingfile_example.out.5
```

The most current file is always `logging_rotatingfile_example.out`, and each time it reaches the size limit it is renamed with the suffix `.1`. Each of the existing backup files is renamed to increment the suffix (`.1` becomes `.2`, etc.) and the `.6` file is erased.

Obviously this example sets the log length much too small as an extreme example. You would want to set *maxBytes* to an appropriate value.

10 An example dictionary-based configuration

Below is an example of a logging configuration dictionary - it's taken from the documentation on the Django project. This dictionary is passed to `dictConfig()` to put the configuration into effect:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': True,
    'formatters': {
        'verbose': {
            'format': '%(levelname)s %(asctime)s %(module)s %(process)d %(thread)d %(message)s'
        },
        'simple': {
            'format': '%(levelname)s %(message)s'
        },
    },
    'filters': {
        'special': {
            '()': 'project.logging.SpecialFilter',
            'foo': 'bar',
        }
    },
    'handlers': {
        'null': {
            'level':'DEBUG',
            'class':'django.utils.log.NullHandler',
```

```
        },
        'console':{
            'level':'DEBUG',
            'class':'logging.StreamHandler',
            'formatter': 'simple'
        },
        'mail_admins': {
            'level': 'ERROR',
            'class': 'django.utils.log.AdminEmailHandler',
            'filters': ['special']
        }
    },
    'loggers': {
        'django': {
            'handlers':['null'],
            'propagate': True,
            'level':'INFO',
        },
        'django.request': {
            'handlers': ['mail_admins'],
            'level': 'ERROR',
            'propagate': False,
        },
        'myproject.custom': {
            'handlers': ['console', 'mail_admins'],
            'level': 'INFO',
            'filters': ['special']
        }
    }
}
```

For more information about this configuration, you can see the relevant section of the Django documentation.

11 Inserting a BOM into messages sent to a SysLogHandler

RFC 5424 requires that a Unicode message be sent to a syslog daemon as a set of bytes which have the following structure: an optional pure-ASCII component, followed by a UTF-8 Byte Order Mark (BOM), followed by Unicode encoded using UTF-8. (See the relevant section of the specification.)

In Python 2.6 and 2.7, code was added to `SysLogHandler` to insert a BOM into the message, but unfortunately, it was implemented incorrectly, with the BOM appearing at the beginning of the message and hence not allowing any pure-ASCII component to appear before it.

As this behaviour is broken, the incorrect BOM insertion code is being removed from Python 2.7.4 and later. However, it is not being replaced, and if you want to produce RFC 5424-compliant messages which include a BOM, an optional pure-ASCII sequence before it and arbitrary Unicode after it, encoded using UTF-8, then you need to do the following:

1. Attach a `Formatter` instance to your `SysLogHandler` instance, with a format string such as:

   ```
   u'ASCII section\ufeffUnicode section'
   ```

   The Unicode code point `u'\ufeff'`, when encoded using UTF-8, will be encoded as a UTF-8 BOM – the byte-string `'\xef\xbb\xbf'`.

2. Replace the ASCII section with whatever placeholders you like, but make sure that the data that appears in there after substitution is always ASCII (that way, it will remain unchanged after UTF-8

encoding).

3. Replace the Unicode section with whatever placeholders you like; if the data which appears there after substitution contains characters outside the ASCII range, that's fine – it will be encoded using UTF-8.

If the formatted message is Unicode, it *will* be encoded using UTF-8 encoding by `SysLogHandler`. If you follow the above rules, you should be able to produce RFC 5424-compliant messages. If you don't, logging may not complain, but your messages will not be RFC 5424-compliant, and your syslog daemon may complain.

12 Implementing structured logging

Although most logging messages are intended for reading by humans, and thus not readily machine-parseable, there might be circumstances where you want to output messages in a structured format which *is* capable of being parsed by a program (without needing complex regular expressions to parse the log message). This is straightforward to achieve using the logging package. There are a number of ways in which this could be achieved, but the following is a simple approach which uses JSON to serialise the event in a machine-parseable manner:

```
import json
import logging

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        return '%s >>> %s' % (self.message, json.dumps(self.kwargs))

_ = StructuredMessage   # optional, to improve readability

logging.basicConfig(level=logging.INFO, format='%(message)s')
logging.info(_('message 1', foo='bar', bar='baz', num=123, fnum=123.456))
```

If the above script is run, it prints:

```
message 1 >>> {"fnum": 123.456, "num": 123, "bar": "baz", "foo": "bar"}
```

Note that the order of items might be different according to the version of Python used.

If you need more specialised processing, you can use a custom JSON encoder, as in the following complete example:

```
from __future__ import unicode_literals

import json
import logging

# This next bit is to ensure the script runs unchanged on 2.x and 3.x
try:
    unicode
except NameError:
    unicode = str

class Encoder(json.JSONEncoder):
    def default(self, o):
```

```
        if isinstance(o, set):
            return tuple(o)
        elif isinstance(o, unicode):
            return o.encode('unicode_escape').decode('ascii')
        return super(Encoder, self).default(o)

class StructuredMessage(object):
    def __init__(self, message, **kwargs):
        self.message = message
        self.kwargs = kwargs

    def __str__(self):
        s = Encoder().encode(self.kwargs)
        return '%s >>> %s' % (self.message, s)

_ = StructuredMessage   # optional, to improve readability

def main():
    logging.basicConfig(level=logging.INFO, format='%(message)s')
    logging.info(_('message 1', set_value=set([1, 2, 3]), snowman='\u2603'))

if __name__ == '__main__':
    main()
```

When the above script is run, it prints:

```
message 1 >>> {"snowman": "\u2603", "set_value": [1, 2, 3]}
```

Note that the order of items might be different according to the version of Python used.

13 Customizing handlers with `dictConfig()`

There are times when you want to customize logging handlers in particular ways, and if you use `dictConfig()` you may be able to do this without subclassing. As an example, consider that you may want to set the ownership of a log file. On POSIX, this is easily done using `os.chown()`, but the file handlers in the stdlib don't offer built-in support. You can customize handler creation using a plain function such as:

```
def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        import os, pwd, grp
        # convert user and group names to uid and gid
        uid = pwd.getpwnam(owner[0]).pw_uid
        gid = grp.getgrnam(owner[1]).gr_gid
        owner = (uid, gid)
        if not os.path.exists(filename):
            open(filename, 'a').close()
        os.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)
```

You can then specify, in a logging configuration passed to `dictConfig()`, that a logging handler be created by calling this function:

```
LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
```

```
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}
```

In this example I am setting the ownership using the `pulse` user and group, just for the purposes of illustration. Putting it together into a working script, `chowntest.py`:

```
import logging, logging.config, os, shutil

def owned_file_handler(filename, mode='a', encoding=None, owner=None):
    if owner:
        if not os.path.exists(filename):
            open(filename, 'a').close()
        shutil.chown(filename, *owner)
    return logging.FileHandler(filename, mode, encoding)

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'default': {
            'format': '%(asctime)s %(levelname)s %(name)s %(message)s'
        },
    },
    'handlers': {
        'file':{
            # The values below are popped from this dictionary and
            # used to create the handler, set the handler's level and
            # its formatter.
            '()': owned_file_handler,
            'level':'DEBUG',
            'formatter': 'default',
            # The values below are passed to the handler creator callable
            # as keyword arguments.
            'owner': ['pulse', 'pulse'],
            'filename': 'chowntest.log',
```

```
            'mode': 'w',
            'encoding': 'utf-8',
        },
    },
    'root': {
        'handlers': ['file'],
        'level': 'DEBUG',
    },
}

logging.config.dictConfig(LOGGING)
logger = logging.getLogger('mylogger')
logger.debug('A debug message')
```

To run this, you will probably need to run as `root`:

```
$ sudo python3.3 chowntest.py
$ cat chowntest.log
2013-11-05 09:34:51,128 DEBUG mylogger A debug message
$ ls -l chowntest.log
-rw-r--r-- 1 pulse pulse 55 2013-11-05 09:34 chowntest.log
```

Note that this example uses Python 3.3 because that's where `shutil.chown()` makes an appearance. This approach should work with any Python version that supports `dictConfig()` - namely, Python 2.7, 3.2 or later. With pre-3.3 versions, you would need to implement the actual ownership change using e.g. `os.chown()`.

In practice, the handler-creating function may be in a utility module somewhere in your project. Instead of the line in the configuration:

```
'()': owned_file_handler,
```

you could use e.g.:

```
'()': 'ext://project.util.owned_file_handler',
```

where `project.util` can be replaced with the actual name of the package where the function resides. In the above working script, using `'ext://__main__.owned_file_handler'` should work. Here, the actual callable is resolved by `dictConfig()` from the `ext://` specification.

This example hopefully also points the way to how you could implement other types of file change - e.g. setting specific POSIX permission bits - in the same way, using `os.chmod()`.

Of course, the approach could also be extended to types of handler other than a `FileHandler` - for example, one of the rotating file handlers, or a different type of handler altogether.

14 Configuring filters with `dictConfig()`

You *can* configure filters using `dictConfig()`, though it might not be obvious at first glance how to do it (hence this recipe). Since `Filter` is the only filter class included in the standard library, and it is unlikely to cater to many requirements (it's only there as a base class), you will typically need to define your own `Filter` subclass with an overridden `filter()` method. To do this, specify the `()` key in the configuration dictionary for the filter, specifying a callable which will be used to create the filter (a class is the most obvious, but you can provide any callable which returns a `Filter` instance). Here is a complete example:

```
import logging
import logging.config
import sys

class MyFilter(logging.Filter):
    def __init__(self, param=None):
        self.param = param

    def filter(self, record):
        if self.param is None:
            allow = True
        else:
            allow = self.param not in record.msg
        if allow:
            record.msg = 'changed: ' + record.msg
        return allow

LOGGING = {
    'version': 1,
    'filters': {
        'myfilter': {
            '()': MyFilter,
            'param': 'noshow',
        }
    },
    'handlers': {
        'console': {
            'class': 'logging.StreamHandler',
            'filters': ['myfilter']
        }
    },
    'root': {
        'level': 'DEBUG',
        'handlers': ['console']
    },
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.debug('hello')
    logging.debug('hello - noshow')
```

This example shows how you can pass configuration data to the callable which constructs the instance, in the form of keyword parameters. When run, the above script will print:

```
changed: hello
```

which shows that the filter is working as configured.

A couple of extra points to note:

- If you can't refer to the callable directly in the configuration (e.g. if it lives in a different module, and you can't import it directly where the configuration dictionary is), you can use the form `ext://...` as described in logging-config-dict-externalobj. For example, you could have used the text `'ext://__main__.MyFilter'` instead of `MyFilter` in the above example.
- As well as for filters, this technique can also be used to configure custom handlers and formatters. See logging-config-dict-userdef for more information on how logging supports using user-defined objects in its configuration, and see the other cookbook recipe *Customizing handlers with dictConfig()* above.

15 Customized exception formatting

There might be times when you want to do customized exception formatting - for argument's sake, let's say you want exactly one line per logged event, even when exception information is present. You can do this with a custom formatter class, as shown in the following example:

```
import logging

class OneLineExceptionFormatter(logging.Formatter):
    def formatException(self, exc_info):
        """
        Format an exception so that it prints on a single line.
        """
        result = super(OneLineExceptionFormatter, self).formatException(exc_info)
        return repr(result) # or format into one line however you want to

    def format(self, record):
        s = super(OneLineExceptionFormatter, self).format(record)
        if record.exc_text:
            s = s.replace('\n', '') + '|'
        return s

def configure_logging():
    fh = logging.FileHandler('output.txt', 'w')
    f = OneLineExceptionFormatter('%(asctime)s|%(levelname)s|%(message)s|',
                                  '%d/%m/%Y %H:%M:%S')
    fh.setFormatter(f)
    root = logging.getLogger()
    root.setLevel(logging.DEBUG)
    root.addHandler(fh)

def main():
    configure_logging()
    logging.info('Sample message')
    try:
        x = 1 / 0
    except ZeroDivisionError as e:
        logging.exception('ZeroDivisionError: %s', e)

if __name__ == '__main__':
    main()
```

When run, this produces a file with exactly two lines:

```
28/01/2015 07:21:23|INFO|Sample message|
28/01/2015 07:21:23|ERROR|ZeroDivisionError: integer division or modulo by zero|'Traceback (most␣
↪recent call last):\n  File "logtest7.py", line 30, in main\n    x = 1 / 0\nZeroDivisionError:␣
↪integer division or modulo by zero'|
```

While the above treatment is simplistic, it points the way to how exception information can be formatted to your liking. The `traceback` module may be helpful for more specialized needs.

16 Speaking logging messages

There might be situations when it is desirable to have logging messages rendered in an audible rather than a visible format. This is easy to do if you have text-to-speech (TTS) functionality available in your system,

even if it doesn't have a Python binding. Most TTS systems have a command line program you can run, and this can be invoked from a handler using `subprocess`. It's assumed here that TTS command line programs won't expect to interact with users or take a long time to complete, and that the frequency of logged messages will be not so high as to swamp the user with messages, and that it's acceptable to have the messages spoken one at a time rather than concurrently, The example implementation below waits for one message to be spoken before the next is processed, and this might cause other handlers to be kept waiting. Here is a short example showing the approach, which assumes that the `espeak` TTS package is available:

```
import logging
import subprocess
import sys

class TTSHandler(logging.Handler):
    def emit(self, record):
        msg = self.format(record)
        # Speak slowly in a female English voice
        cmd = ['espeak', '-s150', '-ven+f3', msg]
        p = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                             stderr=subprocess.STDOUT)
        # wait for the program to finish
        p.communicate()

def configure_logging():
    h = TTSHandler()
    root = logging.getLogger()
    root.addHandler(h)
    # the default formatter just returns the message
    root.setLevel(logging.DEBUG)

def main():
    logging.info('Hello')
    logging.debug('Goodbye')

if __name__ == '__main__':
    configure_logging()
    sys.exit(main())
```

When run, this script should say "Hello" and then "Goodbye" in a female voice.

The above approach can, of course, be adapted to other TTS systems and even other systems altogether which can process messages via external programs run from a command line.

17 Buffering logging messages and outputting them conditionally

There might be situations where you want to log messages in a temporary area and only output them if a certain condition occurs. For example, you may want to start logging debug events in a function, and if the function completes without errors, you don't want to clutter the log with the collected debug information, but if there is an error, you want all the debug information to be output as well as the error.

Here is an example which shows how you could do this using a decorator for your functions where you want logging to behave this way. It makes use of the `logging.handlers.MemoryHandler`, which allows buffering of logged events until some condition occurs, at which point the buffered events are `flushed` - passed to another handler (the `target` handler) for processing. By default, the `MemoryHandler` flushed when its buffer gets filled up or an event whose level is greater than or equal to a specified threshold is seen. You can use this recipe with a more specialised subclass of `MemoryHandler` if you want custom flushing behavior.

The example script has a simple function, `foo`, which just cycles through all the logging levels, writing to

`sys.stderr` to say what level it's about to log at, and then actually logging a message at that level. You can pass a parameter to `foo` which, if true, will log at ERROR and CRITICAL levels - otherwise, it only logs at DEBUG, INFO and WARNING levels.

The script just arranges to decorate `foo` with a decorator which will do the conditional logging that's required. The decorator takes a logger as a parameter and attaches a memory handler for the duration of the call to the decorated function. The decorator can be additionally parameterised using a target handler, a level at which flushing should occur, and a capacity for the buffer. These default to a `StreamHandler` which writes to `sys.stderr`, `logging.ERROR` and `100` respectively.

Here's the script:

```
import logging
from logging.handlers import MemoryHandler
import sys

logger = logging.getLogger(__name__)
logger.addHandler(logging.NullHandler())

def log_if_errors(logger, target_handler=None, flush_level=None, capacity=None):
    if target_handler is None:
        target_handler = logging.StreamHandler()
    if flush_level is None:
        flush_level = logging.ERROR
    if capacity is None:
        capacity = 100
    handler = MemoryHandler(capacity, flushLevel=flush_level, target=target_handler)

    def decorator(fn):
        def wrapper(*args, **kwargs):
            logger.addHandler(handler)
            try:
                return fn(*args, **kwargs)
            except Exception:
                logger.exception('call failed')
                raise
            finally:
                super(MemoryHandler, handler).flush()
                logger.removeHandler(handler)
        return wrapper

    return decorator

def write_line(s):
    sys.stderr.write('%s\n' % s)

def foo(fail=False):
    write_line('about to log at DEBUG ...')
    logger.debug('Actually logged at DEBUG')
    write_line('about to log at INFO ...')
    logger.info('Actually logged at INFO')
    write_line('about to log at WARNING ...')
    logger.warning('Actually logged at WARNING')
    if fail:
        write_line('about to log at ERROR ...')
        logger.error('Actually logged at ERROR')
        write_line('about to log at CRITICAL ...')
        logger.critical('Actually logged at CRITICAL')
    return fail
```

```
decorated_foo = log_if_errors(logger)(foo)

if __name__ == '__main__':
    logger.setLevel(logging.DEBUG)
    write_line('Calling undecorated foo with False')
    assert not foo(False)
    write_line('Calling undecorated foo with True')
    assert foo(True)
    write_line('Calling decorated foo with False')
    assert not decorated_foo(False)
    write_line('Calling decorated foo with True')
    assert decorated_foo(True)
```

When this script is run, the following output should be observed:

```
Calling undecorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling undecorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
about to log at CRITICAL ...
Calling decorated foo with False
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
Calling decorated foo with True
about to log at DEBUG ...
about to log at INFO ...
about to log at WARNING ...
about to log at ERROR ...
Actually logged at DEBUG
Actually logged at INFO
Actually logged at WARNING
Actually logged at ERROR
about to log at CRITICAL ...
Actually logged at CRITICAL
```

As you can see, actual logging output only occurs when an event is logged whose severity is ERROR or greater, but in that case, any previous events at lower severities are also logged.

You can of course use the conventional means of decoration:

```
@log_if_errors(logger)
def foo(fail=False):
    ...
```

18 Formatting times using UTC (GMT) via configuration

Sometimes you want to format times using UTC, which can be done using a class such as *UTCFormatter*, shown below:

```
import logging
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime
```

and you can then use the `UTCFormatter` in your code instead of `Formatter`. If you want to do that via configuration, you can use the `dictConfig()` API with an approach illustrated by the following complete example:

```
import logging
import logging.config
import time

class UTCFormatter(logging.Formatter):
    converter = time.gmtime

LOGGING = {
    'version': 1,
    'disable_existing_loggers': False,
    'formatters': {
        'utc': {
            '()': UTCFormatter,
            'format': '%(asctime)s %(message)s',
        },
        'local': {
            'format': '%(asctime)s %(message)s',
        }
    },
    'handlers': {
        'console1': {
            'class': 'logging.StreamHandler',
            'formatter': 'utc',
        },
        'console2': {
            'class': 'logging.StreamHandler',
            'formatter': 'local',
        },
    },
    'root': {
        'handlers': ['console1', 'console2'],
   }
}

if __name__ == '__main__':
    logging.config.dictConfig(LOGGING)
    logging.warning('The local time is %s', time.asctime())
```

When this script is run, it should print something like:

```
2015-10-17 12:53:29,501 The local time is Sat Oct 17 13:53:29 2015
2015-10-17 13:53:29,501 The local time is Sat Oct 17 13:53:29 2015
```

showing how the time is formatted both as local time and UTC, one for each handler.

19 Using a context manager for selective logging

There are times when it would be useful to temporarily change the logging configuration and revert it back after doing something. For this, a context manager is the most obvious way of saving and restoring the logging context. Here is a simple example of such a context manager, which allows you to optionally change the logging level and add a logging handler purely in the scope of the context manager:

```
import logging
import sys

class LoggingContext(object):
    def __init__(self, logger, level=None, handler=None, close=True):
        self.logger = logger
        self.level = level
        self.handler = handler
        self.close = close

    def __enter__(self):
        if self.level is not None:
            self.old_level = self.logger.level
            self.logger.setLevel(self.level)
        if self.handler:
            self.logger.addHandler(self.handler)

    def __exit__(self, et, ev, tb):
        if self.level is not None:
            self.logger.setLevel(self.old_level)
        if self.handler:
            self.logger.removeHandler(self.handler)
        if self.handler and self.close:
            self.handler.close()
        # implicit return of None => don't swallow exceptions
```

If you specify a level value, the logger's level is set to that value in the scope of the with block covered by the context manager. If you specify a handler, it is added to the logger on entry to the block and removed on exit from the block. You can also ask the manager to close the handler for you on block exit - you could do this if you don't need the handler any more.

To illustrate how it works, we can add the following block of code to the above:

```
if __name__ == '__main__':
    logger = logging.getLogger('foo')
    logger.addHandler(logging.StreamHandler())
    logger.setLevel(logging.INFO)
    logger.info('1. This should appear just once on stderr.')
    logger.debug('2. This should not appear.')
    with LoggingContext(logger, level=logging.DEBUG):
        logger.debug('3. This should appear once on stderr.')
    logger.debug('4. This should not appear.')
    h = logging.StreamHandler(sys.stdout)
    with LoggingContext(logger, level=logging.DEBUG, handler=h, close=True):
        logger.debug('5. This should appear twice - once on stderr and once on stdout.')
    logger.info('6. This should appear just once on stderr.')
    logger.debug('7. This should not appear.')
```

We initially set the logger's level to `INFO`, so message #1 appears and message #2 doesn't. We then change the level to `DEBUG` temporarily in the following `with` block, and so message #3 appears. After the block exits, the logger's level is restored to `INFO` and so message #4 doesn't appear. In the next `with` block, we

set the level to `DEBUG` again but also add a handler writing to `sys.stdout`. Thus, message #5 appears twice on the console (once via `stderr` and once via `stdout`). After the `with` statement's completion, the status is as it was before so message #6 appears (like message #1) whereas message #7 doesn't (just like message #2).

If we run the resulting script, the result is as follows:

```
$ python logctx.py
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

If we run it again, but pipe `stderr` to `/dev/null`, we see the following, which is the only message written to `stdout`:

```
$ python logctx.py 2>/dev/null
5. This should appear twice - once on stderr and once on stdout.
```

Once again, but piping `stdout` to `/dev/null`, we get:

```
$ python logctx.py >/dev/null
1. This should appear just once on stderr.
3. This should appear once on stderr.
5. This should appear twice - once on stderr and once on stdout.
6. This should appear just once on stderr.
```

In this case, the message #5 printed to `stdout` doesn't appear, as expected.

Of course, the approach described here can be generalised, for example to attach logging filters temporarily. Note that the above code works in Python 2 as well as Python 3.

Porting Python 2 Code to Python 3

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

author Brett Cannon

Abstract

With Python 3 being the future of Python while Python 2 is still in active use, it is good to have your project available for both major releases of Python. This guide is meant to help you figure out how best to support both Python 2 & 3 simultaneously.

If you are looking to port an extension module instead of pure Python code, please see cporting-howto.

If you would like to read one core Python developer's take on why Python 3 came into existence, you can read Nick Coghlan's Python 3 Q & A or Brett Cannon's Why Python 3 exists.

For help with porting, you can email the python-porting mailing list with questions.

1 The Short Explanation

To make your project be single-source Python 2/3 compatible, the basic steps are:

1. Only worry about supporting Python 2.7
2. Make sure you have good test coverage (coverage.py can help; `pip install coverage`)
3. Learn the differences between Python 2 & 3
4. Use Futurize (or Modernize) to update your code (e.g. `pip install future`)
5. Use Pylint to help make sure you don't regress on your Python 3 support (`pip install pylint`)
6. Use caniusepython3 to find out which of your dependencies are blocking your use of Python 3 (`pip install caniusepython3`)
7. Once your dependencies are no longer blocking you, use continuous integration to make sure you stay compatible with Python 2 & 3 (tox can help test against multiple versions of Python; `pip install tox`)
8. Consider using optional static type checking to make sure your type usage works in both Python 2 & 3 (e.g. use mypy to check your typing under both Python 2 & Python 3).

2 Details

A key point about supporting Python 2 & 3 simultaneously is that you can start **today**! Even if your dependencies are not supporting Python 3 yet that does not mean you can't modernize your code **now** to support Python 3. Most changes required to support Python 3 lead to cleaner code using newer practices even in Python 2 code.

Another key point is that modernizing your Python 2 code to also support Python 3 is largely automated for you. While you might have to make some API decisions thanks to Python 3 clarifying text data versus binary data, the lower-level work is now mostly done for you and thus can at least benefit from the automated changes immediately.

Keep those key points in mind while you read on about the details of porting your code to support Python 2 & 3 simultaneously.

2.1 Drop support for Python 2.6 and older

While you can make Python 2.5 work with Python 3, it is **much** easier if you only have to work with Python 2.7. If dropping Python 2.5 is not an option then the six project can help you support Python 2.5 & 3 simultaneously (`pip install six`). Do realize, though, that nearly all the projects listed in this HOWTO will not be available to you.

If you are able to skip Python 2.5 and older, then the required changes to your code should continue to look and feel like idiomatic Python code. At worst you will have to use a function instead of a method in some instances or have to import a function instead of using a built-in one, but otherwise the overall transformation should not feel foreign to you.

But you should aim for only supporting Python 2.7. Python 2.6 is no longer freely supported and thus is not receiving bugfixes. This means **you** will have to work around any issues you come across with Python 2.6. There are also some tools mentioned in this HOWTO which do not support Python 2.6 (e.g., Pylint), and this will become more commonplace as time goes on. It will simply be easier for you if you only support the versions of Python that you have to support.

2.2 Make sure you specify the proper version support in your `setup.py` file

In your `setup.py` file you should have the proper trove classifier specifying what versions of Python you support. As your project does not support Python 3 yet you should at least have `Programming Language :: Python :: 2 :: Only` specified. Ideally you should also specify each major/minor version of Python that you do support, e.g. `Programming Language :: Python :: 2.7`.

2.3 Have good test coverage

Once you have your code supporting the oldest version of Python 2 you want it to, you will want to make sure your test suite has good coverage. A good rule of thumb is that if you want to be confident enough in your test suite that any failures that appear after having tools rewrite your code are actual bugs in the tools and not in your code. If you want a number to aim for, try to get over 80% coverage (and don't feel bad if you find it hard to get better than 90% coverage). If you don't already have a tool to measure test coverage then coverage.py is recommended.

2.4 Learn the differences between Python 2 & 3

Once you have your code well-tested you are ready to begin porting your code to Python 3! But to fully understand how your code is going to change and what you want to look out for while you code, you will want to learn what changes Python 3 makes in terms of Python 2. Typically the two best ways of doing that is reading the "What's New" doc for each release of Python 3 and the Porting to Python 3 book (which is free online). There is also a handy cheat sheet from the Python-Future project.

2.5 Update your code

Once you feel like you know what is different in Python 3 compared to Python 2, it's time to update your code! You have a choice between two tools in porting your code automatically: Futurize and Modernize. Which tool you choose will depend on how much like Python 3 you want your code to be. Futurize does its best to make Python 3 idioms and practices exist in Python 2, e.g. backporting the `bytes` type from Python 3 so that you have semantic parity between the major versions of Python. Modernize, on the other hand, is more conservative and targets a Python 2/3 subset of Python, directly relying on six to help provide compatibility. As Python 3 is the future, it might be best to consider Futurize to begin adjusting to any new practices that Python 3 introduces which you are not accustomed to yet.

Regardless of which tool you choose, they will update your code to run under Python 3 while staying compatible with the version of Python 2 you started with. Depending on how conservative you want to be, you may want to run the tool over your test suite first and visually inspect the diff to make sure the transformation is accurate. After you have transformed your test suite and verified that all the tests still pass as expected, then you can transform your application code knowing that any tests which fail is a translation failure.

Unfortunately the tools can't automate everything to make your code work under Python 3 and so there are a handful of things you will need to update manually to get full Python 3 support (which of these steps are necessary vary between the tools). Read the documentation for the tool you choose to use to see what it fixes by default and what it can do optionally to know what will (not) be fixed for you and what you may have to fix on your own (e.g. using `io.open()` over the built-in `open()` function is off by default in Modernize). Luckily, though, there are only a couple of things to watch out for which can be considered large issues that may be hard to debug if not watched for.

Division

In Python 3, `5 / 2 == 2.5` and not 2; all division between `int` values result in a `float`. This change has actually been planned since Python 2.2 which was released in 2002. Since then users have been encouraged to add `from __future__ import division` to any and all files which use the `/` and `//` operators or to be running the interpreter with the `-Q` flag. If you have not been doing this then you will need to go through your code and do two things:

1. Add `from __future__ import division` to your files
2. Update any division operator as necessary to either use `//` to use floor division or continue using `/` and expect a float

The reason that `/` isn't simply translated to `//` automatically is that if an object defines a `__truediv__` method but not `__floordiv__` then your code would begin to fail (e.g. a user-defined class that uses `/` to signify some operation but not `//` for the same thing or at all).

Text versus binary data

In Python 2 you could use the `str` type for both text and binary data. Unfortunately this confluence of two different concepts could lead to brittle code which sometimes worked for either kind of data, sometimes not. It also could lead to confusing APIs if people didn't explicitly state that something that accepted `str` accepted either text or binary data instead of one specific type. This complicated the situation especially for anyone supporting multiple languages as APIs wouldn't bother explicitly supporting `unicode` when they claimed text data support.

To make the distinction between text and binary data clearer and more pronounced, Python 3 did what most languages created in the age of the internet have done and made text and binary data distinct types that cannot blindly be mixed together (Python predates widespread access to the internet). For any code that deals only with text or only binary data, this separation doesn't pose an issue. But for code that has to deal with both, it does mean you might have to now care about when you are using text compared to binary data, which is why this cannot be entirely automated.

To start, you will need to decide which APIs take text and which take binary (it is **highly** recommended you don't design APIs that can take both due to the difficulty of keeping the code working; as stated earlier it is difficult to do well). In Python 2 this means making sure the APIs that take text can work with `unicode` and those that work with binary data work with the `bytes` type from Python 3 (which is a subset of `str` in Python 2 and acts as an alias for `bytes` type in Python 2). Usually the biggest issue is realizing which methods exist on which types in Python 2 & 3 simultaneously (for text that's `unicode` in Python 2 and `str` in Python 3, for binary that's `str`/`bytes` in Python 2 and `bytes` in Python 3). The following table lists the **unique** methods of each data type across Python 2 & 3 (e.g., the `decode()` method is usable on the equivalent binary data type in either Python 2 or 3, but it can't be used by the textual data type consistently between Python 2 and 3 because `str` in Python 3 doesn't have the method). Do note that as of Python 3.5 the `__mod__` method was added to the bytes type.

Text data	Binary data
	decode
encode	
format	
isdecimal	
isnumeric	

Making the distinction easier to handle can be accomplished by encoding and decoding between binary data and text at the edge of your code. This means that when you receive text in binary data, you should immediately decode it. And if your code needs to send text as binary data then encode it as late as possible.

This allows your code to work with only text internally and thus eliminates having to keep track of what type of data you are working with.

The next issue is making sure you know whether the string literals in your code represent text or binary data. You should add a `b` prefix to any literal that presents binary data. For text you should add a `u` prefix to the text literal. (there is a `__future__` import to force all unspecified literals to be Unicode, but usage has shown it isn't as effective as adding a `b` or `u` prefix to all literals explicitly)

As part of this dichotomy you also need to be careful about opening files. Unless you have been working on Windows, there is a chance you have not always bothered to add the `b` mode when opening a binary file (e.g., `rb` for binary reading). Under Python 3, binary files and text files are clearly distinct and mutually incompatible; see the `io` module for details. Therefore, you **must** make a decision of whether a file will be used for binary access (allowing binary data to be read and/or written) or textual access (allowing text data to be read and/or written). You should also use `io.open()` for opening files instead of the built-in `open()` function as the `io` module is consistent from Python 2 to 3 while the built-in `open()` function is not (in Python 3 it's actually `io.open()`). Do not bother with the outdated practice of using `codecs.open()` as that's only necessary for keeping compatibility with Python 2.5.

The constructors of both `str` and `bytes` have different semantics for the same arguments between Python 2 & 3. Passing an integer to `bytes` in Python 2 will give you the string representation of the integer: `bytes(3) == '3'`. But in Python 3, an integer argument to `bytes` will give you a bytes object as long as the integer specified, filled with null bytes: `bytes(3) == b'\x00\x00\x00'`. A similar worry is necessary when passing a bytes object to `str`. In Python 2 you just get the bytes object back: `str(b'3') == b'3'`. But in Python 3 you get the string representation of the bytes object: `str(b'3') == "b'3'"`.

Finally, the indexing of binary data requires careful handling (slicing does **not** require any special handling). In Python 2, `b'123'[1] == b'2'` while in Python 3 `b'123'[1] == 50`. Because binary data is simply a collection of binary numbers, Python 3 returns the integer value for the byte you index on. But in Python 2 because `bytes == str`, indexing returns a one-item slice of bytes. The six project has a function named `six.indexbytes()` which will return an integer like in Python 3: `six.indexbytes(b'123', 1)`.

To summarize:

1. Decide which of your APIs take text and which take binary data
2. Make sure that your code that works with text also works with `unicode` and code for binary data works with `bytes` in Python 2 (see the table above for what methods you cannot use for each type)
3. Mark all binary literals with a `b` prefix, textual literals with a `u` prefix
4. Decode binary data to text as soon as possible, encode text as binary data as late as possible
5. Open files using `io.open()` and make sure to specify the `b` mode when appropriate
6. Be careful when indexing into binary data

Use feature detection instead of version detection

Inevitably you will have code that has to choose what to do based on what version of Python is running. The best way to do this is with feature detection of whether the version of Python you're running under supports what you need. If for some reason that doesn't work then you should make the version check be against Python 2 and not Python 3. To help explain this, let's look at an example.

Let's pretend that you need access to a feature of importlib that is available in Python's standard library since Python 3.3 and available for Python 2 through importlib2 on PyPI. You might be tempted to write code to access e.g. the `importlib.abc` module by doing the following:

```
import sys

if sys.version_info[0] == 3:
    from importlib import abc
```

```
else:
    from importlib2 import abc
```

The problem with this code is what happens when Python 4 comes out? It would be better to treat Python 2 as the exceptional case instead of Python 3 and assume that future Python versions will be more compatible with Python 3 than Python 2:

```
import sys

if sys.version_info[0] > 2:
    from importlib import abc
else:
    from importlib2 import abc
```

The best solution, though, is to do no version detection at all and instead rely on feature detection. That avoids any potential issues of getting the version detection wrong and helps keep you future-compatible:

```
try:
    from importlib import abc
except ImportError:
    from importlib2 import abc
```

2.6 Prevent compatibility regressions

Once you have fully translated your code to be compatible with Python 3, you will want to make sure your code doesn't regress and stop working under Python 3. This is especially true if you have a dependency which is blocking you from actually running under Python 3 at the moment.

To help with staying compatible, any new modules you create should have at least the following block of code at the top of it:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function
```

You can also run Python 2 with the `-3` flag to be warned about various compatibility issues your code triggers during execution. If you turn warnings into errors with `-Werror` then you can make sure that you don't accidentally miss a warning.

You can also use the Pylint project and its `--py3k` flag to lint your code to receive warnings when your code begins to deviate from Python 3 compatibility. This also prevents you from having to run Modernize or Futurize over your code regularly to catch compatibility regressions. This does require you only support Python 2.7 and Python 3.4 or newer as that is Pylint's minimum Python version support.

2.7 Check which dependencies block your transition

After you have made your code compatible with Python 3 you should begin to care about whether your dependencies have also been ported. The caniusepython3 project was created to help you determine which projects – directly or indirectly – are blocking you from supporting Python 3. There is both a command-line tool as well as a web interface at https://caniusepython3.com.

The project also provides code which you can integrate into your test suite so that you will have a failing test when you no longer have dependencies blocking you from using Python 3. This allows you to avoid having to manually check your dependencies and to be notified quickly when you can start running on Python 3.

2.8 Update your `setup.py` file to denote Python 3 compatibility

Once your code works under Python 3, you should update the classifiers in your `setup.py` to contain `Programming Language :: Python :: 3` and to not specify sole Python 2 support. This will tell anyone using your code that you support Python 2 **and** 3. Ideally you will also want to add classifiers for each major/minor version of Python you now support.

2.9 Use continuous integration to stay compatible

Once you are able to fully run under Python 3 you will want to make sure your code always works under both Python 2 & 3. Probably the best tool for running your tests under multiple Python interpreters is tox. You can then integrate tox with your continuous integration system so that you never accidentally break Python 2 or 3 support.

You may also want to use the `-bb` flag with the Python 3 interpreter to trigger an exception when you are comparing bytes to strings or bytes to an int (the latter is available starting in Python 3.5). By default type-differing comparisons simply return `False`, but if you made a mistake in your separation of text/binary data handling or indexing on bytes you wouldn't easily find the mistake. This flag will raise an exception when these kinds of comparisons occur, making the mistake much easier to track down.

And that's mostly it! At this point your code base is compatible with both Python 2 and 3 simultaneously. Your testing will also be set up so that you don't accidentally break Python 2 or 3 compatibility regardless of which version you typically run your tests under while developing.

2.10 Consider using optional static type checking

Another way to help port your code is to use a static type checker like mypy or pytype on your code. These tools can be used to analyze your code as if it's being run under Python 2, then you can run the tool a second time as if your code is running under Python 3. By running a static type checker twice like this you can discover if you're e.g. misusing binary data type in one version of Python compared to another. If you add optional type hints to your code you can also explicitly state whether your APIs use textual or binary data, helping to make sure everything functions as expected in both versions of Python.

Regular Expression HOWTO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author A.M. Kuchling <amk@amk.ca>

Abstract

This document is an introductory tutorial to using regular expressions in Python with the `re` module. It provides a gentler introduction than the corresponding section in the Library Reference.

1 Introduction

The `re` module was added in Python 1.5, and provides Perl-style regular expression patterns. Earlier versions of Python came with the `regex` module, which provided Emacs-style patterns. The `regex` module was removed completely in Python 2.5.

Regular expressions (called REs, or regexes, or regex patterns) are essentially a tiny, highly specialized programming language embedded inside Python and made available through the `re` module. Using this little language, you specify the rules for the set of possible strings that you want to match; this set might contain English sentences, or e-mail addresses, or TeX commands, or anything you like. You can then ask questions such as "Does this string match the pattern?", or "Is there a match for the pattern anywhere in this string?". You can also use REs to modify a string or to split it apart in various ways.

Regular expression patterns are compiled into a series of bytecodes which are then executed by a matching engine written in C. For advanced use, it may be necessary to pay careful attention to how the engine will execute a given RE, and write the RE in a certain way in order to produce bytecode that runs faster. Optimization isn't covered in this document, because it requires that you have a good understanding of the matching engine's internals.

The regular expression language is relatively small and restricted, so not all possible string processing tasks can be done using regular expressions. There are also tasks that *can* be done with regular expressions, but the expressions turn out to be very complicated. In these cases, you may be better off writing Python code to do the processing; while Python code will be slower than an elaborate regular expression, it will also probably be more understandable.

2 Simple Patterns

We'll start by learning about the simplest possible regular expressions. Since regular expressions are used to operate on strings, we'll begin with the most common task: matching characters.

For a detailed explanation of the computer science underlying regular expressions (deterministic and non-deterministic finite automata), you can refer to almost any textbook on writing compilers.

2.1 Matching Characters

Most letters and characters will simply match themselves. For example, the regular expression `test` will match the string `test` exactly. (You can enable a case-insensitive mode that would let this RE match `Test` or `TEST` as well; more about this later.)

There are exceptions to this rule; some characters are special *metacharacters*, and don't match themselves. Instead, they signal that some out-of-the-ordinary thing should be matched, or they affect other portions of the RE by repeating them or changing their meaning. Much of this document is devoted to discussing various metacharacters and what they do.

Here's a complete list of the metacharacters; their meanings will be discussed in the rest of this HOWTO.

```
. ^ $ * + ? { } [ ] \ | ( )
```

The first metacharacters we'll look at are `[` and `]`. They're used for specifying a character class, which is a set of characters that you wish to match. Characters can be listed individually, or a range of characters can be indicated by giving two characters and separating them by a `'-'`. For example, `[abc]` will match any of the characters `a`, `b`, or `c`; this is the same as `[a-c]`, which uses a range to express the same set of characters. If you wanted to match only lowercase letters, your RE would be `[a-z]`.

Metacharacters are not active inside classes. For example, `[akm$]` will match any of the characters `'a'`, `'k'`, `'m'`, or `'$'`; `'$'` is usually a metacharacter, but inside a character class it's stripped of its special nature.

You can match the characters not listed within the class by *complementing* the set. This is indicated by including a `'^'` as the first character of the class; `'^'` outside a character class will simply match the `'^'` character. For example, `[^5]` will match any character except `'5'`.

Perhaps the most important metacharacter is the backslash, `\`. As in Python string literals, the backslash can be followed by various characters to signal various special sequences. It's also used to escape all the metacharacters so you can still match them in patterns; for example, if you need to match a `[` or `\`, you can precede them with a backslash to remove their special meaning: `\[` or `\\`.

Some of the special sequences beginning with `'\'` represent predefined sets of characters that are often useful, such as the set of digits, the set of letters, or the set of anything that isn't whitespace. The following predefined special sequences are a subset of those available. The equivalent classes are for byte string patterns. For a complete list of sequences and expanded class definitions for Unicode string patterns, see the last part of Regular Expression Syntax.

`\d` Matches any decimal digit; this is equivalent to the class `[0-9]`.

`\D` Matches any non-digit character; this is equivalent to the class `[^0-9]`.

`\s` Matches any whitespace character; this is equivalent to the class `[ \t\n\r\f\v]`.

`\S` Matches any non-whitespace character; this is equivalent to the class `[^ \t\n\r\f\v]`.

`\w` Matches any alphanumeric character; this is equivalent to the class `[a-zA-Z0-9_]`.

`\W` Matches any non-alphanumeric character; this is equivalent to the class `[^a-zA-Z0-9_]`.

These sequences can be included inside a character class. For example, `[\s,.]` is a character class that will match any whitespace character, or `','` or `'.'`.

The final metacharacter in this section is `.`. It matches anything except a newline character, and there's an alternate mode (`re.DOTALL`) where it will match even a newline. `'.'` is often used where you want to match "any character".

2.2 Repeating Things

Being able to match varying sets of characters is the first thing regular expressions can do that isn't already possible with the methods available on strings. However, if that was the only additional capability of regexes, they wouldn't be much of an advance. Another capability is that you can specify that portions of the RE must be repeated a certain number of times.

The first metacharacter for repeating things that we'll look at is `*`. `*` doesn't match the literal character `*`; instead, it specifies that the previous character can be matched zero or more times, instead of exactly once.

For example, `ca*t` will match `ct` (0 `a` characters), `cat` (1 `a`), `caaat` (3 `a` characters), and so forth. The RE engine has various internal limitations stemming from the size of C's `int` type that will prevent it from matching over 2 billion `a` characters; you probably don't have enough memory to construct a string that large, so you shouldn't run into that limit.

Repetitions such as * are *greedy*; when repeating a RE, the matching engine will try to repeat it as many times as possible. If later portions of the pattern don't match, the matching engine will then back up and try again with fewer repetitions.

A step-by-step example will make this more obvious. Let's consider the expression `a[bcd]*b`. This matches the letter `'a'`, zero or more letters from the class `[bcd]`, and finally ends with a `'b'`. Now imagine matching this RE against the string `abcbd`.

Step	Matched	Explanation
1	`a`	The `a` in the RE matches.
2	`abcbd`	The engine matches `[bcd]*`, going as far as it can, which is to the end of the string.
3	*Failure*	The engine tries to match `b`, but the current position is at the end of the string, so it fails.
4	`abcb`	Back up, so that `[bcd]*` matches one less character.
5	*Failure*	Try `b` again, but the current position is at the last character, which is a `'d'`.
6	`abc`	Back up again, so that `[bcd]*` is only matching `bc`.
6	`abcb`	Try `b` again. This time the character at the current position is `'b'`, so it succeeds.

The end of the RE has now been reached, and it has matched `abcb`. This demonstrates how the matching engine goes as far as it can at first, and if no match is found it will then progressively back up and retry the rest of the RE again and again. It will back up until it has tried zero matches for `[bcd]*`, and if that subsequently fails, the engine will conclude that the string doesn't match the RE at all.

Another repeating metacharacter is `+`, which matches one or more times. Pay careful attention to the difference between `*` and `+`; `*` matches *zero* or more times, so whatever's being repeated may not be present at all, while `+` requires at least *one* occurrence. To use a similar example, `ca+t` will match `cat` (1 `a`), `caaat` (3 `a`'s), but won't match `ct`.

There are two more repeating qualifiers. The question mark character, `?`, matches either once or zero times; you can think of it as marking something as being optional. For example, `home-?brew` matches either `homebrew` or `home-brew`.

The most complicated repeated qualifier is `{m,n}`, where m and n are decimal integers. This qualifier means there must be at least m repetitions, and at most n. For example, `a/{1,3}b` will match `a/b`, `a//b`, and `a///b`. It won't match `ab`, which has no slashes, or `a////b`, which has four.

You can omit either m or n; in that case, a reasonable value is assumed for the missing value. Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity — actually, the upper bound is the 2-billion limit mentioned earlier, but that might as well be infinity.

Readers of a reductionist bent may notice that the three other qualifiers can all be expressed using this notation. `{0,}` is the same as `*`, `{1,}` is equivalent to `+`, and `{0,1}` is the same as `?`. It's better to use `*`, `+`, or `?` when you can, simply because they're shorter and easier to read.

3 Using Regular Expressions

Now that we've looked at some simple regular expressions, how do we actually use them in Python? The `re` module provides an interface to the regular expression engine, allowing you to compile REs into objects and then perform matches with them.

3.1 Compiling Regular Expressions

Regular expressions are compiled into pattern objects, which have methods for various operations such as searching for pattern matches or performing string substitutions.

```
>>> import re
>>> p = re.compile('ab*')
>>> p
<_sre.SRE_Pattern object at 0x...>
```

`re.compile()` also accepts an optional *flags* argument, used to enable various special features and syntax variations. We'll go over the available settings later, but for now a single example will do:

```
>>> p = re.compile('ab*', re.IGNORECASE)
```

The RE is passed to `re.compile()` as a string. REs are handled as strings because regular expressions aren't part of the core Python language, and no special syntax was created for expressing them. (There are applications that don't need REs at all, so there's no need to bloat the language specification by including them.) Instead, the `re` module is simply a C extension module included with Python, just like the `socket` or `zlib` modules.

Putting REs in strings keeps the Python language simpler, but has one disadvantage which is the topic of the next section.

3.2 The Backslash Plague

As stated earlier, regular expressions use the backslash character (`'\'`) to indicate special forms or to allow special characters to be used without invoking their special meaning. This conflicts with Python's usage of the same character for the same purpose in string literals.

Let's say you want to write a RE that matches the string `\section`, which might be found in a LaTeX file. To figure out what to write in the program code, start with the desired string to be matched. Next, you must escape any backslashes and other metacharacters by preceding them with a backslash, resulting in the string `\\section`. The resulting string that must be passed to `re.compile()` must be `\\section`. However, to express this as a Python string literal, both backslashes must be escaped *again*.

Characters	Stage
`\section`	Text string to be matched
`\\section`	Escaped backslash for `re.compile()`
`"\\\\section"`	Escaped backslashes for a string literal

In short, to match a literal backslash, one has to write `'\\\\'` as the RE string, because the regular expression must be `\\`, and each backslash must be expressed as `\\` inside a regular Python string literal. In REs that feature backslashes repeatedly, this leads to lots of repeated backslashes and makes the resulting strings difficult to understand.

The solution is to use Python's raw string notation for regular expressions; backslashes are not handled in any special way in a string literal prefixed with `'r'`, so `r"\n"` is a two-character string containing `'\'` and `'n'`, while `"\n"` is a one-character string containing a newline. Regular expressions will often be written in Python code using this raw string notation.

Regular String	Raw string
`"ab*"`	`r"ab*"`
`"\\\\section"`	`r"\\section"`
`"\\w+\\s+\\1"`	`r"\w+\s+\1"`

3.3 Performing Matches

Once you have an object representing a compiled regular expression, what do you do with it? Pattern objects have several methods and attributes. Only the most significant ones will be covered here; consult the `re` docs for a complete listing.

Method/Attribute	Purpose
`match()`	Determine if the RE matches at the beginning of the string.
`search()`	Scan through a string, looking for any location where this RE matches.
`findall()`	Find all substrings where the RE matches, and returns them as a list.
`finditer()`	Find all substrings where the RE matches, and returns them as an iterator.

`match()` and `search()` return `None` if no match can be found. If they're successful, a match object instance is returned, containing information about the match: where it starts and ends, the substring it matched, and more.

You can learn about this by interactively experimenting with the `re` module. If you have Tkinter available, you may also want to look at Tools/scripts/redemo.py, a demonstration program included with the Python distribution. It allows you to enter REs and strings, and displays whether the RE matches or fails. `redemo.py` can be quite useful when trying to debug a complicated RE. Phil Schwartz's Kodos is also an interactive tool for developing and testing RE patterns.

This HOWTO uses the standard Python interpreter for its examples. First, run the Python interpreter, import the `re` module, and compile a RE:

```
Python 2.2.2 (#1, Feb 10 2003, 12:57:01)
>>> import re
>>> p = re.compile('[a-z]+')
>>> p  #doctest: +ELLIPSIS
<_sre.SRE_Pattern object at 0x...>
```

Now, you can try matching various strings against the RE `[a-z]+`. An empty string shouldn't match at all, since `+` means 'one or more repetitions'. `match()` should return `None` in this case, which will cause the interpreter to print no output. You can explicitly print the result of `match()` to make this clear.

```
>>> p.match("")
>>> print p.match("")
None
```

Now, let's try it on a string that it should match, such as `tempo`. In this case, `match()` will return a match object, so you should store the result in a variable for later use.

```
>>> m = p.match('tempo')
>>> m
<_sre.SRE_Match object at 0x...>
```

Now you can query the match object for information about the matching string. match object instances also have several methods and attributes; the most important ones are:

Method/Attribute	Purpose
`group()`	Return the string matched by the RE
`start()`	Return the starting position of the match
`end()`	Return the ending position of the match
`span()`	Return a tuple containing the (start, end) positions of the match

Trying these methods will soon clarify their meaning:

```
>>> m.group()
'tempo'
>>> m.start(), m.end()
(0, 5)
>>> m.span()
(0, 5)
```

`group()` returns the substring that was matched by the RE. `start()` and `end()` return the starting and ending index of the match. `span()` returns both start and end indexes in a single tuple. Since the `match()` method only checks if the RE matches at the start of a string, `start()` will always be zero. However, the `search()` method of patterns scans through the string, so the match may not start at zero in that case.

```
>>> print p.match('::: message')
None
>>> m = p.search('::: message'); print m
<_sre.SRE_Match object at 0x...>
>>> m.group()
'message'
>>> m.span()
(4, 11)
```

In actual programs, the most common style is to store the match object in a variable, and then check if it was `None`. This usually looks like:

```
p = re.compile( ... )
m = p.match( 'string goes here' )
if m:
    print 'Match found: ', m.group()
else:
    print 'No match'
```

Two pattern methods return all of the matches for a pattern. `findall()` returns a list of matching strings:

```
>>> p = re.compile('\d+')
>>> p.findall('12 drummers drumming, 11 pipers piping, 10 lords a-leaping')
['12', '11', '10']
```

`findall()` has to create the entire list before it can be returned as the result. The `finditer()` method returns a sequence of match object instances as an iterator.[1]

```
>>> iterator = p.finditer('12 drummers drumming, 11 ... 10 ...')
>>> iterator
<callable-iterator object at 0x...>
>>> for match in iterator:
...     print match.span()
...
(0, 2)
(22, 24)
(29, 31)
```

3.4 Module-Level Functions

You don't have to create a pattern object and call its methods; the `re` module also provides top-level functions called `match()`, `search()`, `findall()`, `sub()`, and so forth. These functions take the same arguments as

[1] Introduced in Python 2.2.2.

the corresponding pattern method, with the RE string added as the first argument, and still return either None or a match object instance.

```
>>> print re.match(r'From\s+', 'Fromage amk')
None
>>> re.match(r'From\s+', 'From amk Thu May 14 19:12:10 1998')
<_sre.SRE_Match object at 0x...>
```

Under the hood, these functions simply create a pattern object for you and call the appropriate method on it. They also store the compiled object in a cache, so future calls using the same RE are faster.

Should you use these module-level functions, or should you get the pattern and call its methods yourself? That choice depends on how frequently the RE will be used, and on your personal coding style. If the RE is being used at only one point in the code, then the module functions are probably more convenient. If a program contains a lot of regular expressions, or re-uses the same ones in several locations, then it might be worthwhile to collect all the definitions in one place, in a section of code that compiles all the REs ahead of time. To take an example from the standard library, here's an extract from the deprecated `xmllib` module:

```
ref = re.compile( ... )
entityref = re.compile( ... )
charref = re.compile( ... )
starttagopen = re.compile( ... )
```

I generally prefer to work with the compiled object, even for one-time uses, but few people will be as much of a purist about this as I am.

3.5 Compilation Flags

Compilation flags let you modify some aspects of how regular expressions work. Flags are available in the `re` module under two names, a long name such as `IGNORECASE` and a short, one-letter form such as `I`. (If you're familiar with Perl's pattern modifiers, the one-letter forms use the same letters; the short form of `re.VERBOSE` is `re.X`, for example.) Multiple flags can be specified by bitwise OR-ing them; `re.I | re.M` sets both the `I` and `M` flags, for example.

Here's a table of the available flags, followed by a more detailed explanation of each one.

Flag	Meaning
`DOTALL`, `S`	Make `.` match any character, including newlines
`IGNORECASE`, `I`	Do case-insensitive matches
`LOCALE`, `L`	Do a locale-aware match
`MULTILINE`, `M`	Multi-line matching, affecting `^` and `$`
`VERBOSE`, `X`	Enable verbose REs, which can be organized more cleanly and understandably.
`UNICODE`, `U`	Makes several escapes like `\w`, `\b`, `\s` and `\d` dependent on the Unicode character database.

`I`
`IGNORECASE`
Perform case-insensitive matching; character class and literal strings will match letters by ignoring case. For example, `[A-Z]` will match lowercase letters, too, and `Spam` will match `Spam`, `spam`, or `spAM`. This lowercasing doesn't take the current locale into account; it will if you also set the `LOCALE` flag.

`L`
`LOCALE`
Make `\w`, `\W`, `\b`, and `\B`, dependent on the current locale.

Locales are a feature of the C library intended to help in writing programs that take account of language differences. For example, if you're processing French text, you'd want to be able to write `\w+` to match words, but `\w` only matches the character class `[A-Za-z]`; it won't match `'é'` or `'ç'`. If your system is configured properly and a French locale is selected, certain C functions will tell the program that `'é'` should also be considered a letter. Setting the `LOCALE` flag when compiling a regular expression will cause the resulting compiled object to use these C functions for `\w`; this is slower, but also enables `\w+` to match French words as you'd expect.

M
MULTILINE

(`^` and `$` haven't been explained yet; they'll be introduced in section *More Metacharacters*.)

Usually `^` matches only at the beginning of the string, and `$` matches only at the end of the string and immediately before the newline (if any) at the end of the string. When this flag is specified, `^` matches at the beginning of the string and at the beginning of each line within the string, immediately following each newline. Similarly, the `$` metacharacter matches either at the end of the string and at the end of each line (immediately preceding each newline).

S
DOTALL

Makes the `'.'` special character match any character at all, including a newline; without this flag, `'.'` will match anything *except* a newline.

U
UNICODE

Make `\w`, `\W`, `\b`, `\B`, `\d`, `\D`, `\s` and `\S` dependent on the Unicode character properties database.

X
VERBOSE

This flag allows you to write regular expressions that are more readable by granting you more flexibility in how you can format them. When this flag has been specified, whitespace within the RE string is ignored, except when the whitespace is in a character class or preceded by an unescaped backslash; this lets you organize and indent the RE more clearly. This flag also lets you put comments within a RE that will be ignored by the engine; comments are marked by a `'#'` that's neither in a character class or preceded by an unescaped backslash.

For example, here's a RE that uses `re.VERBOSE`; see how much easier it is to read?

```
charref = re.compile(r"""
 &[#]                # Start of a numeric entity reference
 (
     0[0-7]+         # Octal form
   | [0-9]+          # Decimal form
   | x[0-9a-fA-F]+   # Hexadecimal form
 )
 ;                   # Trailing semicolon
""", re.VERBOSE)
```

Without the verbose setting, the RE would look like this:

```
charref = re.compile("&#(0[0-7]+"
                     "|[0-9]+"
                     "|x[0-9a-fA-F]+);")
```

In the above example, Python's automatic concatenation of string literals has been used to break up the RE into smaller pieces, but it's still more difficult to understand than the version using `re.VERBOSE`.

4 More Pattern Power

So far we've only covered a part of the features of regular expressions. In this section, we'll cover some new metacharacters, and how to use groups to retrieve portions of the text that was matched.

4.1 More Metacharacters

There are some metacharacters that we haven't covered yet. Most of them will be covered in this section.

Some of the remaining metacharacters to be discussed are *zero-width assertions*. They don't cause the engine to advance through the string; instead, they consume no characters at all, and simply succeed or fail. For example, `\b` is an assertion that the current position is located at a word boundary; the position isn't changed by the `\b` at all. This means that zero-width assertions should never be repeated, because if they match once at a given location, they can obviously be matched an infinite number of times.

`|` Alternation, or the "or" operator. If A and B are regular expressions, `A|B` will match any string that matches either `A` or `B`. `|` has very low precedence in order to make it work reasonably when you're alternating multi-character strings. `Crow|Servo` will match either `Crow` or `Servo`, not `Cro`, a `'w'` or an `'S'`, and `ervo`.

To match a literal `'|'`, use `\|`, or enclose it inside a character class, as in `[|]`.

`^` Matches at the beginning of lines. Unless the `MULTILINE` flag has been set, this will only match at the beginning of the string. In `MULTILINE` mode, this also matches immediately after each newline within the string.

For example, if you wish to match the word `From` only at the beginning of a line, the RE to use is `^From`.

```
>>> print re.search('^From', 'From Here to Eternity')
<_sre.SRE_Match object at 0x...>
>>> print re.search('^From', 'Reciting From Memory')
None
```

`$` Matches at the end of a line, which is defined as either the end of the string, or any location followed by a newline character.

```
>>> print re.search('}$', '{block}')
<_sre.SRE_Match object at 0x...>
>>> print re.search('}$', '{block} ')
None
>>> print re.search('}$', '{block}\n')
<_sre.SRE_Match object at 0x...>
```

To match a literal `'$'`, use `\$` or enclose it inside a character class, as in `[$]`.

`\A` Matches only at the start of the string. When not in `MULTILINE` mode, `\A` and `^` are effectively the same. In `MULTILINE` mode, they're different: `\A` still matches only at the beginning of the string, but `^` may match at any location inside the string that follows a newline character.

`\Z` Matches only at the end of the string.

`\b` Word boundary. This is a zero-width assertion that matches only at the beginning or end of a word. A word is defined as a sequence of alphanumeric characters, so the end of a word is indicated by whitespace or a non-alphanumeric character.

The following example matches `class` only when it's a complete word; it won't match when it's contained inside another word.

```
>>> p = re.compile(r'\bclass\b')
>>> print p.search('no class at all')
<_sre.SRE_Match object at 0x...>
>>> print p.search('the declassified algorithm')
None
>>> print p.search('one subclass is')
None
```

There are two subtleties you should remember when using this special sequence. First, this is the worst collision between Python's string literals and regular expression sequences. In Python's string literals, `\b` is the backspace character, ASCII value 8. If you're not using raw strings, then Python will convert the `\b` to a backspace, and your RE won't match as you expect it to. The following example looks the same as our previous RE, but omits the `'r'` in front of the RE string.

```
>>> p = re.compile('\bclass\b')
>>> print p.search('no class at all')
None
>>> print p.search('\b' + 'class' + '\b')
<_sre.SRE_Match object at 0x...>
```

Second, inside a character class, where there's no use for this assertion, `\b` represents the backspace character, for compatibility with Python's string literals.

`\B` Another zero-width assertion, this is the opposite of `\b`, only matching when the current position is not at a word boundary.

4.2 Grouping

Frequently you need to obtain more information than just whether the RE matched or not. Regular expressions are often used to dissect strings by writing a RE divided into several subgroups which match different components of interest. For example, an RFC-822 header line is divided into a header name and a value, separated by a `':'`, like this:

```
From: author@example.com
User-Agent: Thunderbird 1.5.0.9 (X11/20061227)
MIME-Version: 1.0
To: editor@example.com
```

This can be handled by writing a regular expression which matches an entire header line, and has one group which matches the header name, and another group which matches the header's value.

Groups are marked by the `'('`, `')'` metacharacters. `'('` and `')'` have much the same meaning as they do in mathematical expressions; they group together the expressions contained inside them, and you can repeat the contents of a group with a repeating qualifier, such as `*`, `+`, `?`, or `{m,n}`. For example, `(ab)*` will match zero or more repetitions of `ab`.

```
>>> p = re.compile('(ab)*')
>>> print p.match('abababababab').span()
(0, 10)
```

Groups indicated with `'('`, `')'` also capture the starting and ending index of the text that they match; this can be retrieved by passing an argument to `group()`, `start()`, `end()`, and `span()`. Groups are numbered starting with 0. Group 0 is always present; it's the whole RE, so match object methods all have group 0 as their default argument. Later we'll see how to express groups that don't capture the span of text that they match.

```
>>> p = re.compile('(a)b')
>>> m = p.match('ab')
>>> m.group()
'ab'
>>> m.group(0)
'ab'
```

Subgroups are numbered from left to right, from 1 upward. Groups can be nested; to determine the number, just count the opening parenthesis characters, going from left to right.

```
>>> p = re.compile('(a(b)c)d')
>>> m = p.match('abcd')
>>> m.group(0)
'abcd'
>>> m.group(1)
'abc'
>>> m.group(2)
'b'
```

`group()` can be passed multiple group numbers at a time, in which case it will return a tuple containing the corresponding values for those groups.

```
>>> m.group(2,1,2)
('b', 'abc', 'b')
```

The `groups()` method returns a tuple containing the strings for all the subgroups, from 1 up to however many there are.

```
>>> m.groups()
('abc', 'b')
```

Backreferences in a pattern allow you to specify that the contents of an earlier capturing group must also be found at the current location in the string. For example, `\1` will succeed if the exact contents of group 1 can be found at the current position, and fails otherwise. Remember that Python's string literals also use a backslash followed by numbers to allow including arbitrary characters in a string, so be sure to use a raw string when incorporating backreferences in a RE.

For example, the following RE detects doubled words in a string.

```
>>> p = re.compile(r'\b(\w+)\s+\1\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

Backreferences like this aren't often useful for just searching through a string — there are few text formats which repeat data in this way — but you'll soon find out that they're *very* useful when performing string substitutions.

4.3 Non-capturing and Named Groups

Elaborate REs may use many groups, both to capture substrings of interest, and to group and structure the RE itself. In complex REs, it becomes difficult to keep track of the group numbers. There are two features which help with this problem. Both of them use a common syntax for regular expression extensions, so we'll look at that first.

Perl 5 added several additional features to standard regular expressions, and the Python `re` module supports most of them. It would have been difficult to choose new single-keystroke metacharacters or new special

sequences beginning with `\` to represent the new features without making Perl's regular expressions confusingly different from standard REs. If you chose `&` as a new metacharacter, for example, old expressions would be assuming that `&` was a regular character and wouldn't have escaped it by writing `\&` or `[&]`.

The solution chosen by the Perl developers was to use `(?...)` as the extension syntax. `?` immediately after a parenthesis was a syntax error because the `?` would have nothing to repeat, so this didn't introduce any compatibility problems. The characters immediately after the `?` indicate what extension is being used, so `(?=foo)` is one thing (a positive lookahead assertion) and `(?:foo)` is something else (a non-capturing group containing the subexpression `foo`).

Python adds an extension syntax to Perl's extension syntax. If the first character after the question mark is a `P`, you know that it's an extension that's specific to Python. Currently there are two such extensions: `(?P<name>...)` defines a named group, and `(?P=name)` is a backreference to a named group. If future versions of Perl 5 add similar features using a different syntax, the `re` module will be changed to support the new syntax, while preserving the Python-specific syntax for compatibility's sake.

Now that we've looked at the general extension syntax, we can return to the features that simplify working with groups in complex REs. Since groups are numbered from left to right and a complex expression may use many groups, it can become difficult to keep track of the correct numbering. Modifying such a complex RE is annoying, too: insert a new group near the beginning and you change the numbers of everything that follows it.

Sometimes you'll want to use a group to collect a part of a regular expression, but aren't interested in retrieving the group's contents. You can make this fact explicit by using a non-capturing group: `(?:...)`, where you can replace the `...` with any other regular expression.

```
>>> m = re.match("([abc])+", "abc")
>>> m.groups()
('c',)
>>> m = re.match("(?:[abc])+", "abc")
>>> m.groups()
()
```

Except for the fact that you can't retrieve the contents of what the group matched, a non-capturing group behaves exactly the same as a capturing group; you can put anything inside it, repeat it with a repetition metacharacter such as `*`, and nest it within other groups (capturing or non-capturing). `(?:...)` is particularly useful when modifying an existing pattern, since you can add new groups without changing how all the other groups are numbered. It should be mentioned that there's no performance difference in searching between capturing and non-capturing groups; neither form is any faster than the other.

A more significant feature is named groups: instead of referring to them by numbers, groups can be referenced by a name.

The syntax for a named group is one of the Python-specific extensions: `(?P<name>...)`. *name* is, obviously, the name of the group. Named groups also behave exactly like capturing groups, and additionally associate a name with a group. The match object methods that deal with capturing groups all accept either integers that refer to the group by number or strings that contain the desired group's name. Named groups are still given numbers, so you can retrieve information about a group in two ways:

```
>>> p = re.compile(r'(?P<word>\b\w+\b)')
>>> m = p.search( '(((( Lots of punctuation )))' )
>>> m.group('word')
'Lots'
>>> m.group(1)
'Lots'
```

Named groups are handy because they let you use easily-remembered names, instead of having to remember numbers. Here's an example RE from the `imaplib` module:

```
InternalDate = re.compile(r'INTERNALDATE "'
        r'(?P<day>[ 123][0-9])-(?P<mon>[A-Z][a-z][a-z])-'
        r'(?P<year>[0-9][0-9][0-9][0-9])'
        r' (?P<hour>[0-9][0-9]):(?P<min>[0-9][0-9]):(?P<sec>[0-9][0-9])'
        r' (?P<zonen>[-+])(?P<zoneh>[0-9][0-9])(?P<zonem>[0-9][0-9])'
        r'"')
```

It's obviously much easier to retrieve `m.group('zonem')`, instead of having to remember to retrieve group 9.

The syntax for backreferences in an expression such as `(...)\1` refers to the number of the group. There's naturally a variant that uses the group name instead of the number. This is another Python extension: `(?P=name)` indicates that the contents of the group called *name* should again be matched at the current point. The regular expression for finding doubled words, `\b(\w+)\s+\1\b` can also be written as `\b(?P<word>\w+)\s+(?P=word)\b`:

```
>>> p = re.compile(r'\b(?P<word>\w+)\s+(?P=word)\b')
>>> p.search('Paris in the the spring').group()
'the the'
```

4.4 Lookahead Assertions

Another zero-width assertion is the lookahead assertion. Lookahead assertions are available in both positive and negative form, and look like this:

`(?=...)` Positive lookahead assertion. This succeeds if the contained regular expression, represented here by `...`, successfully matches at the current location, and fails otherwise. But, once the contained expression has been tried, the matching engine doesn't advance at all; the rest of the pattern is tried right where the assertion started.

`(?!...)` Negative lookahead assertion. This is the opposite of the positive assertion; it succeeds if the contained expression *doesn't* match at the current position in the string.

To make this concrete, let's look at a case where a lookahead is useful. Consider a simple pattern to match a filename and split it apart into a base name and an extension, separated by a `.`. For example, in `news.rc`, `news` is the base name, and `rc` is the filename's extension.

The pattern to match this is quite simple:

`.*[.].*$`

Notice that the `.` needs to be treated specially because it's a metacharacter; I've put it inside a character class. Also notice the trailing `$`; this is added to ensure that all the rest of the string must be included in the extension. This regular expression matches `foo.bar` and `autoexec.bat` and `sendmail.cf` and `printers.conf`.

Now, consider complicating the problem a bit; what if you want to match filenames where the extension is not `bat`? Some incorrect attempts:

`.*[.][^b].*$` The first attempt above tries to exclude `bat` by requiring that the first character of the extension is not a `b`. This is wrong, because the pattern also doesn't match `foo.bar`.

`.*[.]([^b]..|.[^a].|..[^t])$`

The expression gets messier when you try to patch up the first solution by requiring one of the following cases to match: the first character of the extension isn't `b`; the second character isn't `a`; or the third character isn't `t`. This accepts `foo.bar` and rejects `autoexec.bat`, but it requires a three-letter extension and won't accept a filename with a two-letter extension such as `sendmail.cf`. We'll complicate the pattern again in an effort to fix it.

`.*[.]([^b].?.?|.[^a]?.?|..?[^t]?)$`

In the third attempt, the second and third letters are all made optional in order to allow matching extensions shorter than three characters, such as `sendmail.cf`.

The pattern's getting really complicated now, which makes it hard to read and understand. Worse, if the problem changes and you want to exclude both `bat` and `exe` as extensions, the pattern would get even more complicated and confusing.

A negative lookahead cuts through all this confusion:

`.*[.](?!bat$)[^.]*$` The negative lookahead means: if the expression `bat` doesn't match at this point, try the rest of the pattern; if `bat$` does match, the whole pattern will fail. The trailing `$` is required to ensure that something like `sample.batch`, where the extension only starts with `bat`, will be allowed. The `[^.]*` makes sure that the pattern works when there are multiple dots in the filename.

Excluding another filename extension is now easy; simply add it as an alternative inside the assertion. The following pattern excludes filenames that end in either `bat` or `exe`:

`.*[.](?!bat$|exe$)[^.]*$`

5 Modifying Strings

Up to this point, we've simply performed searches against a static string. Regular expressions are also commonly used to modify strings in various ways, using the following pattern methods:

Method/Attribute	Purpose
`split()`	Split the string into a list, splitting it wherever the RE matches
`sub()`	Find all substrings where the RE matches, and replace them with a different string
`subn()`	Does the same thing as `sub()`, but returns the new string and the number of replacements

5.1 Splitting Strings

The `split()` method of a pattern splits a string apart wherever the RE matches, returning a list of the pieces. It's similar to the `split()` method of strings but provides much more generality in the delimiters that you can split by; `split()` only supports splitting by whitespace or by a fixed string. As you'd expect, there's a module-level `re.split()` function, too.

`.split(`*string*[, *maxsplit=0*]`)`
: Split *string* by the matches of the regular expression. If capturing parentheses are used in the RE, then their contents will also be returned as part of the resulting list. If *maxsplit* is nonzero, at most *maxsplit* splits are performed.

You can limit the number of splits made, by passing a value for *maxsplit*. When *maxsplit* is nonzero, at most *maxsplit* splits will be made, and the remainder of the string is returned as the final element of the list. In the following example, the delimiter is any sequence of non-alphanumeric characters.

```
>>> p = re.compile(r'\W+')
>>> p.split('This is a test, short and sweet, of split().')
['This', 'is', 'a', 'test', 'short', 'and', 'sweet', 'of', 'split', '']
>>> p.split('This is a test, short and sweet, of split().', 3)
['This', 'is', 'a', 'test, short and sweet, of split().']
```

Sometimes you're not only interested in what the text between delimiters is, but also need to know what the delimiter was. If capturing parentheses are used in the RE, then their values are also returned as part of the list. Compare the following calls:

```
>>> p = re.compile(r'\W+')
>>> p2 = re.compile(r'(\W+)')
>>> p.split('This... is a test.')
['This', 'is', 'a', 'test', '']
>>> p2.split('This... is a test.')
['This', '... ', 'is', ' ', 'a', ' ', 'test', '.', '']
```

The module-level function `re.split()` adds the RE to be used as the first argument, but is otherwise the same.

```
>>> re.split('[\W]+', 'Words, words, words.')
['Words', 'words', 'words', '']
>>> re.split('([\W]+)', 'Words, words, words.')
['Words', ', ', 'words', ', ', 'words', '.', '']
>>> re.split('[\W]+', 'Words, words, words.', 1)
['Words', 'words, words.']
```

5.2 Search and Replace

Another common task is to find all the matches for a pattern, and replace them with a different string. The `sub()` method takes a replacement value, which can be either a string or a function, and the string to be processed.

`.sub(`*replacement*, *string*[, *count=0*]`)`

Returns the string obtained by replacing the leftmost non-overlapping occurrences of the RE in *string* by the replacement *replacement*. If the pattern isn't found, *string* is returned unchanged.

The optional argument *count* is the maximum number of pattern occurrences to be replaced; *count* must be a non-negative integer. The default value of 0 means to replace all occurrences.

Here's a simple example of using the `sub()` method. It replaces colour names with the word `colour`:

```
>>> p = re.compile('(blue|white|red)')
>>> p.sub('colour', 'blue socks and red shoes')
'colour socks and colour shoes'
>>> p.sub('colour', 'blue socks and red shoes', count=1)
'colour socks and red shoes'
```

The `subn()` method does the same work, but returns a 2-tuple containing the new string value and the number of replacements that were performed:

```
>>> p = re.compile('(blue|white|red)')
>>> p.subn('colour', 'blue socks and red shoes')
('colour socks and colour shoes', 2)
>>> p.subn('colour', 'no colours at all')
('no colours at all', 0)
```

Empty matches are replaced only when they're not adjacent to a previous match.

```
>>> p = re.compile('x*')
>>> p.sub('-', 'abxd')
'-a-b-d-'
```

If *replacement* is a string, any backslash escapes in it are processed. That is, `\n` is converted to a single newline character, `\r` is converted to a carriage return, and so forth. Unknown escapes such as `\j` are left alone. Backreferences, such as `\6`, are replaced with the substring matched by the corresponding group in the RE. This lets you incorporate portions of the original text in the resulting replacement string.

This example matches the word `section` followed by a string enclosed in `{`, `}`, and changes `section` to `subsection`:

```
>>> p = re.compile('section{ ( [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First} section{second}')
'subsection{First} subsection{second}'
```

There's also a syntax for referring to named groups as defined by the `(?P<name>...)` syntax. `\g<name>` will use the substring matched by the group named `name`, and `\g<number>` uses the corresponding group number. `\g<2>` is therefore equivalent to `\2`, but isn't ambiguous in a replacement string such as `\g<2>0`. (`\20` would be interpreted as a reference to group 20, not a reference to group 2 followed by the literal character `'0'`.) The following substitutions are all equivalent, but use all three variations of the replacement string.

```
>>> p = re.compile('section{ (?P<name> [^}]* ) }', re.VERBOSE)
>>> p.sub(r'subsection{\1}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<1>}','section{First}')
'subsection{First}'
>>> p.sub(r'subsection{\g<name>}','section{First}')
'subsection{First}'
```

replacement can also be a function, which gives you even more control. If *replacement* is a function, the function is called for every non-overlapping occurrence of *pattern*. On each call, the function is passed a match object argument for the match and can use this information to compute the desired replacement string and return it.

In the following example, the replacement function translates decimals into hexadecimal:

```
>>> def hexrepl(match):
...     "Return the hex string for a decimal number"
...     value = int(match.group())
...     return hex(value)
...
>>> p = re.compile(r'\d+')
>>> p.sub(hexrepl, 'Call 65490 for printing, 49152 for user code.')
'Call 0xffd2 for printing, 0xc000 for user code.'
```

When using the module-level `re.sub()` function, the pattern is passed as the first argument. The pattern may be provided as an object or as a string; if you need to specify regular expression flags, you must either use a pattern object as the first parameter, or use embedded modifiers in the pattern string, e.g. `sub("(?i)b+", "x", "bbbb BBBB")` returns `'x x'`.

6 Common Problems

Regular expressions are a powerful tool for some applications, but in some ways their behaviour isn't intuitive and at times they don't behave the way you may expect them to. This section will point out some of the most common pitfalls.

6.1 Use String Methods

Sometimes using the `re` module is a mistake. If you're matching a fixed string, or a single character class, and you're not using any `re` features such as the `IGNORECASE` flag, then the full power of regular expressions may not be required. Strings have several methods for performing operations with fixed strings and they're usually much faster, because the implementation is a single small C loop that's been optimized for the purpose, instead of the large, more generalized regular expression engine.

One example might be replacing a single fixed string with another one; for example, you might replace `word` with `deed`. `re.sub()` seems like the function to use for this, but consider the `replace()` method. Note that `replace()` will also replace `word` inside words, turning `swordfish` into `sdeedfish`, but the naive RE `word` would have done that, too. (To avoid performing the substitution on parts of words, the pattern would have to be `\bword\b`, in order to require that `word` have a word boundary on either side. This takes the job beyond `replace()`'s abilities.)

Another common task is deleting every occurrence of a single character from a string or replacing it with another single character. You might do this with something like `re.sub('\n', ' ', S)`, but `translate()` is capable of doing both tasks and will be faster than any regular expression operation can be.

In short, before turning to the `re` module, consider whether your problem can be solved with a faster and simpler string method.

6.2 match() versus search()

The `match()` function only checks if the RE matches at the beginning of the string while `search()` will scan forward through the string for a match. It's important to keep this distinction in mind. Remember, `match()` will only report a successful match which will start at 0; if the match wouldn't start at zero, `match()` will *not* report it.

```
>>> print re.match('super', 'superstition').span()
(0, 5)
>>> print re.match('super', 'insuperable')
None
```

On the other hand, `search()` will scan forward through the string, reporting the first match it finds.

```
>>> print re.search('super', 'superstition').span()
(0, 5)
>>> print re.search('super', 'insuperable').span()
(2, 7)
```

Sometimes you'll be tempted to keep using `re.match()`, and just add `.*` to the front of your RE. Resist this temptation and use `re.search()` instead. The regular expression compiler does some analysis of REs in order to speed up the process of looking for a match. One such analysis figures out what the first character of a match must be; for example, a pattern starting with `Crow` must match starting with a `'C'`. The analysis lets the engine quickly scan through the string looking for the starting character, only trying the full match if a `'C'` is found.

Adding `.*` defeats this optimization, requiring scanning to the end of the string and then backtracking to find a match for the rest of the RE. Use `re.search()` instead.

6.3 Greedy versus Non-Greedy

When repeating a regular expression, as in `a*`, the resulting action is to consume as much of the pattern as possible. This fact often bites you when you're trying to match a pair of balanced delimiters, such as the

angle brackets surrounding an HTML tag. The naive pattern for matching a single HTML tag doesn't work because of the greedy nature of `.*`.

```
>>> s = '<html><head><title>Title</title>'
>>> len(s)
32
>>> print re.match('<.*>', s).span()
(0, 32)
>>> print re.match('<.*>', s).group()
<html><head><title>Title</title>
```

The RE matches the `'<'` in `<html>`, and the `.*` consumes the rest of the string. There's still more left in the RE, though, and the `>` can't match at the end of the string, so the regular expression engine has to backtrack character by character until it finds a match for the `>`. The final match extends from the `'<'` in `<html>` to the `'>'` in `</title>`, which isn't what you want.

In this case, the solution is to use the non-greedy qualifiers `*?`, `+?`, `??`, or `{m,n}?`, which match as *little* text as possible. In the above example, the `'>'` is tried immediately after the first `'<'` matches, and when it fails, the engine advances a character at a time, retrying the `'>'` at every step. This produces just the right result:

```
>>> print re.match('<.*?>', s).group()
<html>
```

(Note that parsing HTML or XML with regular expressions is painful. Quick-and-dirty patterns will handle common cases, but HTML and XML have special cases that will break the obvious regular expression; by the time you've written a regular expression that handles all of the possible cases, the patterns will be *very* complicated. Use an HTML or XML parser module for such tasks.)

6.4 Using re.VERBOSE

By now you've probably noticed that regular expressions are a very compact notation, but they're not terribly readable. REs of moderate complexity can become lengthy collections of backslashes, parentheses, and metacharacters, making them difficult to read and understand.

For such REs, specifying the `re.VERBOSE` flag when compiling the regular expression can be helpful, because it allows you to format the regular expression more clearly.

The `re.VERBOSE` flag has several effects. Whitespace in the regular expression that *isn't* inside a character class is ignored. This means that an expression such as `dog | cat` is equivalent to the less readable `dog|cat`, but `[a b]` will still match the characters `'a'`, `'b'`, or a space. In addition, you can also put comments inside a RE; comments extend from a `#` character to the next newline. When used with triple-quoted strings, this enables REs to be formatted more neatly:

```
pat = re.compile(r"""
 \s*                 # Skip leading whitespace
 (?P<header>[^:]+)   # Header name
 \s* :               # Whitespace, and a colon
 (?P<value>.*?)      # The header's value -- *? used to
                     # lose the following trailing whitespace
 \s*$                # Trailing whitespace to end-of-line
""", re.VERBOSE)
```

This is far more readable than:

```
pat = re.compile(r"\s*(?P<header>[^:]+)\s*:(?P<value>.*?)\s*$")
```

7 Feedback

Regular expressions are a complicated topic. Did this document help you understand them? Were there parts that were unclear, or Problems you encountered that weren't covered here? If so, please send suggestions for improvements to the author.

The most complete book on regular expressions is almost certainly Jeffrey Friedl's Mastering Regular Expressions, published by O'Reilly. Unfortunately, it exclusively concentrates on Perl and Java's flavours of regular expressions, and doesn't contain any Python material at all, so it won't be useful as a reference for programming in Python. (The first edition covered Python's now-removed `regex` module, which won't help you much.) Consider checking it out from your library.

Socket Programming HOWTO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Gordon McMillan

Abstract

Sockets are used nearly everywhere, but are one of the most severely misunderstood technologies around. This is a 10,000 foot overview of sockets. It's not really a tutorial - you'll still have work to do in getting things operational. It doesn't cover the fine points (and there are a lot of them), but I hope it will give you enough background to begin using them decently.

1 Sockets

I'm only going to talk about INET sockets, but they account for at least 99% of the sockets in use. And I'll only talk about STREAM sockets - unless you really know what you're doing (in which case this HOWTO

isn't for you!), you'll get better behavior and performance from a STREAM socket than anything else. I will try to clear up the mystery of what a socket is, as well as some hints on how to work with blocking and non-blocking sockets. But I'll start by talking about blocking sockets. You'll need to know how they work before dealing with non-blocking sockets.

Part of the trouble with understanding these things is that "socket" can mean a number of subtly different things, depending on context. So first, let's make a distinction between a "client" socket - an endpoint of a conversation, and a "server" socket, which is more like a switchboard operator. The client application (your browser, for example) uses "client" sockets exclusively; the web server it's talking to uses both "server" sockets and "client" sockets.

1.1 History

Of the various forms of IPC (Inter Process Communication), sockets are by far the most popular. On any given platform, there are likely to be other forms of IPC that are faster, but for cross-platform communication, sockets are about the only game in town.

They were invented in Berkeley as part of the BSD flavor of Unix. They spread like wildfire with the Internet. With good reason — the combination of sockets with INET makes talking to arbitrary machines around the world unbelievably easy (at least compared to other schemes).

2 Creating a Socket

Roughly speaking, when you clicked on the link that brought you to this page, your browser did something like the following:

```
#create an INET, STREAMing socket
s = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#now connect to the web server on port 80
# - the normal http port
s.connect(("www.mcmillan-inc.com", 80))
```

When the `connect` completes, the socket `s` can be used to send in a request for the text of the page. The same socket will read the reply, and then be destroyed. That's right, destroyed. Client sockets are normally only used for one exchange (or a small set of sequential exchanges).

What happens in the web server is a bit more complex. First, the web server creates a "server socket":

```
#create an INET, STREAMing socket
serversocket = socket.socket(
    socket.AF_INET, socket.SOCK_STREAM)
#bind the socket to a public host,
# and a well-known port
serversocket.bind((socket.gethostname(), 80))
#become a server socket
serversocket.listen(5)
```

A couple things to notice: we used `socket.gethostname()` so that the socket would be visible to the outside world. If we had used `s.bind(('localhost', 80))` or `s.bind(('127.0.0.1', 80))` we would still have a "server" socket, but one that was only visible within the same machine. `s.bind(('', 80))` specifies that the socket is reachable by any address the machine happens to have.

A second thing to note: low number ports are usually reserved for "well known" services (HTTP, SNMP etc). If you're playing around, use a nice high number (4 digits).

Finally, the argument to `listen` tells the socket library that we want it to queue up as many as 5 connect requests (the normal max) before refusing outside connections. If the rest of the code is written properly, that should be plenty.

Now that we have a "server" socket, listening on port 80, we can enter the mainloop of the web server:

```
while 1:
    #accept connections from outside
    (clientsocket, address) = serversocket.accept()
    #now do something with the clientsocket
    #in this case, we'll pretend this is a threaded server
    ct = client_thread(clientsocket)
    ct.run()
```

There's actually 3 general ways in which this loop could work - dispatching a thread to handle `clientsocket`, create a new process to handle `clientsocket`, or restructure this app to use non-blocking sockets, and multiplex between our "server" socket and any active `clientsocket`s using `select`. More about that later. The important thing to understand now is this: this is *all* a "server" socket does. It doesn't send any data. It doesn't receive any data. It just produces "client" sockets. Each `clientsocket` is created in response to some *other* "client" socket doing a `connect()` to the host and port we're bound to. As soon as we've created that `clientsocket`, we go back to listening for more connections. The two "clients" are free to chat it up - they are using some dynamically allocated port which will be recycled when the conversation ends.

2.1 IPC

If you need fast IPC between two processes on one machine, you should look into whatever form of shared memory the platform offers. A simple protocol based around shared memory and locks or semaphores is by far the fastest technique.

If you do decide to use sockets, bind the "server" socket to `'localhost'`. On most platforms, this will take a shortcut around a couple of layers of network code and be quite a bit faster.

3 Using a Socket

The first thing to note, is that the web browser's "client" socket and the web server's "client" socket are identical beasts. That is, this is a "peer to peer" conversation. Or to put it another way, *as the designer, you will have to decide what the rules of etiquette are for a conversation*. Normally, the `connect`ing socket starts the conversation, by sending in a request, or perhaps a signon. But that's a design decision - it's not a rule of sockets.

Now there are two sets of verbs to use for communication. You can use `send` and `recv`, or you can transform your client socket into a file-like beast and use `read` and `write`. The latter is the way Java presents its sockets. I'm not going to talk about it here, except to warn you that you need to use `flush` on sockets. These are buffered "files", and a common mistake is to `write` something, and then `read` for a reply. Without a `flush` in there, you may wait forever for the reply, because the request may still be in your output buffer.

Now we come to the major stumbling block of sockets - `send` and `recv` operate on the network buffers. They do not necessarily handle all the bytes you hand them (or expect from them), because their major focus is handling the network buffers. In general, they return when the associated network buffers have been filled (`send`) or emptied (`recv`). They then tell you how many bytes they handled. It is *your* responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed (or is in the process of closing) the connection. You will not receive any more data on this connection. Ever. You may be able to send data successfully; I'll talk more about this later.

A protocol like HTTP uses a socket for only one transfer. The client sends a request, then reads a reply. That's it. The socket is discarded. This means that a client can detect the end of the reply by receiving 0 bytes.

But if you plan to reuse your socket for further transfers, you need to realize that *there is no* EOT (End of Transfer) *on a socket.* I repeat: if a socket `send` or `recv` returns after handling 0 bytes, the connection has been broken. If the connection has *not* been broken, you may wait on a `recv` forever, because the socket will *not* tell you that there's nothing more to read (for now). Now if you think about that a bit, you'll come to realize a fundamental truth of sockets: *messages must either be fixed length* (yuck), *or be delimited* (shrug), *or indicate how long they are* (much better), *or end by shutting down the connection.* The choice is entirely yours, (but some ways are righter than others).

Assuming you don't want to end the connection, the simplest solution is a fixed length message:

```
class mysocket:
    '''demonstration class only
      - coded for clarity, not efficiency
    '''

    def __init__(self, sock=None):
        if sock is None:
            self.sock = socket.socket(
                socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock

    def connect(self, host, port):
        self.sock.connect((host, port))

    def mysend(self, msg):
        totalsent = 0
        while totalsent < MSGLEN:
            sent = self.sock.send(msg[totalsent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            totalsent = totalsent + sent

    def myreceive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < MSGLEN:
            chunk = self.sock.recv(min(MSGLEN - bytes_recd, 2048))
            if chunk == '':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd = bytes_recd + len(chunk)
        return ''.join(chunks)
```

The sending code here is usable for almost any messaging scheme - in Python you send strings, and you can use `len()` to determine its length (even if it has embedded `\0` characters). It's mostly the receiving code that gets more complex. (And in C, it's not much worse, except you can't use `strlen` if the message has embedded `\0`s.)

The easiest enhancement is to make the first character of the message an indicator of message type, and have the type determine the length. Now you have two `recv`s - the first to get (at least) that first character so you can look up the length, and the second in a loop to get the rest. If you decide to go the delimited route, you'll be receiving in some arbitrary chunk size, (4096 or 8192 is frequently a good match for network buffer sizes), and scanning what you've received for a delimiter.

One complication to be aware of: if your conversational protocol allows multiple messages to be sent back to back (without some kind of reply), and you pass `recv` an arbitrary chunk size, you may end up reading the start of a following message. You'll need to put that aside and hold onto it, until it's needed.

Prefixing the message with its length (say, as 5 numeric characters) gets more complex, because (believe it or not), you may not get all 5 characters in one `recv`. In playing around, you'll get away with it; but in high network loads, your code will very quickly break unless you use two `recv` loops - the first to determine the length, the second to get the data part of the message. Nasty. This is also when you'll discover that `send` does not always manage to get rid of everything in one pass. And despite having read this, you will eventually get bit by it!

In the interests of space, building your character, (and preserving my competitive position), these enhancements are left as an exercise for the reader. Lets move on to cleaning up.

3.1 Binary Data

It is perfectly possible to send binary data over a socket. The major problem is that not all machines use the same formats for binary data. For example, a Motorola chip will represent a 16 bit integer with the value 1 as the two hex bytes 00 01. Intel and DEC, however, are byte-reversed - that same 1 is 01 00. Socket libraries have calls for converting 16 and 32 bit integers - `ntohl, htonl, ntohs, htons` where "n" means *network* and "h" means *host*, "s" means *short* and "l" means *long*. Where network order is host order, these do nothing, but where the machine is byte-reversed, these swap the bytes around appropriately.

In these days of 32 bit machines, the ascii representation of binary data is frequently smaller than the binary representation. That's because a surprising amount of the time, all those longs have the value 0, or maybe 1. The string "0" would be two bytes, while binary is four. Of course, this doesn't fit well with fixed-length messages. Decisions, decisions.

4 Disconnecting

Strictly speaking, you're supposed to use `shutdown` on a socket before you `close` it. The `shutdown` is an advisory to the socket at the other end. Depending on the argument you pass it, it can mean "I'm not going to send anymore, but I'll still listen", or "I'm not listening, good riddance!". Most socket libraries, however, are so used to programmers neglecting to use this piece of etiquette that normally a `close` is the same as `shutdown(); close()`. So in most situations, an explicit `shutdown` is not needed.

One way to use `shutdown` effectively is in an HTTP-like exchange. The client sends a request and then does a `shutdown(1)`. This tells the server "This client is done sending, but can still receive." The server can detect "EOF" by a receive of 0 bytes. It can assume it has the complete request. The server sends a reply. If the `send` completes successfully then, indeed, the client was still receiving.

Python takes the automatic shutdown a step further, and says that when a socket is garbage collected, it will automatically do a `close` if it's needed. But relying on this is a very bad habit. If your socket just disappears without doing a `close`, the socket at the other end may hang indefinitely, thinking you're just being slow. *Please* `close` your sockets when you're done.

4.1 When Sockets Die

Probably the worst thing about using blocking sockets is what happens when the other side comes down hard (without doing a `close`). Your socket is likely to hang. SOCKSTREAM is a reliable protocol, and it will wait a long, long time before giving up on a connection. If you're using threads, the entire thread is essentially dead. There's not much you can do about it. As long as you aren't doing something dumb, like holding a lock while doing a blocking read, the thread isn't really consuming much in the way of resources. Do *not* try to kill the thread - part of the reason that threads are more efficient than processes is that they

avoid the overhead associated with the automatic recycling of resources. In other words, if you do manage to kill the thread, your whole process is likely to be screwed up.

5 Non-blocking Sockets

If you've understood the preceding, you already know most of what you need to know about the mechanics of using sockets. You'll still use the same calls, in much the same ways. It's just that, if you do it right, your app will be almost inside-out.

In Python, you use `socket.setblocking(0)` to make it non-blocking. In C, it's more complex, (for one thing, you'll need to choose between the BSD flavor `O_NONBLOCK` and the almost indistinguishable Posix flavor `O_NDELAY`, which is completely different from `TCP_NODELAY`), but it's the exact same idea. You do this after creating the socket, but before using it. (Actually, if you're nuts, you can switch back and forth.)

The major mechanical difference is that `send`, `recv`, `connect` and `accept` can return without having done anything. You have (of course) a number of choices. You can check return code and error codes and generally drive yourself crazy. If you don't believe me, try it sometime. Your app will grow large, buggy and suck CPU. So let's skip the brain-dead solutions and do it right.

Use `select`.

In C, coding `select` is fairly complex. In Python, it's a piece of cake, but it's close enough to the C version that if you understand `select` in Python, you'll have little trouble with it in C:

```
ready_to_read, ready_to_write, in_error = \
               select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

You pass `select` three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The `select` call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If a socket is in the output readable list, you can be as-close-to-certain-as-we-ever-get-in-this-business that a `recv` on that socket will return *something*. Same idea for the writable list. You'll be able to send *something*. Maybe not all you want to, but *something* is better than nothing. (Actually, any reasonably healthy socket will return as writable - it just means outbound network buffer space is available.)

If you have a "server" socket, put it in the potential_readers list. If it comes out in the readable list, your `accept` will (almost certainly) work. If you have created a new socket to `connect` to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

One very nasty problem with `select`: if somewhere in those input lists of sockets is one which has died a nasty death, the `select` will fail. You then need to loop through every single damn socket in all those lists and do a `select([sock],[],[],0)` until you find the bad one. That timeout of 0 means it won't take long, but it's ugly.

Actually, `select` can be handy even with blocking sockets. It's one way of determining whether you will block - the socket returns as readable when there's something in the buffers. However, this still doesn't help with the problem of determining whether the other end is done, or just busy with something else.

Portability alert: On Unix, `select` works both with the sockets and files. Don't try this on Windows. On Windows, `select` works with sockets only. Also note that in C, many of the more advanced socket options are done differently on Windows. In fact, on Windows I usually use threads (which work very, very well) with my sockets. Face it, if you want any kind of performance, your code will look very different on Windows than on Unix.

5.1 Performance

There's no question that the fastest sockets code uses non-blocking sockets and select to multiplex them. You can put together something that will saturate a LAN connection without putting any strain on the CPU. The trouble is that an app written this way can't do much of anything else - it needs to be ready to shuffle bytes around at all times.

Assuming that your app is actually supposed to do something more than that, threading is the optimal solution, (and using non-blocking sockets will be faster than using blocking sockets). Unfortunately, threading support in Unixes varies both in API and quality. So the normal Unix solution is to fork a subprocess to deal with each connection. The overhead for this is significant (and don't do this on Windows - the overhead of process creation is enormous there). It also means that unless each subprocess is completely independent, you'll need to use another form of IPC, say a pipe, or shared memory and semaphores, to communicate between the parent and child processes.

Finally, remember that even though blocking sockets are somewhat slower than non-blocking, in many cases they are the "right" solution. After all, if your app is driven by the data it receives over a socket, there's not much sense in complicating the logic just so your app can wait on `select` instead of `recv`.

Sorting HOW TO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Andrew Dalke and Raymond Hettinger

Release 0.1

Python lists have a built-in `list.sort()` method that modifies the list in-place. There is also a `sorted()` built-in function that builds a new sorted list from an iterable.

In this document, we explore the various techniques for sorting data using Python.

1 Sorting Basics

A simple ascending sort is very easy: just call the `sorted()` function. It returns a new sorted list:

```
>>> sorted([5, 2, 3, 1, 4])
[1, 2, 3, 4, 5]
```

You can also use the `list.sort()` method of a list. It modifies the list in-place (and returns `None` to avoid confusion). Usually it's less convenient than `sorted()` - but if you don't need the original list, it's slightly more efficient.

```
>>> a = [5, 2, 3, 1, 4]
>>> a.sort()
>>> a
[1, 2, 3, 4, 5]
```

Another difference is that the `list.sort()` method is only defined for lists. In contrast, the `sorted()` function accepts any iterable.

```
>>> sorted({1: 'D', 2: 'B', 3: 'B', 4: 'E', 5: 'A'})
[1, 2, 3, 4, 5]
```

2 Key Functions

Starting with Python 2.4, both `list.sort()` and `sorted()` added a *key* parameter to specify a function to be called on each list element prior to making comparisons.

For example, here's a case-insensitive string comparison:

```
>>> sorted("This is a test string from Andrew".split(), key=str.lower)
['a', 'Andrew', 'from', 'is', 'string', 'test', 'This']
```

The value of the *key* parameter should be a function that takes a single argument and returns a key to use for sorting purposes. This technique is fast because the key function is called exactly once for each input record.

A common pattern is to sort complex objects using some of the object's indices as keys. For example:

```
>>> student_tuples = [
...     ('john', 'A', 15),
...     ('jane', 'B', 12),
...     ('dave', 'B', 10),
... ]
>>> sorted(student_tuples, key=lambda student: student[2])   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The same technique works for objects with named attributes. For example:

```
>>> class Student:
...     def __init__(self, name, grade, age):
...         self.name = name
...         self.grade = grade
...         self.age = age
...     def __repr__(self):
...         return repr((self.name, self.grade, self.age))
```

```
>>> student_objects = [
...     Student('john', 'A', 15),
...     Student('jane', 'B', 12),
...     Student('dave', 'B', 10),
... ]
>>> sorted(student_objects, key=lambda student: student.age)   # sort by age
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

3 Operator Module Functions

The key-function patterns shown above are very common, so Python provides convenience functions to make accessor functions easier and faster. The operator module has `operator.itemgetter()`, `operator.attrgetter()`, and starting in Python 2.5 an `operator.methodcaller()` function.

Using those functions, the above examples become simpler and faster:

```
>>> from operator import itemgetter, attrgetter
```

```
>>> sorted(student_tuples, key=itemgetter(2))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

```
>>> sorted(student_objects, key=attrgetter('age'))
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The operator module functions allow multiple levels of sorting. For example, to sort by *grade* then by *age*:

```
>>> sorted(student_tuples, key=itemgetter(1,2))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

```
>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]
```

The `operator.methodcaller()` function makes method calls with fixed parameters for each object being sorted. For example, the `str.count()` method could be used to compute message priority by counting the number of exclamation marks in a message:

```
>>> from operator import methodcaller
>>> messages = ['critical!!!', 'hurry!', 'standby', 'immediate!!']
>>> sorted(messages, key=methodcaller('count', '!'))
['standby', 'hurry!', 'immediate!!', 'critical!!!']
```

4 Ascending and Descending

Both `list.sort()` and `sorted()` accept a *reverse* parameter with a boolean value. This is used to flag descending sorts. For example, to get the student data in reverse *age* order:

```
>>> sorted(student_tuples, key=itemgetter(2), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

```
>>> sorted(student_objects, key=attrgetter('age'), reverse=True)
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

5 Sort Stability and Complex Sorts

Starting with Python 2.2, sorts are guaranteed to be stable. That means that when multiple records have the same key, their original order is preserved.

```
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> sorted(data, key=itemgetter(0))
[('blue', 1), ('blue', 2), ('red', 1), ('red', 2)]
```

Notice how the two records for *blue* retain their original order so that `('blue', 1)` is guaranteed to precede `('blue', 2)`.

This wonderful property lets you build complex sorts in a series of sorting steps. For example, to sort the student data by descending *grade* and then ascending *age*, do the *age* sort first and then sort again using *grade*:

```
>>> s = sorted(student_objects, key=attrgetter('age'))     # sort on secondary key
>>> sorted(s, key=attrgetter('grade'), reverse=True)       # now sort on primary key, descending
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

The Timsort algorithm used in Python does multiple sorts efficiently because it can take advantage of any ordering already present in a dataset.

6 The Old Way Using Decorate-Sort-Undecorate

This idiom is called Decorate-Sort-Undecorate after its three steps:

- First, the initial list is decorated with new values that control the sort order.
- Second, the decorated list is sorted.
- Finally, the decorations are removed, creating a list that contains only the initial values in the new order.

For example, to sort the student data by *grade* using the DSU approach:

```
>>> decorated = [(student.grade, i, student) for i, student in enumerate(student_objects)]
>>> decorated.sort()
>>> [student for grade, i, student in decorated]               # undecorate
[('john', 'A', 15), ('jane', 'B', 12), ('dave', 'B', 10)]
```

This idiom works because tuples are compared lexicographically; the first items are compared; if they are the same then the second items are compared, and so on.

It is not strictly necessary in all cases to include the index i in the decorated list, but including it gives two benefits:

- The sort is stable – if two items have the same key, their order will be preserved in the sorted list.
- The original items do not have to be comparable because the ordering of the decorated tuples will be determined by at most the first two items. So for example the original list could contain complex numbers which cannot be sorted directly.

Another name for this idiom is Schwartzian transform, after Randal L. Schwartz, who popularized it among Perl programmers.

For large lists and lists where the comparison information is expensive to calculate, and Python versions before 2.4, DSU is likely to be the fastest way to sort the list. For 2.4 and later, key functions provide the same functionality.

7 The Old Way Using the *cmp* Parameter

Many constructs given in this HOWTO assume Python 2.4 or later. Before that, there was no `sorted()` builtin and `list.sort()` took no keyword arguments. Instead, all of the Py2.x versions supported a *cmp* parameter to handle user specified comparison functions.

In Python 3, the *cmp* parameter was removed entirely (as part of a larger effort to simplify and unify the language, eliminating the conflict between rich comparisons and the `__cmp__()` magic method).

In Python 2, `sort()` allowed an optional function which can be called for doing the comparisons. That function should take two arguments to be compared and then return a negative value for less-than, return zero if they are equal, or return a positive value for greater-than. For example, we can do:

```
>>> def numeric_compare(x, y):
...     return x - y
>>> sorted([5, 2, 4, 1, 3], cmp=numeric_compare)
[1, 2, 3, 4, 5]
```

Or you can reverse the order of comparison with:

```
>>> def reverse_numeric(x, y):
...     return y - x
>>> sorted([5, 2, 4, 1, 3], cmp=reverse_numeric)
[5, 4, 3, 2, 1]
```

When porting code from Python 2.x to 3.x, the situation can arise when you have the user supplying a comparison function and you need to convert that to a key function. The following wrapper makes that easy to do:

```
def cmp_to_key(mycmp):
    'Convert a cmp= function into a key= function'
    class K(object):
        def __init__(self, obj, *args):
            self.obj = obj
        def __lt__(self, other):
            return mycmp(self.obj, other.obj) < 0
        def __gt__(self, other):
            return mycmp(self.obj, other.obj) > 0
        def __eq__(self, other):
            return mycmp(self.obj, other.obj) == 0
        def __le__(self, other):
            return mycmp(self.obj, other.obj) <= 0
        def __ge__(self, other):
            return mycmp(self.obj, other.obj) >= 0
        def __ne__(self, other):
            return mycmp(self.obj, other.obj) != 0
    return K
```

To convert to a key function, just wrap the old comparison function:

```
>>> sorted([5, 2, 4, 1, 3], key=cmp_to_key(reverse_numeric))
[5, 4, 3, 2, 1]
```

In Python 2.7, the `functools.cmp_to_key()` function was added to the functools module.

8 Odd and Ends

- For locale aware sorting, use `locale.strxfrm()` for a key function or `locale.strcoll()` for a comparison function.
- The *reverse* parameter still maintains sort stability (so that records with equal keys retain their original order). Interestingly, that effect can be simulated without the parameter by using the builtin `reversed()` function twice:

```
>>> data = [('red', 1), ('blue', 1), ('red', 2), ('blue', 2)]
>>> standard_way = sorted(data, key=itemgetter(0), reverse=True)
>>> double_reversed = list(reversed(sorted(reversed(data), key=itemgetter(0))))
>>> assert standard_way == double_reversed
>>> standard_way
[('red', 1), ('red', 2), ('blue', 1), ('blue', 2)]
```

- To create a standard sort order for a class, just add the appropriate rich comparison methods:

```
>>> Student.__eq__ = lambda self, other: self.age == other.age
>>> Student.__ne__ = lambda self, other: self.age != other.age
>>> Student.__lt__ = lambda self, other: self.age < other.age
>>> Student.__le__ = lambda self, other: self.age <= other.age
>>> Student.__gt__ = lambda self, other: self.age > other.age
>>> Student.__ge__ = lambda self, other: self.age >= other.age
>>> sorted(student_objects)
[('dave', 'B', 10), ('jane', 'B', 12), ('john', 'A', 15)]
```

  For general purpose comparisons, the recommended approach is to define all six rich comparison operators. The `functools.total_ordering()` class decorator makes this easy to implement.

- Key functions need not depend directly on the objects being sorted. A key function can also access external resources. For instance, if the student grades are stored in a dictionary, they can be used to sort a separate list of student names:

```
>>> students = ['dave', 'john', 'jane']
>>> grades = {'john': 'F', 'jane':'A', 'dave': 'C'}
>>> sorted(students, key=grades.__getitem__)
['jane', 'dave', 'john']
```

Unicode HOWTO

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Release 1.03

This HOWTO discusses Python 2.x's support for Unicode, and explains various problems that people commonly encounter when trying to work with Unicode. For the Python 3 version, see <https://docs.python.org/3/howto/unicode.html>.

1 Introduction to Unicode

1.1 History of Character Codes

In 1968, the American Standard Code for Information Interchange, better known by its acronym ASCII, was standardized. ASCII defined numeric codes for various characters, with the numeric values running from 0 to 127. For example, the lowercase letter 'a' is assigned 97 as its code value.

ASCII was an American-developed standard, so it only defined unaccented characters. There was an 'e', but no 'é' or 'Í'. This meant that languages which required accented characters couldn't be faithfully represented in ASCII. (Actually the missing accents matter for English, too, which contains words such as 'naïve' and 'café', and some publications have house styles which require spellings such as 'coöperate'.)

For a while people just wrote programs that didn't display accents. I remember looking at Apple][BASIC programs, published in French-language publications in the mid-1980s, that had lines like these:

```
PRINT "MISE A JOUR TERMINEE"
PRINT "PARAMETRES ENREGISTRES"
```

Those messages should contain accents, and they just look wrong to someone who can read French.

In the 1980s, almost all personal computers were 8-bit, meaning that bytes could hold values ranging from 0 to 255. ASCII codes only went up to 127, so some machines assigned values between 128 and 255 to accented characters. Different machines had different codes, however, which led to problems exchanging files. Eventually various commonly used sets of values for the 128–255 range emerged. Some were true standards, defined by the International Organization for Standardization, and some were *de facto* conventions that were invented by one company or another and managed to catch on.

255 characters aren't very many. For example, you can't fit both the accented characters used in Western Europe and the Cyrillic alphabet used for Russian into the 128–255 range because there are more than 128 such characters.

You could write files using different codes (all your Russian files in a coding system called KOI8, all your French files in a different coding system called Latin1), but what if you wanted to write a French document that quotes some Russian text? In the 1980s people began to want to solve this problem, and the Unicode standardization effort began.

Unicode started out using 16-bit characters instead of 8-bit characters. 16 bits means you have 2^16 = 65,536 distinct values available, making it possible to represent many different characters from many different alphabets; an initial goal was to have Unicode contain the alphabets for every single human language. It turns out that even 16 bits isn't enough to meet that goal, and the modern Unicode specification uses a wider range of codes, 0–1,114,111 (0x10ffff in base-16).

There's a related ISO standard, ISO 10646. Unicode and ISO 10646 were originally separate efforts, but the specifications were merged with the 1.1 revision of Unicode.

(This discussion of Unicode's history is highly simplified. I don't think the average Python programmer needs to worry about the historical details; consult the Unicode consortium site listed in the References for more information.)

1.2 Definitions

A **character** is the smallest possible component of a text. 'A', 'B', 'C', etc., are all different characters. So are 'È' and 'Í'. Characters are abstractions, and vary depending on the language or context you're talking about. For example, the symbol for ohms (Ω) is usually drawn much like the capital letter omega (Ω) in the Greek alphabet (they may even be the same in some fonts), but these are two different characters that have different meanings.

The Unicode standard describes how characters are represented by **code points**. A code point is an integer value, usually denoted in base 16. In the standard, a code point is written using the notation U+12ca to mean the character with value 0x12ca (4810 decimal). The Unicode standard contains a lot of tables listing characters and their corresponding code points:

```
0061    'a'; LATIN SMALL LETTER A
0062    'b'; LATIN SMALL LETTER B
0063    'c'; LATIN SMALL LETTER C
...
007B    '{'; LEFT CURLY BRACKET
```

Strictly, these definitions imply that it's meaningless to say 'this is character U+12ca'. U+12ca is a code point, which represents some particular character; in this case, it represents the character 'ETHIOPIC SYLLABLE WI'. In informal contexts, this distinction between code points and characters will sometimes be forgotten.

A character is represented on a screen or on paper by a set of graphical elements that's called a **glyph**. The glyph for an uppercase A, for example, is two diagonal strokes and a horizontal stroke, though the exact details will depend on the font being used. Most Python code doesn't need to worry about glyphs; figuring out the correct glyph to display is generally the job of a GUI toolkit or a terminal's font renderer.

1.3 Encodings

To summarize the previous section: a Unicode string is a sequence of code points, which are numbers from 0 to 0x10ffff. This sequence needs to be represented as a set of bytes (meaning, values from 0–255) in memory. The rules for translating a Unicode string into a sequence of bytes are called an **encoding**.

The first encoding you might think of is an array of 32-bit integers. In this representation, the string "Python" would look like this:

```
   P           y           t           h           o           n
0x50 00 00 00 79 00 00 00 74 00 00 00 68 00 00 00 6f 00 00 00 6e 00 00 00
   0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
```

This representation is straightforward but using it presents a number of problems.

1. It's not portable; different processors order the bytes differently.
2. It's very wasteful of space. In most texts, the majority of the code points are less than 127, or less than 255, so a lot of space is occupied by zero bytes. The above string takes 24 bytes compared to the 6 bytes needed for an ASCII representation. Increased RAM usage doesn't matter too much (desktop computers have megabytes of RAM, and strings aren't usually that large), but expanding our usage of disk and network bandwidth by a factor of 4 is intolerable.
3. It's not compatible with existing C functions such as `strlen()`, so a new family of wide string functions would need to be used.
4. Many Internet standards are defined in terms of textual data, and can't handle content with embedded zero bytes.

Generally people don't use this encoding, instead choosing other encodings that are more efficient and convenient. UTF-8 is probably the most commonly supported encoding; it will be discussed below.

Encodings don't have to handle every possible Unicode character, and most encodings don't. For example, Python's default encoding is the 'ascii' encoding. The rules for converting a Unicode string into the ASCII encoding are simple; for each code point:

1. If the code point is < 128, each byte is the same as the value of the code point.

2. If the code point is 128 or greater, the Unicode string can't be represented in this encoding. (Python raises a `UnicodeEncodeError` exception in this case.)

Latin-1, also known as ISO-8859-1, is a similar encoding. Unicode code points 0–255 are identical to the Latin-1 values, so converting to this encoding simply requires converting code points to byte values; if a code point larger than 255 is encountered, the string can't be encoded into Latin-1.

Encodings don't have to be simple one-to-one mappings like Latin-1. Consider IBM's EBCDIC, which was used on IBM mainframes. Letter values weren't in one block: 'a' through 'i' had values from 129 to 137, but 'j' through 'r' were 145 through 153. If you wanted to use EBCDIC as an encoding, you'd probably use some sort of lookup table to perform the conversion, but this is largely an internal detail.

UTF-8 is one of the most commonly used encodings. UTF stands for "Unicode Transformation Format", and the '8' means that 8-bit numbers are used in the encoding. (There's also a UTF-16 encoding, but it's less frequently used than UTF-8.) UTF-8 uses the following rules:

1. If the code point is <128, it's represented by the corresponding byte value.
2. If the code point is between 128 and 0x7ff, it's turned into two byte values between 128 and 255.
3. Code points >0x7ff are turned into three- or four-byte sequences, where each byte of the sequence is between 128 and 255.

UTF-8 has several convenient properties:

1. It can handle any Unicode code point.
2. A Unicode string is turned into a string of bytes containing no embedded zero bytes. This avoids byte-ordering issues, and means UTF-8 strings can be processed by C functions such as `strcpy()` and sent through protocols that can't handle zero bytes.
3. A string of ASCII text is also valid UTF-8 text.
4. UTF-8 is fairly compact; the majority of code points are turned into two bytes, and values less than 128 occupy only a single byte.
5. If bytes are corrupted or lost, it's possible to determine the start of the next UTF-8-encoded code point and resynchronize. It's also unlikely that random 8-bit data will look like valid UTF-8.

1.4 References

The Unicode Consortium site at <http://www.unicode.org> has character charts, a glossary, and PDF versions of the Unicode specification. Be prepared for some difficult reading. <http://www.unicode.org/history/> is a chronology of the origin and development of Unicode.

To help understand the standard, Jukka Korpela has written an introductory guide to reading the Unicode character tables, available at <https://www.cs.tut.fi/~jkorpela/unicode/guide.html>.

Another good introductory article was written by Joel Spolsky <http://www.joelonsoftware.com/articles/Unicode.html>. If this introduction didn't make things clear to you, you should try reading this alternate article before continuing.

Wikipedia entries are often helpful; see the entries for "character encoding" <http://en.wikipedia.org/wiki/Character_encoding> and UTF-8 <http://en.wikipedia.org/wiki/UTF-8>, for example.

2 Python 2.x's Unicode Support

Now that you've learned the rudiments of Unicode, we can look at Python's Unicode features.

2.1 The Unicode Type

Unicode strings are expressed as instances of the `unicode` type, one of Python's repertoire of built-in types. It derives from an abstract type called `basestring`, which is also an ancestor of the `str` type; you can therefore check if a value is a string type with `isinstance(value, basestring)`. Under the hood, Python represents Unicode strings as either 16- or 32-bit integers, depending on how the Python interpreter was compiled.

The `unicode()` constructor has the signature `unicode(string[, encoding, errors])`. All of its arguments should be 8-bit strings. The first argument is converted to Unicode using the specified encoding; if you leave off the `encoding` argument, the ASCII encoding is used for the conversion, so characters greater than 127 will be treated as errors:

```
>>> unicode('abcdef')
u'abcdef'
>>> s = unicode('abcdef')
>>> type(s)
<type 'unicode'>
>>> unicode('abcdef' + chr(255))
Traceback (most recent call last):
...
UnicodeDecodeError: 'ascii' codec can't decode byte 0xff in position 6:
ordinal not in range(128)
```

The `errors` argument specifies the response when the input string can't be converted according to the encoding's rules. Legal values for this argument are 'strict' (raise a `UnicodeDecodeError` exception), 'replace' (add U+FFFD, 'REPLACEMENT CHARACTER'), or 'ignore' (just leave the character out of the Unicode result). The following examples show the differences:

```
>>> unicode('\x80abc', errors='strict')
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'ascii' codec can't decode byte 0x80 in position 0:
ordinal not in range(128)
>>> unicode('\x80abc', errors='replace')
u'\ufffdabc'
>>> unicode('\x80abc', errors='ignore')
u'abc'
```

Encodings are specified as strings containing the encoding's name. Python 2.7 comes with roughly 100 different encodings; see the Python Library Reference at standard-encodings for a list. Some encodings have multiple names; for example, 'latin-1', 'iso_8859_1' and '8859' are all synonyms for the same encoding.

One-character Unicode strings can also be created with the `unichr()` built-in function, which takes integers and returns a Unicode string of length 1 that contains the corresponding code point. The reverse operation is the built-in `ord()` function that takes a one-character Unicode string and returns the code point value:

```
>>> unichr(40960)
u'\ua000'
>>> ord(u'\ua000')
40960
```

Instances of the `unicode` type have many of the same methods as the 8-bit string type for operations such as searching and formatting:

```
>>> s = u'Was ever feather so lightly blown to and fro as this multitude?'
>>> s.count('e')
5
```

```
>>> s.find('feather')
9
>>> s.find('bird')
-1
>>> s.replace('feather', 'sand')
u'Was ever sand so lightly blown to and fro as this multitude?'
>>> s.upper()
u'WAS EVER FEATHER SO LIGHTLY BLOWN TO AND FRO AS THIS MULTITUDE?'
```

Note that the arguments to these methods can be Unicode strings or 8-bit strings. 8-bit strings will be converted to Unicode before carrying out the operation; Python's default ASCII encoding will be used, so characters greater than 127 will cause an exception:

```
>>> s.find('Was\x9f')
Traceback (most recent call last):
    ...
UnicodeDecodeError: 'ascii' codec can't decode byte 0x9f in position 3:
ordinal not in range(128)
>>> s.find(u'Was\x9f')
-1
```

Much Python code that operates on strings will therefore work with Unicode strings without requiring any changes to the code. (Input and output code needs more updating for Unicode; more on this later.)

Another important method is `.encode([encoding], [errors='strict'])`, which returns an 8-bit string version of the Unicode string, encoded in the requested encoding. The `errors` parameter is the same as the parameter of the `unicode()` constructor, with one additional possibility; as well as 'strict', 'ignore', and 'replace', you can also pass 'xmlcharrefreplace' which uses XML's character references. The following example shows the different results:

```
>>> u = unichr(40960) + u'abcd' + unichr(1972)
>>> u.encode('utf-8')
'\xea\x80\x80abcd\xde\xb4'
>>> u.encode('ascii')
Traceback (most recent call last):
    ...
UnicodeEncodeError: 'ascii' codec can't encode character u'\ua000' in
position 0: ordinal not in range(128)
>>> u.encode('ascii', 'ignore')
'abcd'
>>> u.encode('ascii', 'replace')
'?abcd?'
>>> u.encode('ascii', 'xmlcharrefreplace')
'&#40960;abcd&#1972;'
```

Python's 8-bit strings have a `.decode([encoding], [errors])` method that interprets the string using the given encoding:

```
>>> u = unichr(40960) + u'abcd' + unichr(1972)   # Assemble a string
>>> utf8_version = u.encode('utf-8')             # Encode as UTF-8
>>> type(utf8_version), utf8_version
(<type 'str'>, '\xea\x80\x80abcd\xde\xb4')
>>> u2 = utf8_version.decode('utf-8')            # Decode using UTF-8
>>> u == u2                                      # The two strings match
True
```

The low-level routines for registering and accessing the available encodings are found in the `codecs` module. However, the encoding and decoding functions returned by this module are usually more low-level than is

comfortable, so I'm not going to describe the `codecs` module here. If you need to implement a completely new encoding, you'll need to learn about the `codecs` module interfaces, but implementing encodings is a specialized task that also won't be covered here. Consult the Python documentation to learn more about this module.

The most commonly used part of the `codecs` module is the `codecs.open()` function which will be discussed in the section on input and output.

2.2 Unicode Literals in Python Source Code

In Python source code, Unicode literals are written as strings prefixed with the 'u' or 'U' character: `u'abcdefghijk'`. Specific code points can be written using the `\u` escape sequence, which is followed by four hex digits giving the code point. The `\U` escape sequence is similar, but expects 8 hex digits, not 4.

Unicode literals can also use the same escape sequences as 8-bit strings, including `\x`, but `\x` only takes two hex digits so it can't express an arbitrary code point. Octal escapes can go up to U+01ff, which is octal 777.

```
>>> s = u"a\xac\u1234\u20ac\U00008000"
... #      ^^^^ two-digit hex escape
... #          ^^^^^^ four-digit Unicode escape
... #                      ^^^^^^^^^^ eight-digit Unicode escape
>>> for c in s:  print ord(c),
...
97 172 4660 8364 32768
```

Using escape sequences for code points greater than 127 is fine in small doses, but becomes an annoyance if you're using many accented characters, as you would in a program with messages in French or some other accent-using language. You can also assemble strings using the `unichr()` built-in function, but this is even more tedious.

Ideally, you'd want to be able to write literals in your language's natural encoding. You could then edit Python source code with your favorite editor which would display the accented characters naturally, and have the right characters used at runtime.

Python supports writing Unicode literals in any encoding, but you have to declare the encoding being used. This is done by including a special comment as either the first or second line of the source file:

```
#!/usr/bin/env python
# -*- coding: latin-1 -*-

u = u'abcdé'
print ord(u[-1])
```

The syntax is inspired by Emacs's notation for specifying variables local to a file. Emacs supports many different variables, but Python only supports 'coding'. The `-*-` symbols indicate to Emacs that the comment is special; they have no significance to Python but are a convention. Python looks for `coding: name` or `coding=name` in the comment.

If you don't include such a comment, the default encoding used will be ASCII. Versions of Python before 2.4 were Euro-centric and assumed Latin-1 as a default encoding for string literals; in Python 2.4, characters greater than 127 still work but result in a warning. For example, the following program has no encoding declaration:

```
#!/usr/bin/env python
u = u'abcdé'
print ord(u[-1])
```

When you run it with Python 2.4, it will output the following warning:

```
amk:~$ python2.4 p263.py
sys:1: DeprecationWarning: Non-ASCII character '\xe9'
     in file p263.py on line 2, but no encoding declared;
     see https://www.python.org/peps/pep-0263.html for details
```

Python 2.5 and higher are stricter and will produce a syntax error:

```
amk:~$ python2.5 p263.py
File "/tmp/p263.py", line 2
SyntaxError: Non-ASCII character '\xc3' in file /tmp/p263.py
  on line 2, but no encoding declared; see
  https://www.python.org/peps/pep-0263.html for details
```

2.3 Unicode Properties

The Unicode specification includes a database of information about code points. For each code point that's defined, the information includes the character's name, its category, the numeric value if applicable (Unicode has characters representing the Roman numerals and fractions such as one-third and four-fifths). There are also properties related to the code point's use in bidirectional text and other display-related properties.

The following program displays some information about several characters, and prints the numeric value of one particular character:

```
import unicodedata

u = unichr(233) + unichr(0x0bf2) + unichr(3972) + unichr(6000) + unichr(13231)

for i, c in enumerate(u):
    print i, '%04x' % ord(c), unicodedata.category(c),
    print unicodedata.name(c)

# Get numeric value of second character
print unicodedata.numeric(u[1])
```

When run, this prints:

```
0 00e9 Ll LATIN SMALL LETTER E WITH ACUTE
1 0bf2 No TAMIL NUMBER ONE THOUSAND
2 0f84 Mn TIBETAN MARK HALANTA
3 1770 Lo TAGBANWA LETTER SA
4 33af So SQUARE RAD OVER S SQUARED
1000.0
```

The category codes are abbreviations describing the nature of the character. These are grouped into categories such as "Letter", "Number", "Punctuation", or "Symbol", which in turn are broken up into subcategories. To take the codes from the above output, `'Ll'` means 'Letter, lowercase', `'No'` means "Number, other", `'Mn'` is "Mark, nonspacing", and `'So'` is "Symbol, other". See <http://www.unicode.org/reports/tr44/#General_Category_Values> for a list of category codes.

2.4 References

The Unicode and 8-bit string types are described in the Python library reference at typesseq.

The documentation for the `unicodedata` module.

The documentation for the `codecs` module.

Marc-André Lemburg gave a presentation at EuroPython 2002 titled “Python and Unicode”. A PDF version of his slides is available at <https://downloads.egenix.com/python/Unicode-EPC2002-Talk.pdf>, and is an excellent overview of the design of Python’s Unicode features.

3 Reading and Writing Unicode Data

Once you’ve written some code that works with Unicode data, the next problem is input/output. How do you get Unicode strings into your program, and how do you convert Unicode into a form suitable for storage or transmission?

It’s possible that you may not need to do anything depending on your input sources and output destinations; you should check whether the libraries used in your application support Unicode natively. XML parsers often return Unicode data, for example. Many relational databases also support Unicode-valued columns and can return Unicode values from an SQL query.

Unicode data is usually converted to a particular encoding before it gets written to disk or sent over a socket. It’s possible to do all the work yourself: open a file, read an 8-bit string from it, and convert the string with `unicode(str, encoding)`. However, the manual approach is not recommended.

One problem is the multi-byte nature of encodings; one Unicode character can be represented by several bytes. If you want to read the file in arbitrary-sized chunks (say, 1K or 4K), you need to write error-handling code to catch the case where only part of the bytes encoding a single Unicode character are read at the end of a chunk. One solution would be to read the entire file into memory and then perform the decoding, but that prevents you from working with files that are extremely large; if you need to read a 2Gb file, you need 2Gb of RAM. (More, really, since for at least a moment you’d need to have both the encoded string and its Unicode version in memory.)

The solution would be to use the low-level decoding interface to catch the case of partial coding sequences. The work of implementing this has already been done for you: the `codecs` module includes a version of the `open()` function that returns a file-like object that assumes the file’s contents are in a specified encoding and accepts Unicode parameters for methods such as `.read()` and `.write()`.

The function’s parameters are `open(filename, mode='rb', encoding=None, errors='strict', buffering=1)`. `mode` can be `'r'`, `'w'`, or `'a'`, just like the corresponding parameter to the regular built-in `open()` function; add a `'+'` to update the file. `buffering` is similarly parallel to the standard function’s parameter. `encoding` is a string giving the encoding to use; if it’s left as `None`, a regular Python file object that accepts 8-bit strings is returned. Otherwise, a wrapper object is returned, and data written to or read from the wrapper object will be converted as needed. `errors` specifies the action for encoding errors and can be one of the usual values of ‘strict’, ‘ignore’, and ‘replace’.

Reading Unicode from a file is therefore simple:

```
import codecs
f = codecs.open('unicode.rst', encoding='utf-8')
for line in f:
    print repr(line)
```

It’s also possible to open files in update mode, allowing both reading and writing:

```
f = codecs.open('test', encoding='utf-8', mode='w+')
f.write(u'\u4500 blah blah blah\n')
f.seek(0)
print repr(f.readline()[:1])
f.close()
```

Unicode character U+FEFF is used as a byte-order mark (BOM), and is often written as the first character of a file in order to assist with autodetection of the file’s byte ordering. Some encodings, such as UTF-

16, expect a BOM to be present at the start of a file; when such an encoding is used, the BOM will be automatically written as the first character and will be silently dropped when the file is read. There are variants of these encodings, such as 'utf-16-le' and 'utf-16-be' for little-endian and big-endian encodings, that specify one particular byte ordering and don't skip the BOM.

3.1 Unicode filenames

Most of the operating systems in common use today support filenames that contain arbitrary Unicode characters. Usually this is implemented by converting the Unicode string into some encoding that varies depending on the system. For example, Mac OS X uses UTF-8 while Windows uses a configurable encoding; on Windows, Python uses the name "mbcs" to refer to whatever the currently configured encoding is. On Unix systems, there will only be a filesystem encoding if you've set the `LANG` or `LC_CTYPE` environment variables; if you haven't, the default encoding is ASCII.

The `sys.getfilesystemencoding()` function returns the encoding to use on your current system, in case you want to do the encoding manually, but there's not much reason to bother. When opening a file for reading or writing, you can usually just provide the Unicode string as the filename, and it will be automatically converted to the right encoding for you:

```
filename = u'filename\u4500abc'
f = open(filename, 'w')
f.write('blah\n')
f.close()
```

Functions in the `os` module such as `os.stat()` will also accept Unicode filenames.

`os.listdir()`, which returns filenames, raises an issue: should it return the Unicode version of filenames, or should it return 8-bit strings containing the encoded versions? `os.listdir()` will do both, depending on whether you provided the directory path as an 8-bit string or a Unicode string. If you pass a Unicode string as the path, filenames will be decoded using the filesystem's encoding and a list of Unicode strings will be returned, while passing an 8-bit path will return the 8-bit versions of the filenames. For example, assuming the default filesystem encoding is UTF-8, running the following program:

```
fn = u'filename\u4500abc'
f = open(fn, 'w')
f.close()

import os
print os.listdir('.')
print os.listdir(u'.')
```

will produce the following output:

```
amk:~$ python t.py
['.svn', 'filename\xe4\x94\x80abc', ...]
[u'.svn', u'filename\u4500abc', ...]
```

The first list contains UTF-8-encoded filenames, and the second list contains the Unicode versions.

3.2 Tips for Writing Unicode-aware Programs

This section provides some suggestions on writing software that deals with Unicode.

The most important tip is:

> Software should only work with Unicode strings internally, converting to a particular encoding on output.

If you attempt to write processing functions that accept both Unicode and 8-bit strings, you will find your program vulnerable to bugs wherever you combine the two different kinds of strings. Python's default encoding is ASCII, so whenever a character with an ASCII value > 127 is in the input data, you'll get a `UnicodeDecodeError` because that character can't be handled by the ASCII encoding.

It's easy to miss such problems if you only test your software with data that doesn't contain any accents; everything will seem to work, but there's actually a bug in your program waiting for the first user who attempts to use characters > 127. A second tip, therefore, is:

> Include characters > 127 and, even better, characters > 255 in your test data.

When using data coming from a web browser or some other untrusted source, a common technique is to check for illegal characters in a string before using the string in a generated command line or storing it in a database. If you're doing this, be careful to check the string once it's in the form that will be used or stored; it's possible for encodings to be used to disguise characters. This is especially true if the input data also specifies the encoding; many encodings leave the commonly checked-for characters alone, but Python includes some encodings such as `'base64'` that modify every single character.

For example, let's say you have a content management system that takes a Unicode filename, and you want to disallow paths with a '/' character. You might write this code:

```
def read_file (filename, encoding):
    if '/' in filename:
        raise ValueError("'/' not allowed in filenames")
    unicode_name = filename.decode(encoding)
    f = open(unicode_name, 'r')
    # ... return contents of file ...
```

However, if an attacker could specify the `'base64'` encoding, they could pass `'L2V0Yy9wYXNzd2Q='`, which is the base-64 encoded form of the string `'/etc/passwd'`, to read a system file. The above code looks for `'/'` characters in the encoded form and misses the dangerous character in the resulting decoded form.

3.3 References

The PDF slides for Marc-André Lemburg's presentation "Writing Unicode-aware Applications in Python" are available at <https://downloads.egenix.com/python/LSM2005-Developing-Unicode-aware-applications-in-Python.pdf> and discuss questions of character encodings as well as how to internationalize and localize an application.

4 Revision History and Acknowledgements

Thanks to the following people who have noted errors or offered suggestions on this article: Nicholas Bastin, Marius Gedminas, Kent Johnson, Ken Krugler, Marc-André Lemburg, Martin von Löwis, Chad Whitacre.

Version 1.0: posted August 5 2005.

Version 1.01: posted August 7 2005. Corrects factual and markup errors; adds several links.

Version 1.02: posted August 16 2005. Corrects factual errors.

Version 1.03: posted June 20 2010. Notes that Python 3.x is not covered, and that the HOWTO only covers 2.x.

HOWTO Fetch Internet Resources Using urllib2

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Michael Foord

Note: There is a French translation of an earlier revision of this HOWTO, available at urllib2 - Le Manuel manquant.

1 Introduction

Related Articles

You may also find useful the following article on fetching web resources with Python:

- Basic Authentication

  A tutorial on *Basic Authentication*, with examples in Python.

urllib2 is a Python module for fetching URLs (Uniform Resource Locators). It offers a very simple interface, in the form of the *urlopen* function. This is capable of fetching URLs using a variety of different protocols. It also offers a slightly more complex interface for handling common situations - like basic authentication, cookies, proxies and so on. These are provided by objects called handlers and openers.

urllib2 supports fetching URLs for many "URL schemes" (identified by the string before the `":"` in URL - for example `"ftp"` is the URL scheme of `"ftp://python.org/"`) using their associated network protocols (e.g. FTP, HTTP). This tutorial focuses on the most common case, HTTP.

For straightforward situations *urlopen* is very easy to use. But as soon as you encounter errors or non-trivial cases when opening HTTP URLs, you will need some understanding of the HyperText Transfer Protocol. The most comprehensive and authoritative reference to HTTP is **RFC 2616**. This is a technical document and not intended to be easy to read. This HOWTO aims to illustrate using *urllib2*, with enough detail about HTTP to help you through. It is not intended to replace the `urllib2` docs, but is supplementary to them.

2 Fetching URLs

The simplest way to use urllib2 is as follows:

```
import urllib2
response = urllib2.urlopen('http://python.org/')
html = response.read()
```

Many uses of urllib2 will be that simple (note that instead of an 'http:' URL we could have used a URL starting with 'ftp:', 'file:', etc.). However, it's the purpose of this tutorial to explain the more complicated cases, concentrating on HTTP.

HTTP is based on requests and responses - the client makes requests and servers send responses. urllib2 mirrors this with a `Request` object which represents the HTTP request you are making. In its simplest form you create a Request object that specifies the URL you want to fetch. Calling `urlopen` with this Request object returns a response object for the URL requested. This response is a file-like object, which means you can for example call `.read()` on the response:

```
import urllib2

req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
the_page = response.read()
```

Note that urllib2 makes use of the same Request interface to handle all URL schemes. For example, you can make an FTP request like so:

```
req = urllib2.Request('ftp://example.com/')
```

In the case of HTTP, there are two extra things that Request objects allow you to do: First, you can pass data to be sent to the server. Second, you can pass extra information ("metadata") *about* the data or the about request itself, to the server - this information is sent as HTTP "headers". Let's look at each of these in turn.

2.1 Data

Sometimes you want to send data to a URL (often the URL will refer to a CGI (Common Gateway Interface) script[1] or other web application). With HTTP, this is often done using what's known as a **POST** request. This is often what your browser does when you submit a HTML form that you filled in on the web. Not all POSTs have to come from forms: you can use a POST to transmit arbitrary data to your own application. In the common case of HTML forms, the data needs to be encoded in a standard way, and then passed to the Request object as the `data` argument. The encoding is done using a function from the `urllib` library *not* from `urllib2`.

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
values = {'name' : 'Michael Foord',
          'location' : 'Northampton',
          'language' : 'Python' }

data = urllib.urlencode(values)
req = urllib2.Request(url, data)
response = urllib2.urlopen(req)
the_page = response.read()
```

Note that other encodings are sometimes required (e.g. for file upload from HTML forms - see HTML Specification, Form Submission for more details).

If you do not pass the `data` argument, urllib2 uses a **GET** request. One way in which GET and POST requests differ is that POST requests often have "side-effects": they change the state of the system in some way (for example by placing an order with the website for a hundredweight of tinned spam to be delivered to your door). Though the HTTP standard makes it clear that POSTs are intended to *always* cause side-effects, and GET requests *never* to cause side-effects, nothing prevents a GET request from having side-effects, nor a POST requests from having no side-effects. Data can also be passed in an HTTP GET request by encoding it in the URL itself.

This is done as follows:

```
>>> import urllib2
>>> import urllib
>>> data = {}
>>> data['name'] = 'Somebody Here'
>>> data['location'] = 'Northampton'
>>> data['language'] = 'Python'
>>> url_values = urllib.urlencode(data)
>>> print url_values  # The order may differ.
name=Somebody+Here&language=Python&location=Northampton
```

[1] For an introduction to the CGI protocol see Writing Web Applications in Python.

```
>>> url = 'http://www.example.com/example.cgi'
>>> full_url = url + '?' + url_values
>>> data = urllib2.urlopen(full_url)
```

Notice that the full URL is created by adding a `?` to the URL, followed by the encoded values.

2.2 Headers

We'll discuss here one particular HTTP header, to illustrate how to add headers to your HTTP request.

Some websites[2] dislike being browsed by programs, or send different versions to different browsers[3]. By default urllib2 identifies itself as `Python-urllib/x.y` (where `x` and `y` are the major and minor version numbers of the Python release, e.g. `Python-urllib/2.5`), which may confuse the site, or just plain not work. The way a browser identifies itself is through the `User-Agent` header[4]. When you create a Request object you can pass a dictionary of headers in. The following example makes the same request as above, but identifies itself as a version of Internet Explorer[5].

```
import urllib
import urllib2

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name': 'Michael Foord',
          'location': 'Northampton',
          'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urllib.urlencode(values)
req = urllib2.Request(url, data, headers)
response = urllib2.urlopen(req)
the_page = response.read()
```

The response also has two useful methods. See the section on *info and geturl* which comes after we have a look at what happens when things go wrong.

3 Handling Exceptions

urlopen raises `URLError` when it cannot handle a response (though as usual with Python APIs, built-in exceptions such as `ValueError`, `TypeError` etc. may also be raised).

`HTTPError` is the subclass of `URLError` raised in the specific case of HTTP URLs.

3.1 URLError

Often, URLError is raised because there is no network connection (no route to the specified server), or the specified server doesn't exist. In this case, the exception raised will have a 'reason' attribute, which is a tuple containing an error code and a text error message.

e.g.

[2] Google for example.

[3] Browser sniffing is a very bad practice for website design - building sites using web standards is much more sensible. Unfortunately a lot of sites still send different versions to different browsers.

[4] The user agent for MSIE 6 is *'Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 1.1.4322)'*

[5] For details of more HTTP request headers, see Quick Reference to HTTP Headers.

```
>>> req = urllib2.Request('http://www.pretend_server.org')
>>> try: urllib2.urlopen(req)
... except urllib2.URLError as e:
...    print e.reason
...
(4, 'getaddrinfo failed')
```

3.2 HTTPError

Every HTTP response from the server contains a numeric "status code". Sometimes the status code indicates that the server is unable to fulfil the request. The default handlers will handle some of these responses for you (for example, if the response is a "redirection" that requests the client fetch the document from a different URL, urllib2 will handle that for you). For those it can't handle, urlopen will raise an `HTTPError`. Typical errors include '404' (page not found), '403' (request forbidden), and '401' (authentication required).

See section 10 of RFC 2616 for a reference on all the HTTP error codes.

The `HTTPError` instance raised will have an integer 'code' attribute, which corresponds to the error sent by the server.

Error Codes

Because the default handlers handle redirects (codes in the 300 range), and codes in the 100–299 range indicate success, you will usually only see error codes in the 400–599 range.

`BaseHTTPServer.BaseHTTPRequestHandler.responses` is a useful dictionary of response codes in that shows all the response codes used by RFC 2616. The dictionary is reproduced here for convenience

```
# Table mapping response codes to messages; entries have the
# form {code: (shortmessage, longmessage)}.
responses = {
    100: ('Continue', 'Request received, please continue'),
    101: ('Switching Protocols',
          'Switching to new protocol; obey Upgrade header'),

    200: ('OK', 'Request fulfilled, document follows'),
    201: ('Created', 'Document created, URL follows'),
    202: ('Accepted',
          'Request accepted, processing continues off-line'),
    203: ('Non-Authoritative Information', 'Request fulfilled from cache'),
    204: ('No Content', 'Request fulfilled, nothing follows'),
    205: ('Reset Content', 'Clear input form for further input.'),
    206: ('Partial Content', 'Partial content follows.'),

    300: ('Multiple Choices',
          'Object has several resources -- see URI list'),
    301: ('Moved Permanently', 'Object moved permanently -- see URI list'),
    302: ('Found', 'Object moved temporarily -- see URI list'),
    303: ('See Other', 'Object moved -- see Method and URL list'),
    304: ('Not Modified',
          'Document has not changed since given time'),
    305: ('Use Proxy',
          'You must use proxy specified in Location to access this '
          'resource.'),
    307: ('Temporary Redirect',
          'Object moved temporarily -- see URI list'),
```

```
    400: ('Bad Request',
          'Bad request syntax or unsupported method'),
    401: ('Unauthorized',
          'No permission -- see authorization schemes'),
    402: ('Payment Required',
          'No payment -- see charging schemes'),
    403: ('Forbidden',
          'Request forbidden -- authorization will not help'),
    404: ('Not Found', 'Nothing matches the given URI'),
    405: ('Method Not Allowed',
          'Specified method is invalid for this server.'),
    406: ('Not Acceptable', 'URI not available in preferred format.'),
    407: ('Proxy Authentication Required', 'You must authenticate with '
          'this proxy before proceeding.'),
    408: ('Request Timeout', 'Request timed out; try again later.'),
    409: ('Conflict', 'Request conflict.'),
    410: ('Gone',
          'URI no longer exists and has been permanently removed.'),
    411: ('Length Required', 'Client must specify Content-Length.'),
    412: ('Precondition Failed', 'Precondition in headers is false.'),
    413: ('Request Entity Too Large', 'Entity is too large.'),
    414: ('Request-URI Too Long', 'URI is too long.'),
    415: ('Unsupported Media Type', 'Entity body in unsupported format.'),
    416: ('Requested Range Not Satisfiable',
          'Cannot satisfy request range.'),
    417: ('Expectation Failed',
          'Expect condition could not be satisfied.'),

    500: ('Internal Server Error', 'Server got itself in trouble'),
    501: ('Not Implemented',
          'Server does not support this operation'),
    502: ('Bad Gateway', 'Invalid responses from another server/proxy.'),
    503: ('Service Unavailable',
          'The server cannot process the request due to a high load'),
    504: ('Gateway Timeout',
          'The gateway server did not receive a timely response'),
    505: ('HTTP Version Not Supported', 'Cannot fulfill request.'),
    }
```

When an error is raised the server responds by returning an HTTP error code *and* an error page. You can use the `HTTPError` instance as a response on the page returned. This means that as well as the code attribute, it also has read, geturl, and info, methods.

```
>>> req = urllib2.Request('http://www.python.org/fish.html')
>>> try:
...     urllib2.urlopen(req)
... except urllib2.HTTPError as e:
...     print e.code
...     print e.read()
...
404
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
<title>Page Not Found</title>
...
```

3.3 Wrapping it Up

So if you want to be prepared for `HTTPError` *or* `URLError` there are two basic approaches. I prefer the second approach.

Number 1

```
from urllib2 import Request, urlopen, URLError, HTTPError
req = Request(someurl)
try:
    response = urlopen(req)
except HTTPError as e:
    print 'The server couldn\'t fulfill the request.'
    print 'Error code: ', e.code
except URLError as e:
    print 'We failed to reach a server.'
    print 'Reason: ', e.reason
else:
    # everything is fine
```

Note: The `except HTTPError` *must* come first, otherwise `except URLError` will *also* catch an `HTTPError`.

Number 2

```
from urllib2 import Request, urlopen, URLError
req = Request(someurl)
try:
    response = urlopen(req)
except URLError as e:
    if hasattr(e, 'reason'):
        print 'We failed to reach a server.'
        print 'Reason: ', e.reason
    elif hasattr(e, 'code'):
        print 'The server couldn\'t fulfill the request.'
        print 'Error code: ', e.code
else:
    # everything is fine
```

4 info and geturl

The response returned by urlopen (or the `HTTPError` instance) has two useful methods `info()` and `geturl()`.

geturl - this returns the real URL of the page fetched. This is useful because `urlopen` (or the opener object used) may have followed a redirect. The URL of the page fetched may not be the same as the URL requested.

info - this returns a dictionary-like object that describes the page fetched, particularly the headers sent by the server. It is currently an `httplib.HTTPMessage` instance.

Typical headers include 'Content-length', 'Content-type', and so on. See the Quick Reference to HTTP Headers for a useful listing of HTTP headers with brief explanations of their meaning and use.

5 Openers and Handlers

When you fetch a URL you use an opener (an instance of the perhaps confusingly-named `urllib2.OpenerDirector`). Normally we have been using the default opener - via `urlopen` - but you can create custom openers. Openers use handlers. All the "heavy lifting" is done by the handlers. Each handler knows how to open URLs for a particular URL scheme (http, ftp, etc.), or how to handle an aspect of URL opening, for example HTTP redirections or HTTP cookies.

You will want to create openers if you want to fetch URLs with specific handlers installed, for example to get an opener that handles cookies, or to get an opener that does not handle redirections.

To create an opener, instantiate an `OpenerDirector`, and then call `.add_handler(some_handler_instance)` repeatedly.

Alternatively, you can use `build_opener`, which is a convenience function for creating opener objects with a single function call. `build_opener` adds several handlers by default, but provides a quick way to add more and/or override the default handlers.

Other sorts of handlers you might want to can handle proxies, authentication, and other common but slightly specialised situations.

`install_opener` can be used to make an `opener` object the (global) default opener. This means that calls to `urlopen` will use the opener you have installed.

Opener objects have an `open` method, which can be called directly to fetch urls in the same way as the `urlopen` function: there's no need to call `install_opener`, except as a convenience.

6 Basic Authentication

To illustrate creating and installing a handler we will use the `HTTPBasicAuthHandler`. For a more detailed discussion of this subject – including an explanation of how Basic Authentication works - see the Basic Authentication Tutorial.

When authentication is required, the server sends a header (as well as the 401 error code) requesting authentication. This specifies the authentication scheme and a 'realm'. The header looks like: `WWW-Authenticate: SCHEME realm="REALM"`.

e.g.

```
WWW-Authenticate: Basic realm="cPanel Users"
```

The client should then retry the request with the appropriate name and password for the realm included as a header in the request. This is 'basic authentication'. In order to simplify this process we can create an instance of `HTTPBasicAuthHandler` and an opener to use this handler.

The `HTTPBasicAuthHandler` uses an object called a password manager to handle the mapping of URLs and realms to passwords and usernames. If you know what the realm is (from the authentication header sent by the server), then you can use a `HTTPPasswordMgr`. Frequently one doesn't care what the realm is. In that case, it is convenient to use `HTTPPasswordMgrWithDefaultRealm`. This allows you to specify a default username and password for a URL. This will be supplied in the absence of you providing an alternative combination for a specific realm. We indicate this by providing `None` as the realm argument to the `add_password` method.

The top-level URL is the first URL that requires authentication. URLs "deeper" than the URL you pass to .add_password() will also match.

```
# create a password manager
password_mgr = urllib2.HTTPPasswordMgrWithDefaultRealm()
```

```
# Add the username and password.
# If we knew the realm, we could use it instead of None.
top_level_url = "http://example.com/foo/"
password_mgr.add_password(None, top_level_url, username, password)

handler = urllib2.HTTPBasicAuthHandler(password_mgr)

# create "opener" (OpenerDirector instance)
opener = urllib2.build_opener(handler)

# use the opener to fetch a URL
opener.open(a_url)

# Install the opener.
# Now all calls to urllib2.urlopen use our opener.
urllib2.install_opener(opener)
```

Note: In the above example we only supplied our `HTTPBasicAuthHandler` to `build_opener`. By default openers have the handlers for normal situations – `ProxyHandler` (if a proxy setting such as an `http_proxy` environment variable is set), `UnknownHandler`, `HTTPHandler`, `HTTPDefaultErrorHandler`, `HTTPRedirectHandler`, `FTPHandler`, `FileHandler`, `HTTPErrorProcessor`.

`top_level_url` is in fact *either* a full URL (including the 'http:' scheme component and the hostname and optionally the port number) e.g. `"http://example.com/"` *or* an "authority" (i.e. the hostname, optionally including the port number) e.g. `"example.com"` or `"example.com:8080"` (the latter example includes a port number). The authority, if present, must NOT contain the "userinfo" component - for example `"joe:password@example.com"` is not correct.

7 Proxies

urllib2 will auto-detect your proxy settings and use those. This is through the `ProxyHandler`, which is part of the normal handler chain when a proxy setting is detected. Normally that's a good thing, but there are occasions when it may not be helpful[6]. One way to do this is to setup our own `ProxyHandler`, with no proxies defined. This is done using similar steps to setting up a Basic Authentication handler:

```
>>> proxy_support = urllib2.ProxyHandler({})
>>> opener = urllib2.build_opener(proxy_support)
>>> urllib2.install_opener(opener)
```

Note: Currently `urllib2` *does not* support fetching of `https` locations through a proxy. However, this can be enabled by extending urllib2 as shown in the recipe[7].

Note: `HTTP_PROXY` will be ignored if a variable `REQUEST_METHOD` is set; see the documentation on `getproxies()`.

---

[6] In my case I have to use a proxy to access the internet at work. If you attempt to fetch *localhost* URLs through this proxy it blocks them. IE is set to use the proxy, which urllib2 picks up on. In order to test scripts with a localhost server, I have to prevent urllib2 from using the proxy.

[7] urllib2 opener for SSL proxy (CONNECT method): ASPN Cookbook Recipe.

8 Sockets and Layers

The Python support for fetching resources from the web is layered. urllib2 uses the httplib library, which in turn uses the socket library.

As of Python 2.3 you can specify how long a socket should wait for a response before timing out. This can be useful in applications which have to fetch web pages. By default the socket module has *no timeout* and can hang. Currently, the socket timeout is not exposed at the httplib or urllib2 levels. However, you can set the default timeout globally for all sockets using

```
import socket
import urllib2

# timeout in seconds
timeout = 10
socket.setdefaulttimeout(timeout)

# this call to urllib2.urlopen now uses the default timeout
# we have set in the socket module
req = urllib2.Request('http://www.voidspace.org.uk')
response = urllib2.urlopen(req)
```

---

9 Footnotes

This document was reviewed and revised by John Lee.

Index

E

H

R

HOWTO Use Python in the web

Release 2.7.14

Guido van Rossum
and the Python development team

April 05, 2018

Python Software Foundation
Email: docs@python.org

Contents

Author Marek Kubica

Abstract

This document shows how Python fits into the web. It presents some ways to integrate Python with a web server, and general practices useful for developing web sites.

Programming for the Web has become a hot topic since the rise of "Web 2.0", which focuses on user-generated content on web sites. It has always been possible to use Python for creating web sites, but it was a rather tedious task. Therefore, many frameworks and helper tools have been created to assist developers in creating faster and more robust sites. This HOWTO describes some of the methods used to combine Python with a web server to create dynamic content. It is not meant as a complete introduction, as this topic is far too broad to be covered in one single document. However, a short overview of the most popular libraries is provided.

See also:

While this HOWTO tries to give an overview of Python in the web, it cannot always be as up to date as desired. Web development in Python is rapidly moving forward, so the wiki page on Web Programming may be more in sync with recent development.

1 The Low-Level View

When a user enters a web site, their browser makes a connection to the site's web server (this is called the *request*). The server looks up the file in the file system and sends it back to the user's browser, which displays it (this is the *response*). This is roughly how the underlying protocol, HTTP, works.

Dynamic web sites are not based on files in the file system, but rather on programs which are run by the web server when a request comes in, and which *generate* the content that is returned to the user. They can do all sorts of useful things, like display the postings of a bulletin board, show your email, configure software, or just display the current time. These programs can be written in any programming language the server supports. Since most servers support Python, it is easy to use Python to create dynamic web sites.

Most HTTP servers are written in C or C++, so they cannot execute Python code directly – a bridge is needed between the server and the program. These bridges, or rather interfaces, define how programs interact with the server. There have been numerous attempts to create the best possible interface, but there are only a few worth mentioning.

Not every web server supports every interface. Many web servers only support old, now-obsolete interfaces; however, they can often be extended using third-party modules to support newer ones.

1.1 Common Gateway Interface

This interface, most commonly referred to as "CGI", is the oldest, and is supported by nearly every web server out of the box. Programs using CGI to communicate with their web server need to be started by the server for every request. So, every request starts a new Python interpreter – which takes some time to start up – thus making the whole interface only usable for low load situations.

The upside of CGI is that it is simple – writing a Python program which uses CGI is a matter of about three lines of code. This simplicity comes at a price: it does very few things to help the developer.

Writing CGI programs, while still possible, is no longer recommended. With *WSGI*, a topic covered later in this document, it is possible to write programs that emulate CGI, so they can be run as CGI if no better option is available.

See also:

The Python standard library includes some modules that are helpful for creating plain CGI programs:

- `cgi` – Handling of user input in CGI scripts
- `cgitb` – Displays nice tracebacks when errors happen in CGI applications, instead of presenting a "500 Internal Server Error" message

The Python wiki features a page on CGI scripts with some additional information about CGI in Python.

Simple script for testing CGI

To test whether your web server works with CGI, you can use this short and simple CGI program:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

# enable debugging
import cgitb
cgitb.enable()

print "Content-Type: text/plain;charset=utf-8"
print

print "Hello World!"
```

Depending on your web server configuration, you may need to save this code with a `.py` or `.cgi` extension. Additionally, this file may also need to be in a `cgi-bin` folder, for security reasons.

You might wonder what the `cgitb` line is about. This line makes it possible to display a nice traceback instead of just crashing and displaying an "Internal Server Error" in the user's browser. This is useful for debugging, but it might risk exposing some confidential data to the user. You should not use `cgitb` in production code for this reason. You should *always* catch exceptions, and display proper error pages – end-users don't like to see nondescript "Internal Server Errors" in their browsers.

Setting up CGI on your own server

If you don't have your own web server, this does not apply to you. You can check whether it works as-is, and if not you will need to talk to the administrator of your web server. If it is a big host, you can try filing a ticket asking for Python support.

If you are your own administrator or want to set up CGI for testing purposes on your own computers, you have to configure it by yourself. There is no single way to configure CGI, as there are many web servers with different configuration options. Currently the most widely used free web server is Apache HTTPd, or Apache for short. Apache can be easily installed on nearly every system using the system's package management tool. lighttpd is another alternative and is said to have better performance. On many systems this server can also be installed using the package management tool, so manually compiling the web server may not be needed.

- On Apache you can take a look at the Dynamic Content with CGI tutorial, where everything is described. Most of the time it is enough just to set `+ExecCGI`. The tutorial also describes the most common gotchas that might arise.
- On lighttpd you need to use the CGI module, which can be configured in a straightforward way. It boils down to setting `cgi.assign` properly.

Common problems with CGI scripts

Using CGI sometimes leads to small annoyances while trying to get these scripts to run. Sometimes a seemingly correct script does not work as expected, the cause being some small hidden problem that's

difficult to spot.

Some of these potential problems are:

- The Python script is not marked as executable. When CGI scripts are not executable most web servers will let the user download it, instead of running it and sending the output to the user. For CGI scripts to run properly on Unix-like operating systems, the `+x` bit needs to be set. Using `chmod a+x your_script.py` may solve this problem.
- On a Unix-like system, The line endings in the program file must be Unix style line endings. This is important because the web server checks the first line of the script (called shebang) and tries to run the program specified there. It gets easily confused by Windows line endings (Carriage Return & Line Feed, also called CRLF), so you have to convert the file to Unix line endings (only Line Feed, LF). This can be done automatically by uploading the file via FTP in text mode instead of binary mode, but the preferred way is just telling your editor to save the files with Unix line endings. Most editors support this.
- Your web server must be able to read the file, and you need to make sure the permissions are correct. On unix-like systems, the server often runs as user and group `www-data`, so it might be worth a try to change the file ownership, or making the file world readable by using `chmod a+r your_script.py`.
- The web server must know that the file you're trying to access is a CGI script. Check the configuration of your web server, as it may be configured to expect a specific file extension for CGI scripts.
- On Unix-like systems, the path to the interpreter in the shebang (`#!/usr/bin/env python`) must be correct. This line calls `/usr/bin/env` to find Python, but it will fail if there is no `/usr/bin/env`, or if Python is not in the web server's path. If you know where your Python is installed, you can also use that full path. The commands `whereis python` and `type -p python` could help you find where it is installed. Once you know the path, you can change the shebang accordingly: `#!/usr/bin/python`.
- The file must not contain a BOM (Byte Order Mark). The BOM is meant for determining the byte order of UTF-16 and UTF-32 encodings, but some editors write this also into UTF-8 files. The BOM interferes with the shebang line, so be sure to tell your editor not to write the BOM.
- If the web server is using *mod_python*, `mod_python` may be having problems. `mod_python` is able to handle CGI scripts by itself, but it can also be a source of issues.

1.2 mod_python

People coming from PHP often find it hard to grasp how to use Python in the web. Their first thought is mostly mod_python, because they think that this is the equivalent to `mod_php`. Actually, there are many differences. What `mod_python` does is embed the interpreter into the Apache process, thus speeding up requests by not having to start a Python interpreter for each request. On the other hand, it is not "Python intermixed with HTML" in the way that PHP is often intermixed with HTML. The Python equivalent of that is a template engine. `mod_python` itself is much more powerful and provides more access to Apache internals. It can emulate CGI, work in a "Python Server Pages" mode (similar to JSP) which is "HTML intermingled with Python", and it has a "Publisher" which designates one file to accept all requests and decide what to do with them.

`mod_python` does have some problems. Unlike the PHP interpreter, the Python interpreter uses caching when executing files, so changes to a file will require the web server to be restarted. Another problem is the basic concept – Apache starts child processes to handle the requests, and unfortunately every child process needs to load the whole Python interpreter even if it does not use it. This makes the whole web server slower. Another problem is that, because `mod_python` is linked against a specific version of `libpython`, it is not possible to switch from an older version to a newer (e.g. 2.4 to 2.5) without recompiling `mod_python`. `mod_python` is also bound to the Apache web server, so programs written for `mod_python` cannot easily run on other web servers.

These are the reasons why `mod_python` should be avoided when writing new programs. In some circumstances it still might be a good idea to use `mod_python` for deployment, but WSGI makes it possible to run WSGI programs under `mod_python` as well.

1.3 FastCGI and SCGI

FastCGI and SCGI try to solve the performance problem of CGI in another way. Instead of embedding the interpreter into the web server, they create long-running background processes. There is still a module in the web server which makes it possible for the web server to "speak" with the background process. As the background process is independent of the server, it can be written in any language, including Python. The language just needs to have a library which handles the communication with the webserver.

The difference between FastCGI and SCGI is very small, as SCGI is essentially just a "simpler FastCGI". As the web server support for SCGI is limited, most people use FastCGI instead, which works the same way. Almost everything that applies to SCGI also applies to FastCGI as well, so we'll only cover the latter.

These days, FastCGI is never used directly. Just like `mod_python`, it is only used for the deployment of WSGI applications.

Setting up FastCGI

Each web server requires a specific module.

- Apache has both mod_fastcgi and mod_fcgid. `mod_fastcgi` is the original one, but it has some licensing issues, which is why it is sometimes considered non-free. `mod_fcgid` is a smaller, compatible alternative. One of these modules needs to be loaded by Apache.
- lighttpd ships its own FastCGI module as well as an SCGI module.
- nginx also supports FastCGI.

Once you have installed and configured the module, you can test it with the following WSGI-application:

```
#!/usr/bin/env python
# -*- coding: UTF-8 -*-

from cgi import escape
import sys, os
from flup.server.fcgi import WSGIServer

def app(environ, start_response):
    start_response('200 OK', [('Content-Type', 'text/html')])

    yield '<h1>FastCGI Environment</h1>'
    yield '<table>'
    for k, v in sorted(environ.items()):
        yield '<tr><th>%s</th><td>%s</td></tr>' % (escape(k), escape(v))
    yield '</table>'

WSGIServer(app).run()
```

This is a simple WSGI application, but you need to install flup first, as flup handles the low level FastCGI access.

See also:

There is some documentation on setting up Django with WSGI, most of which can be reused for other WSGI-compliant frameworks and libraries. Only the `manage.py` part has to be changed, the example used here can be used instead. Django does more or less the exact same thing.

1.4 mod_wsgi

mod_wsgi is an attempt to get rid of the low level gateways. Given that FastCGI, SCGI, and mod_python are mostly used to deploy WSGI applications, mod_wsgi was started to directly embed WSGI applications into the Apache web server. mod_wsgi is specifically designed to host WSGI applications. It makes the deployment of WSGI applications much easier than deployment using other low level methods, which need glue code. The downside is that mod_wsgi is limited to the Apache web server; other servers would need their own implementations of mod_wsgi.

mod_wsgi supports two modes: embedded mode, in which it integrates with the Apache process, and daemon mode, which is more FastCGI-like. Unlike FastCGI, mod_wsgi handles the worker-processes by itself, which makes administration easier.

2 Step back: WSGI

WSGI has already been mentioned several times, so it has to be something important. In fact it really is, and now it is time to explain it.

The *Web Server Gateway Interface*, or WSGI for short, is defined in **PEP 333** and is currently the best way to do Python web programming. While it is great for programmers writing frameworks, a normal web developer does not need to get in direct contact with it. When choosing a framework for web development it is a good idea to choose one which supports WSGI.

The big benefit of WSGI is the unification of the application programming interface. When your program is compatible with WSGI – which at the outer level means that the framework you are using has support for WSGI – your program can be deployed via any web server interface for which there are WSGI wrappers. You do not need to care about whether the application user uses mod_python or FastCGI or mod_wsgi – with WSGI your application will work on any gateway interface. The Python standard library contains its own WSGI server, `wsgiref`, which is a small web server that can be used for testing.

A really great WSGI feature is middleware. Middleware is a layer around your program which can add various functionality to it. There is quite a bit of middleware already available. For example, instead of writing your own session management (HTTP is a stateless protocol, so to associate multiple HTTP requests with a single user your application must create and manage such state via a session), you can just download middleware which does that, plug it in, and get on with coding the unique parts of your application. The same thing with compression – there is existing middleware which handles compressing your HTML using gzip to save on your server's bandwidth. Authentication is another problem that is easily solved using existing middleware.

Although WSGI may seem complex, the initial phase of learning can be very rewarding because WSGI and the associated middleware already have solutions to many problems that might arise while developing web sites.

2.1 WSGI Servers

The code that is used to connect to various low level gateways like CGI or mod_python is called a *WSGI server*. One of these servers is `flup`, which supports FastCGI and SCGI, as well as AJP. Some of these servers are written in Python, as `flup` is, but there also exist others which are written in C and can be used as drop-in replacements.

There are many servers already available, so a Python web application can be deployed nearly anywhere. This is one big advantage that Python has compared with other web technologies.

See also:

A good overview of WSGI-related code can be found in the WSGI homepage, which contains an extensive list of WSGI servers which can be used by *any* application supporting WSGI.

You might be interested in some WSGI-supporting modules already contained in the standard library, namely:

- `wsgiref` – some tiny utilities and servers for WSGI

2.2 Case study: MoinMoin

What does WSGI give the web application developer? Let's take a look at an application that's been around for a while, which was written in Python without using WSGI.

One of the most widely used wiki software packages is MoinMoin. It was created in 2000, so it predates WSGI by about three years. Older versions needed separate code to run on CGI, mod_python, FastCGI and standalone.

It now includes support for WSGI. Using WSGI, it is possible to deploy MoinMoin on any WSGI compliant server, with no additional glue code. Unlike the pre-WSGI versions, this could include WSGI servers that the authors of MoinMoin know nothing about.

3 Model-View-Controller

The term *MVC* is often encountered in statements such as "framework *foo* supports MVC". MVC is more about the overall organization of code, rather than any particular API. Many web frameworks use this model to help the developer bring structure to their program. Bigger web applications can have lots of code, so it is a good idea to have an effective structure right from the beginning. That way, even users of other frameworks (or even other languages, since MVC is not Python-specific) can easily understand the code, given that they are already familiar with the MVC structure.

MVC stands for three components:

- The *model*. This is the data that will be displayed and modified. In Python frameworks, this component is often represented by the classes used by an object-relational mapper.
- The *view*. This component's job is to display the data of the model to the user. Typically this component is implemented via templates.
- The *controller*. This is the layer between the user and the model. The controller reacts to user actions (like opening some specific URL), tells the model to modify the data if necessary, and tells the view code what to display,

While one might think that MVC is a complex design pattern, in fact it is not. It is used in Python because it has turned out to be useful for creating clean, maintainable web sites.

Note: While not all Python frameworks explicitly support MVC, it is often trivial to create a web site which uses the MVC pattern by separating the data logic (the model) from the user interaction logic (the controller) and the templates (the view). That's why it is important not to write unnecessary Python code in the templates – it works against the MVC model and creates chaos in the code base, making it harder to understand and modify.

See also:

The English Wikipedia has an article about the Model-View-Controller pattern. It includes a long list of web frameworks for various programming languages.

4 Ingredients for Websites

Websites are complex constructs, so tools have been created to help web developers make their code easier to write and more maintainable. Tools like these exist for all web frameworks in all languages. Developers are not forced to use these tools, and often there is no "best" tool. It is worth learning about the available tools because they can greatly simplify the process of developing a web site.

See also:

There are far more components than can be presented here. The Python wiki has a page about these components, called Web Components.

4.1 Templates

Mixing of HTML and Python code is made possible by a few libraries. While convenient at first, it leads to horribly unmaintainable code. That's why templates exist. Templates are, in the simplest case, just HTML files with placeholders. The HTML is sent to the user's browser after filling in the placeholders.

Python already includes two ways to build simple templates:

```
>>> template = "<html><body><h1>Hello %s!</h1></body></html>"
>>> print template % "Reader"
<html><body><h1>Hello Reader!</h1></body></html>

>>> from string import Template
>>> template = Template("<html><body><h1>Hello ${name}</h1></body></html>")
>>> print template.substitute(dict(name='Dinsdale'))
<html><body><h1>Hello Dinsdale!</h1></body></html>
```

To generate complex HTML based on non-trivial model data, conditional and looping constructs like Python's *for* and *if* are generally needed. *Template engines* support templates of this complexity.

There are a lot of template engines available for Python which can be used with or without a *framework*. Some of these define a plain-text programming language which is easy to learn, partly because it is limited in scope. Others use XML, and the template output is guaranteed to be always be valid XML. There are many other variations.

Some *frameworks* ship their own template engine or recommend one in particular. In the absence of a reason to use a different template engine, using the one provided by or recommended by the framework is a good idea.

Popular template engines include:

- Mako
- Genshi
- Jinja

See also:

There are many template engines competing for attention, because it is pretty easy to create them in Python. The page Templating in the wiki lists a big, ever-growing number of these. The three listed above are considered "second generation" template engines and are a good place to start.

4.2 Data persistence

Data persistence, while sounding very complicated, is just about storing data. This data might be the text of blog entries, the postings on a bulletin board or the text of a wiki page. There are, of course, a number

of different ways to store information on a web server.

Often, relational database engines like MySQL or PostgreSQL are used because of their good performance when handling very large databases consisting of millions of entries. There is also a small database engine called SQLite, which is bundled with Python in the `sqlite3` module, and which uses only one file. It has no other dependencies. For smaller sites SQLite is just enough.

Relational databases are *queried* using a language called SQL. Python programmers in general do not like SQL too much, as they prefer to work with objects. It is possible to save Python objects into a database using a technology called ORM (Object Relational Mapping). ORM translates all object-oriented access into SQL code under the hood, so the developer does not need to think about it. Most *frameworks* use ORMs, and it works quite well.

A second possibility is storing data in normal, plain text files (some times called "flat files"). This is very easy for simple sites, but can be difficult to get right if the web site is performing many updates to the stored data.

A third possibility are object oriented databases (also called "object databases"). These databases store the object data in a form that closely parallels the way the objects are structured in memory during program execution. (By contrast, ORMs store the object data as rows of data in tables and relations between those rows.) Storing the objects directly has the advantage that nearly all objects can be saved in a straightforward way, unlike in relational databases where some objects are very hard to represent.

Frameworks often give hints on which data storage method to choose. It is usually a good idea to stick to the data store recommended by the framework unless the application has special requirements better satisfied by an alternate storage mechanism.

See also:

- Persistence Tools lists possibilities on how to save data in the file system. Some of these modules are part of the standard library
- Database Programming helps with choosing a method for saving data
- SQLAlchemy, the most powerful OR-Mapper for Python, and Elixir, which makes SQLAlchemy easier to use
- SQLObject, another popular OR-Mapper
- ZODB and Durus, two object oriented databases

5 Frameworks

The process of creating code to run web sites involves writing code to provide various services. The code to provide a particular service often works the same way regardless of the complexity or purpose of the web site in question. Abstracting these common solutions into reusable code produces what are called "frameworks" for web development. Perhaps the most well-known framework for web development is Ruby on Rails, but Python has its own frameworks. Some of these were partly inspired by Rails, or borrowed ideas from Rails, but many existed a long time before Rails.

Originally Python web frameworks tended to incorporate all of the services needed to develop web sites as a giant, integrated set of tools. No two web frameworks were interoperable: a program developed for one could not be deployed on a different one without considerable re-engineering work. This led to the development of "minimalist" web frameworks that provided just the tools to communicate between the Python code and the http protocol, with all other services to be added on top via separate components. Some ad hoc standards were developed that allowed for limited interoperability between frameworks, such as a standard that allowed different template engines to be used interchangeably.

Since the advent of WSGI, the Python web framework world has been evolving toward interoperability based on the WSGI standard. Now many web frameworks, whether "full stack" (providing all the tools one needs

to deploy the most complex web sites) or minimalist, or anything in between, are built from collections of reusable components that can be used with more than one framework.

The majority of users will probably want to select a "full stack" framework that has an active community. These frameworks tend to be well documented, and provide the easiest path to producing a fully functional web site in minimal time.

5.1 Some notable frameworks

There are an incredible number of frameworks, so they cannot all be covered here. Instead we will briefly touch on some of the most popular.

Django

Django is a framework consisting of several tightly coupled elements which were written from scratch and work together very well. It includes an ORM which is quite powerful while being simple to use, and has a great online administration interface which makes it possible to edit the data in the database with a browser. The template engine is text-based and is designed to be usable for page designers who cannot write Python. It supports template inheritance and filters (which work like Unix pipes). Django has many handy features bundled, such as creation of RSS feeds or generic views, which make it possible to create web sites almost without writing any Python code.

It has a big, international community, the members of which have created many web sites. There are also a lot of add-on projects which extend Django's normal functionality. This is partly due to Django's well written online documentation and the Django book.

Note: Although Django is an MVC-style framework, it names the elements differently, which is described in the Django FAQ.

TurboGears

Another popular web framework for Python is TurboGears. TurboGears takes the approach of using already existing components and combining them with glue code to create a seamless experience. TurboGears gives the user flexibility in choosing components. For example the ORM and template engine can be changed to use packages different from those used by default.

The documentation can be found in the TurboGears documentation, where links to screencasts can be found. TurboGears has also an active user community which can respond to most related questions. There is also a TurboGears book published, which is a good starting point.

The newest version of TurboGears, version 2.0, moves even further in direction of WSGI support and a component-based architecture. TurboGears 2 is based on the WSGI stack of another popular component-based web framework, Pylons.

Zope

The Zope framework is one of the "old original" frameworks. Its current incarnation in Zope2 is a tightly integrated full-stack framework. One of its most interesting feature is its tight integration with a powerful object database called the ZODB (Zope Object Database). Because of its highly integrated nature, Zope wound up in a somewhat isolated ecosystem: code written for Zope wasn't very usable outside of Zope, and vice-versa. To solve this problem the Zope 3 effort was started. Zope 3 re-engineers Zope as a set of more cleanly isolated components. This effort was started before the advent of the WSGI standard, but there is WSGI support for Zope 3 from the Repoze project. Zope components have many years of production use

behind them, and the Zope 3 project gives access to these components to the wider Python community. There is even a separate framework based on the Zope components: Grok.

Zope is also the infrastructure used by the Plone content management system, one of the most powerful and popular content management systems available.

Other notable frameworks

Of course these are not the only frameworks that are available. There are many other frameworks worth mentioning.

Another framework that's already been mentioned is Pylons. Pylons is much like TurboGears, but with an even stronger emphasis on flexibility, which comes at the cost of being more difficult to use. Nearly every component can be exchanged, which makes it necessary to use the documentation of every single component, of which there are many. Pylons builds upon Paste, an extensive set of tools which are handy for WSGI.

And that's still not everything. The most up-to-date information can always be found in the Python wiki.

See also:

The Python wiki contains an extensive list of web frameworks.

Most frameworks also have their own mailing lists and IRC channels, look out for these on the projects' web sites.

Index

P