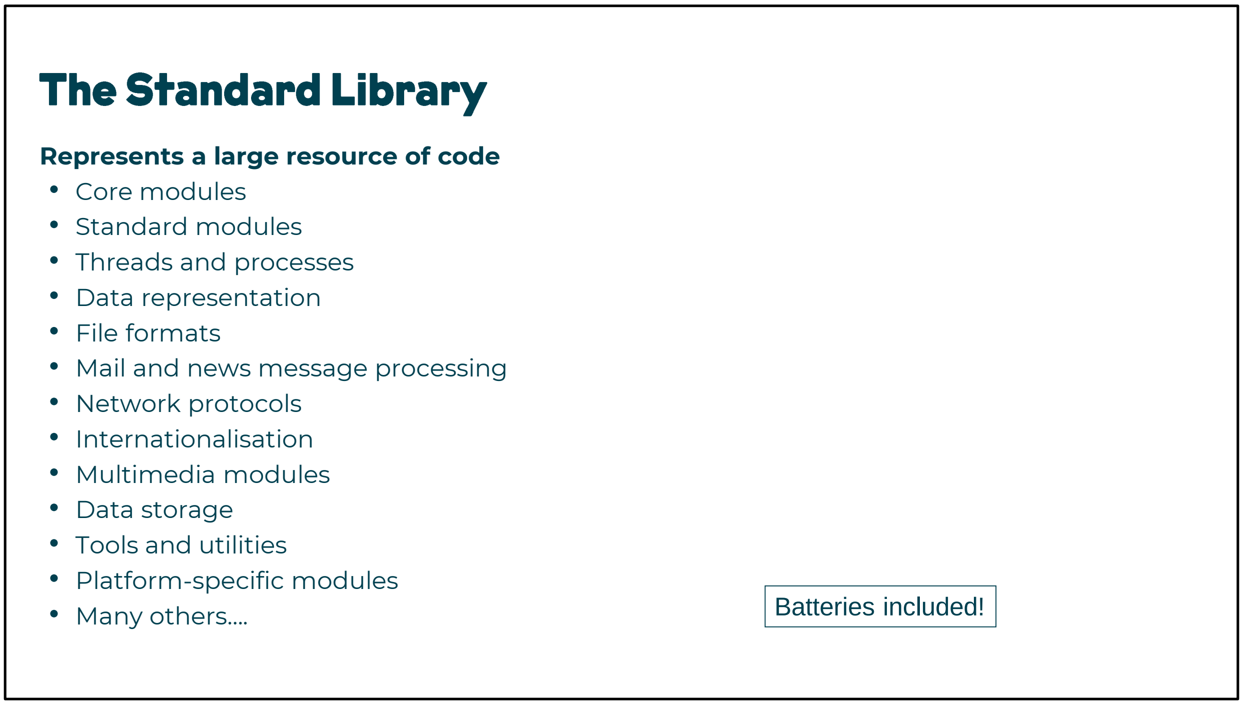
POWERING POTENTIAL

Python 3 Programming

The Python Standard Library

THE PYTHON STANDARD LIBRARY

Contents



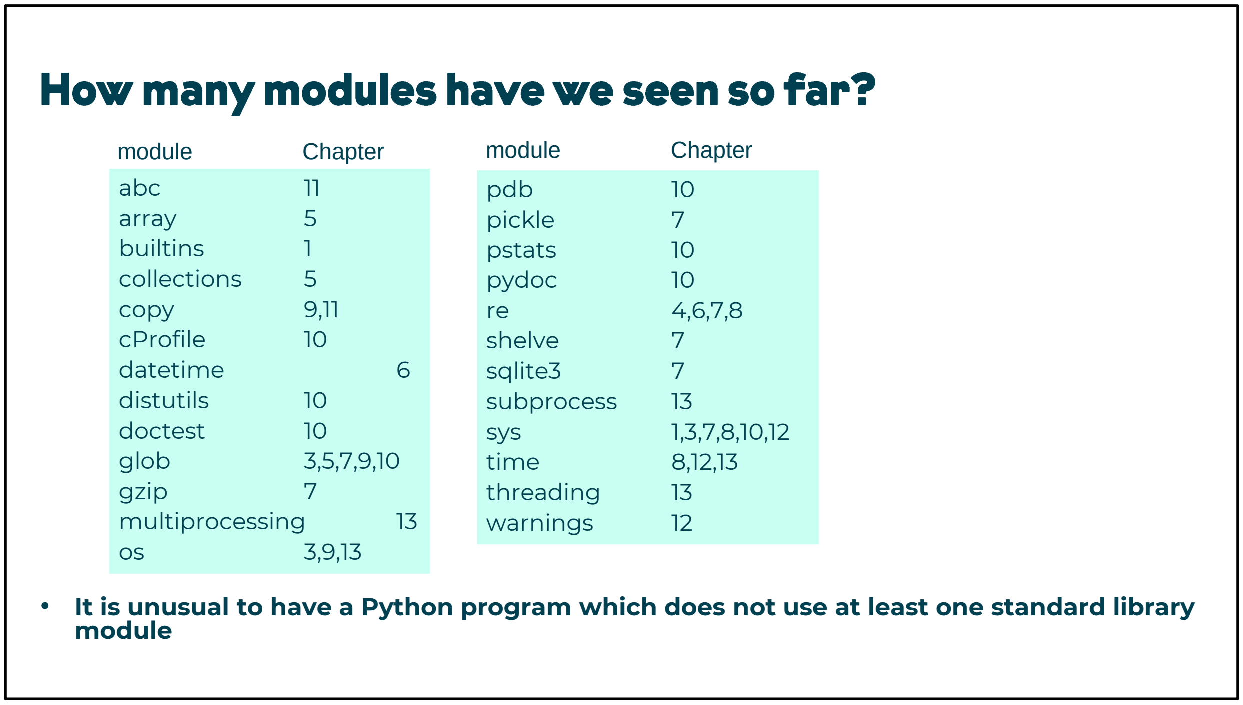
The Standard Library

Represents a large resource of code

- Core modules
- Standard modules
- Threads and processes
- Data representation
- File formats
- Mail and news message processing
- Network protocols
- Internationalisation
- Multimedia modules
- Data storage
- Tools and utilities
- Platform-specific modules
- Many others....

Batteries included!

There are a large number of modules (around 200) in the standard library, far more than you could reasonably expect, and they are all there and available as soon as you install Python. As you can see from the list, they cover a huge range of needs - usually more than we will ever use.
If we described each module, and each function within it, then this chapter would be very long and very boring. Anyway, we have the online documentation for that.
Instead, we are going to show you some of the commonly used functions, the kind of thing you are likely to need at once. As for the rest: RTFM (Read The Fine Manual).



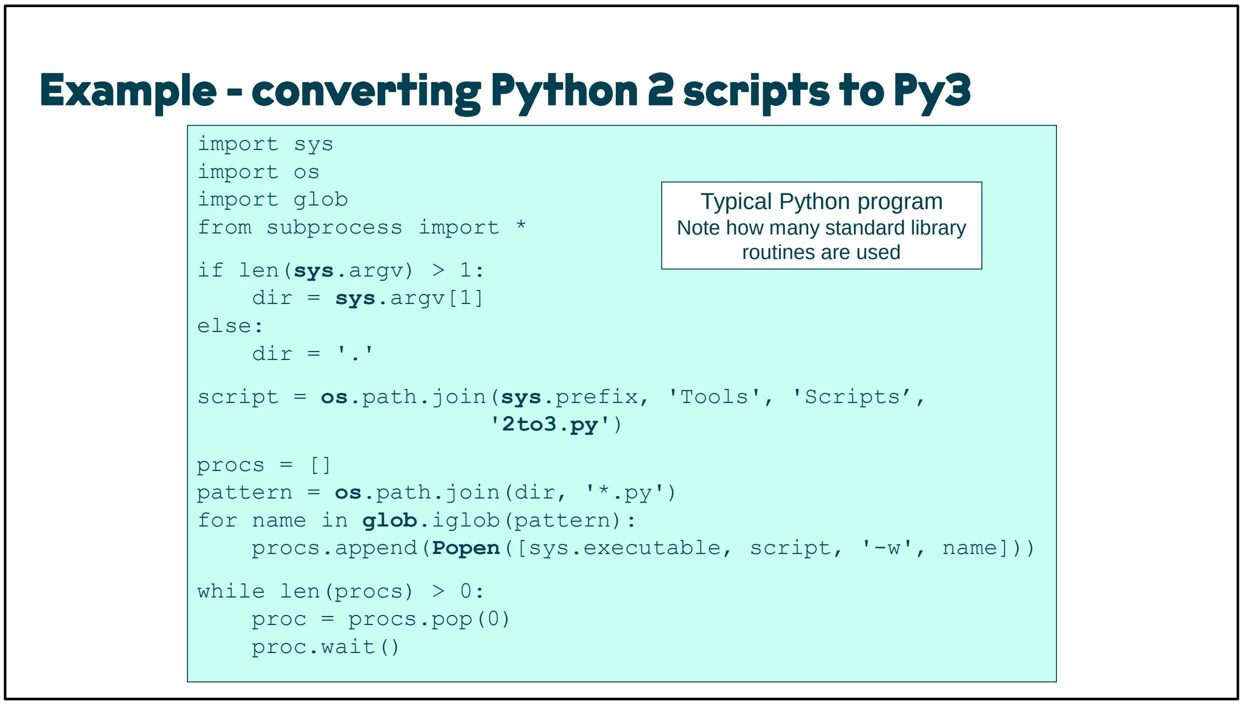
How many modules have we seen so far?

module	Chapter
abc	11
array	5
builtins	1
collections	5
copy	9,11
cProfile	10
datetime	6
distutils	10
doctest	10
glob	3,5,7,9,10
gzip	7
multiprocessing	13
os	3,9,13

module	Chapter
pdb	10
pickle	7
pstats	10
pydoc	10
re	4,6,7,8
shelve	7
sqlite3	7
subprocess	13
sys	1,3,7,8,10,12
time	8,12,13
threading	13
warnings	12

- **It is unusual to have a Python program which does not use at least one standard library module**

We have seen quite a few modules from the standard library already, although some were on slides after a chapter summary (like `abc`). We also used `tkinter` in the exercise for chapter 1.
We can't really say all that much about Python without mentioning the standard library, they are one of the most important features of the language. So, as a Python programmer, you will be expected to at least have an appreciation of what is available in the standard library, and where to look.
This chapter will not cover examples of every module in the standard library - the book which does that is 1300 pages long! Instead, we are going to show a selection of some of the more commonly used ones. Of course, every site is different, and your installation might have its own favourite modules that are not in our list. In fact, we would be surprised if that was not the case. Learn a few, then learn each one as you need it. You can't learn them all in one go and there would be little point in doing so, since it is likely that you will use more than half of them.



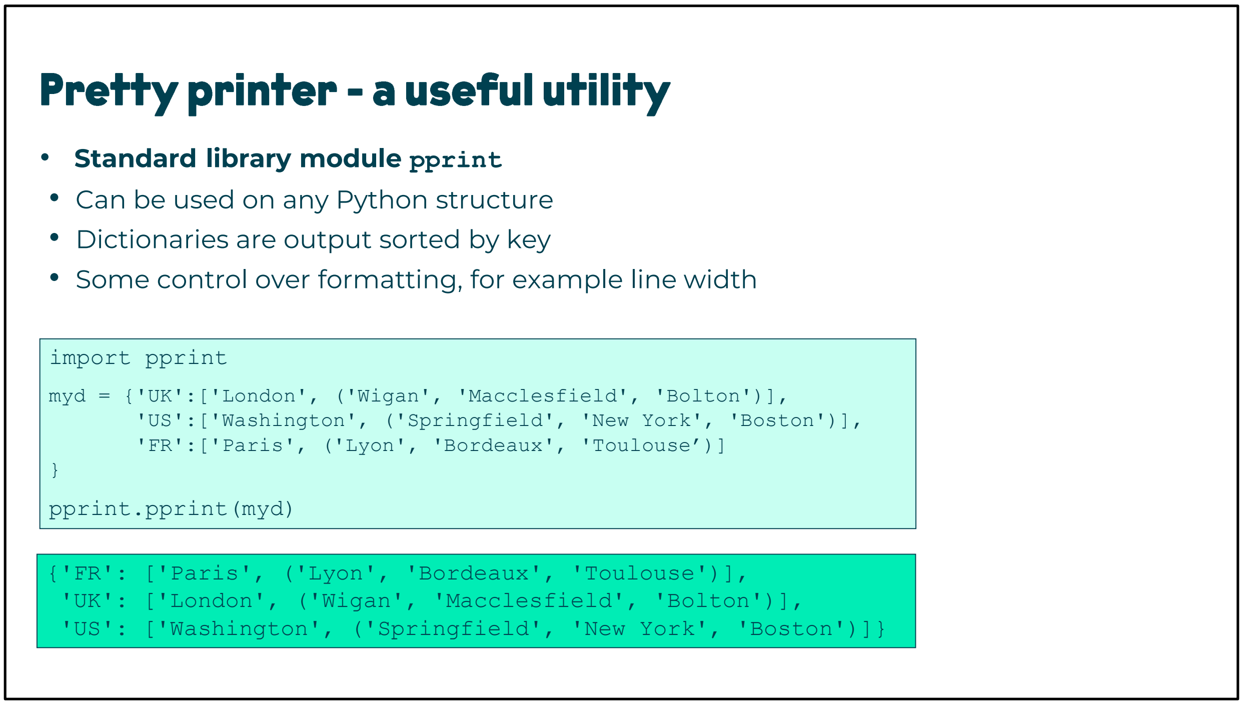
Example - converting Python 2 scripts to Py3

```
import sys
import os
import glob
from subprocess import *

if len(sys.argv) > 1:
    dir = sys.argv[1]
else:
    dir = '.'

script = os.path.join(sys.prefix, 'Tools', 'Scripts’,
                     '2to3.py')

procs = []
pattern = os.path.join(dir, '*.py')
for name in glob.iglob(pattern):
    procs.append(Popen([sys.executable, script, '-w', name]))

while len(procs) > 0:
    proc = procs.pop(0)
    proc.wait()
```

Typical Python program
Note how many standard library routines are used

The **`2to3`** script is supplied with Python 3. It converts, where possible, Python 2 scripts into Python 3. There is no reliable library function to run it, so it is run as a separate process. In the example code, we are iterating through a directory, running **`2to3.py`** on each python file. This fairly simple task requires the use of several of the standard library functions, and is typical of a general Python program.
First we get the first command-line argument, using **`sys.argv`**, which is an array of command line arguments (the first element is the name of the program).
The **`2to3.py`** script resides in a specific Python 3 folder, and constructing the full filename takes several steps. We are using **`sys.prefix`** to get the Python system base directory name, then we join all the elements together using **`os.path.join`**, which ensures that the correct directory separator for the current operating system is used.
The next step is to get the filenames of the Python scripts to be converted. Again, we are using **`os.path.join`**, but this time to create a *glob construct* - sometimes known as *wildcards*, or *filename expansion*. The actual filenames are generated by **`glob.iglob`**. The '**`i`**' indicates that it can be used as an iterator, which means we get just one filename at a time. We could have

used `glob.glob` instead, since it returns a list of filename, but we might exceed the limit of 255 command-line arguments most operating systems). Instead, we play it safe and run one process for each script.
Notice we are using **`subprocess.Popen`**, but not immediately waiting for each one. We start each process, storing the process object in an array. Once they have all been started, we go back and start waiting on the first one, which has probably ended by this point. There are other patterns we can use here.



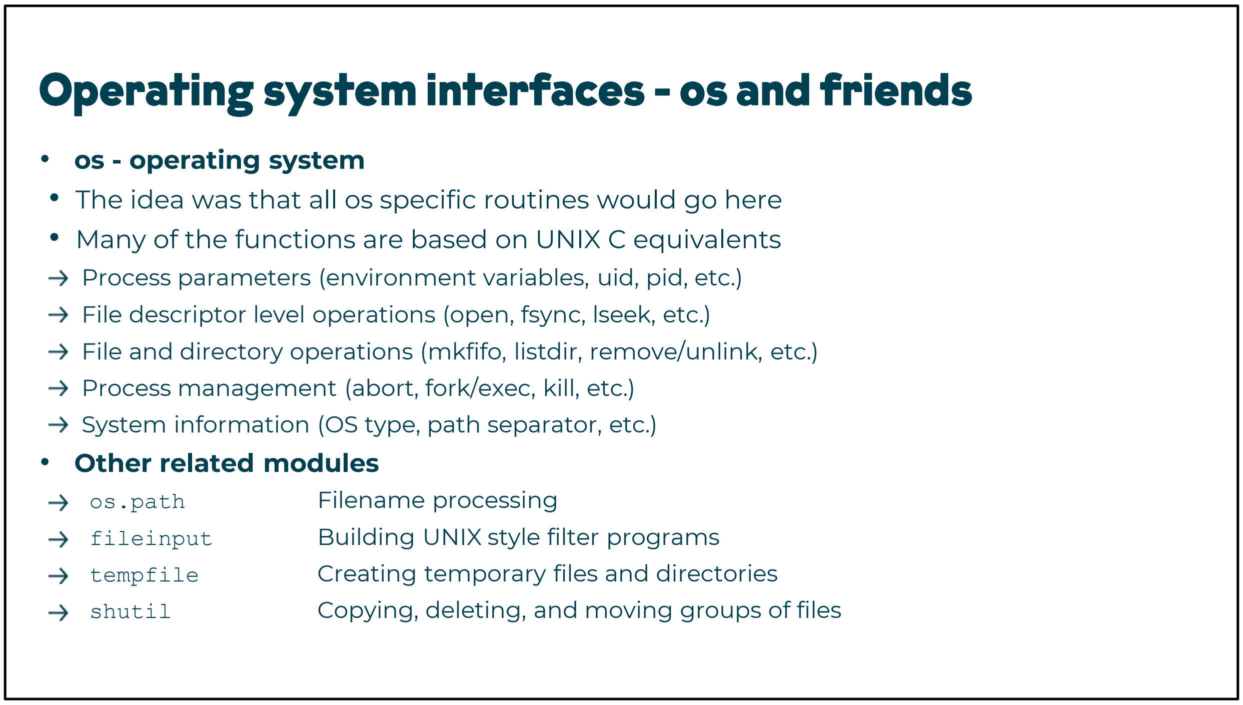
Pretty printer - a useful utility

- **Standard library module `pprint`**
- Can be used on any Python structure
- Dictionaries are output sorted by key
- Some control over formatting, for example line width

```
import pprint
myd = {'UK':['London', ('Wigan', 'Macclesfield', 'Bolton')],
       'US':['Washington', ('Springfield', 'New York', 'Boston')],
       'FR':['Paris', ('Lyon', 'Bordeaux', 'Toulouse’)]
}
pprint.pprint(myd)
```

```
{'FR': ['Paris', ('Lyon', 'Bordeaux', 'Toulouse')],
 'UK': ['London', ('Wigan', 'Macclesfield', 'Bolton')],
 'US': ['Washington', ('Springfield', 'New York', 'Boston')]}
```

The pretty printer module, **`pprint`**, is useful when displaying the values of containers, which is often for debugging.
The output from `print()` using the example on the slide would be:
```
{'FR': ['Paris', ('Lyon', 'Bordeaux', 'Toulouse')],
'US': ['Washington', ('Springfield', 'New York',
'Boston')],'UK': ['London', ('Wigan', 'Macclesfield',
'Bolton')]}
```
which is not so easy to read - and that is with a simple structure and only three keys.
Control over indentation, width, depth, and IO stream is available by creating a **`pprint.PrettyPrinter`** object with these attributes, then calling methods on that object.
See the Python Standard Library documentation for further details.



Operating system interfaces - os and friends

- **os - operating system**
- The idea was that all os specific routines would go here
- Many of the functions are based on UNIX C equivalents
  - → Process parameters (environment variables, uid, pid, etc.)
  - → File descriptor level operations (open, fsync, lseek, etc.)
  - → File and directory operations (mkfifo, listdir, remove/unlink, etc.)
  - → Process management (abort, fork/exec, kill, etc.)
  - → System information (OS type, path separator, etc.)
- **Other related modules**

→ `os.path`	Filename processing
→ `fileinput`	Building UNIX style filter programs
→ `tempfile`	Creating temporary files and directories
→ `shutil`	Copying, deleting, and moving groups of files

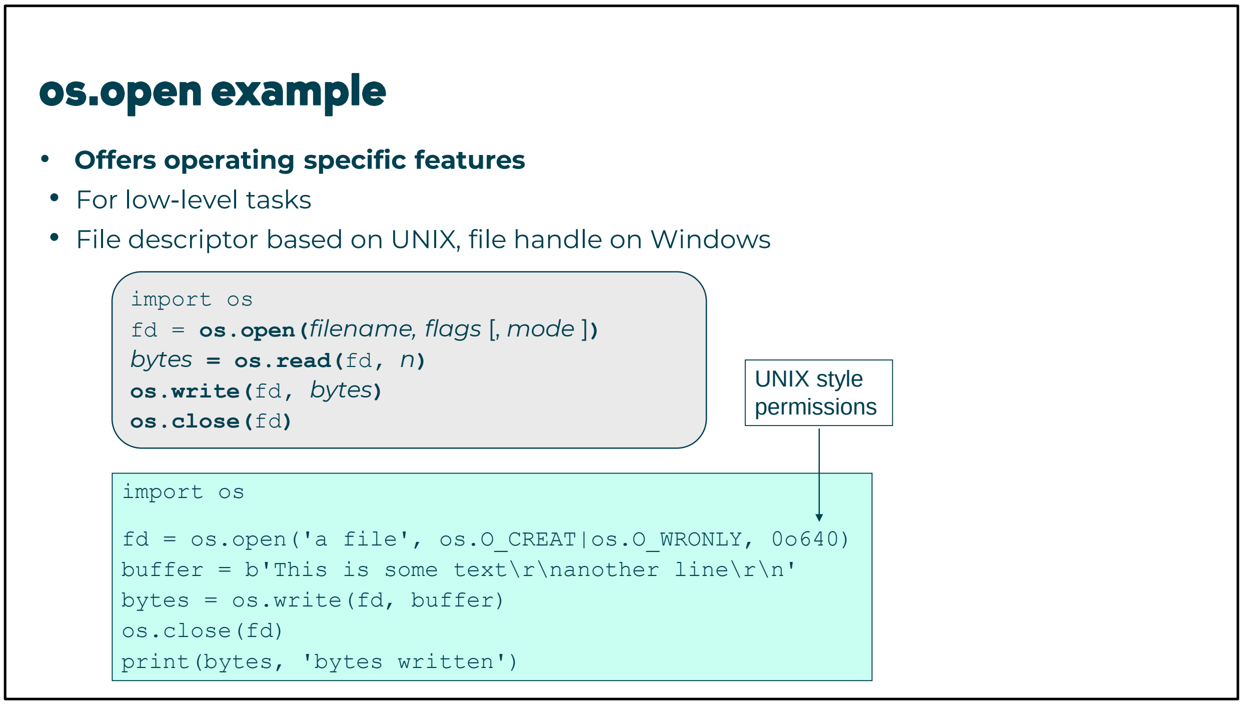
In practice, there are `os` specifics all over, just look at the **`subprocess`** module, for example. Many of the functions in this module are operating system specific, or not implemented across all installations. For example, **`getppid`** (get parent process id) is easily implemented on UNIX, but takes a lot more effort on Windows, so is not supplied. On the other hand, Windows has **`os.startfile()`**, which runs the program associated with the specified file: just as if you has double-clicked on the file from Windows Explorer.
The **`os`** module is one of those modules you will probably use every day, so it is worth browsing through the help text for it to get an idea of what is available.

os.scandir example

- **Added at Python 3.5**
  - More efficient than the older methods
- **The `stat()` method returns a `stat_result` object**
  - as `os.stat()`

```
a.py            598
bstr.py         244
glb.py          105
restuff.py      486
stuff           directory
```

```
import os
path = '.'
for entry in os.scandir(path):
   stat = entry.stat()
   if entry.is_file():
       print("%-16s %d" % (entry.name, stat.st_size))
   elif entry.is_dir():
       print("%-16s directory" % (entry.name))
```



os.open example

- **Offers operating specific features**
  - For low-level tasks
  - File descriptor based on UNIX, file handle on Windows

```
import os
fd = os.open(filename, flags [, mode ])
bytes = os.read(fd, n)
os.write(fd, bytes)
os.close(fd)
```

UNIX style permissions

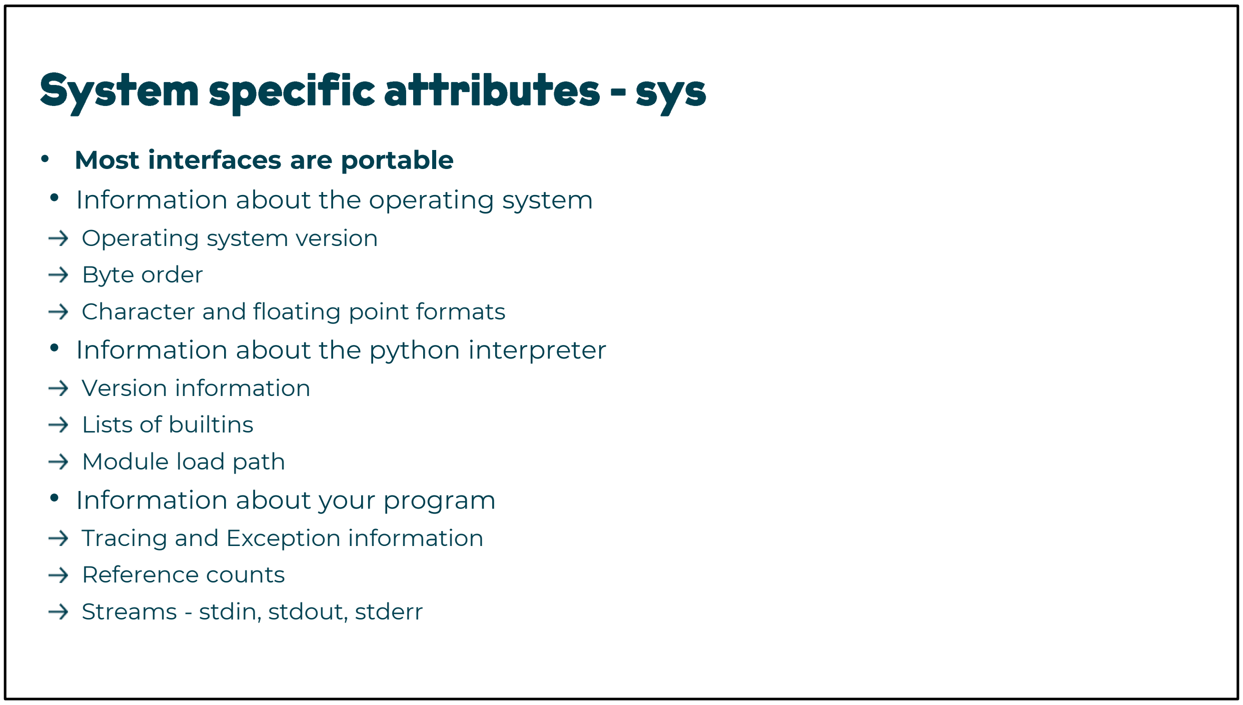
```
import os

fd = os.open('a file', os.O_CREAT|os.O_WRONLY, 0o640)
buffer = b'This is some text\r\nanother line\r\n'
bytes = os.write(fd, buffer)
os.close(fd)
print(bytes, 'bytes written')
```

Anyone coming from a UNIX C background will recognise this - but to others this does not really fit into the culture. These interfaces are designed for low-level access, where complete control is required.
The documentation for os.read() and os.write() specifically states that strings are used for IO, but this is out of date. In Python 3, strings are Unicode, and these interfaces only support the byte type.
In os.open, flags may be:

O_RDONLY	Open file for reading
O_WRONLY	Open file for writing
O_RDWR	Open file for read/write
O_APPEND	Append to the end of the file
O_CREAT	Create file if it doesn't exit
O_NONBLOCK	Don't block on open, read, or write
O_TRUNC	Truncate to zero length
O_TEXT	Text mode (Windows)
O_BINARY	Binary mode (Windows)

Notice how these flags are (binary) OR'ed together, a common practice with bit-masks such as these.

The os module contains many other low-level operations.



System specific attributes - sys

- **Most interfaces are portable**
- Information about the operating system
  - → Operating system version
  - → Byte order
  - → Character and floating point formats
- Information about the python interpreter
  - → Version information
  - → Lists of builtins
  - → Module load path
- Information about your program
  - → Tracing and Exception information
  - → Reference counts
  - → Streams - stdin, stdout, stderr

Whereas, the os module provides mostly low-level interfaces into the operating system, the **`sys`** module gives (and sets) attributes about the environment in which your program is running. These two modules do appear to overlap at first, but you soon get used to what is where. There are a few system specific interfaces, for example **`winver`**, but most are portable. Attributes also vary between releases, for example 3.1 introduced **`sys.int_info`**, which shows the size of an internal integer.

Filter programs - fileinput module

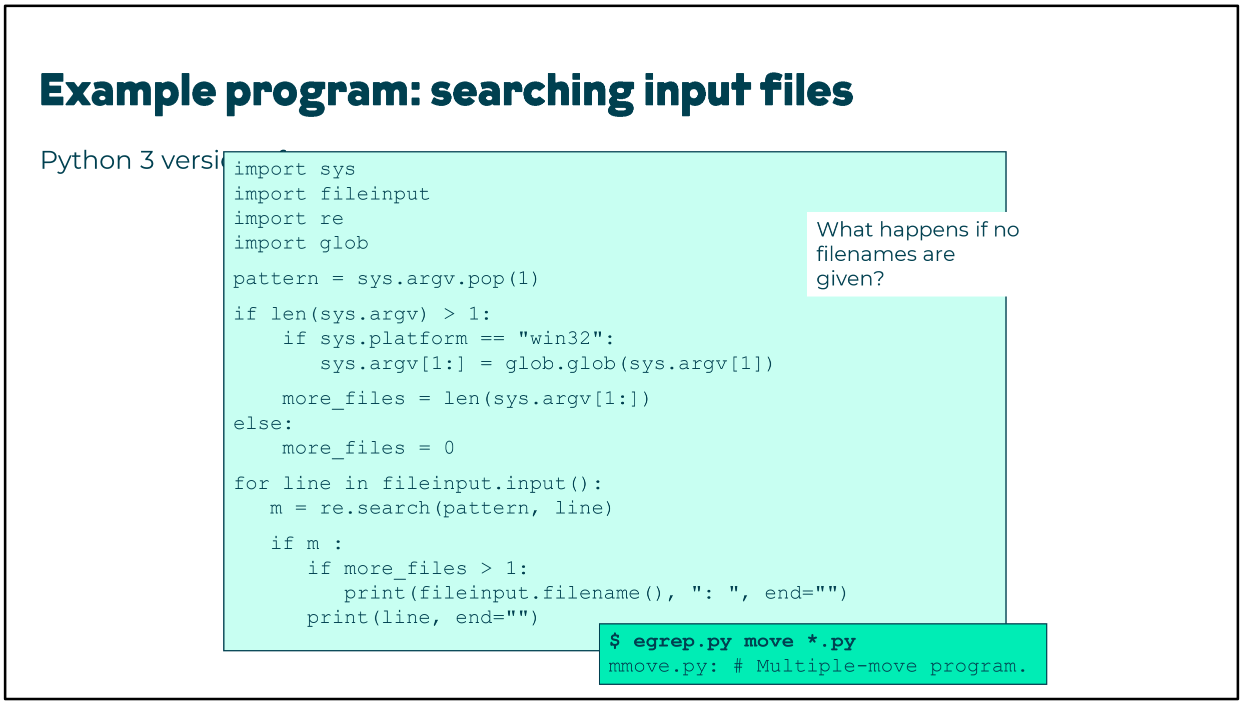
The typical behaviour of filter programs is:

- Command line arguments are names of files to process
- If no arguments are given, read standard input instead
- A hyphen on the command line indicates standard input as well
- Examples are `grep`, `sed`, `awk`, `wc`, `lpr`, …

The fileinput module makes it easy to create Python filter programs

- **`sys.argv`** is used for the list of filenames (omitting first element)
- `fileinput.input` gives each line of the files in turn
- `fileinput.filename` gives the current filename being read
- `fileinput.filelineno` gives line number in the current file
- and more

Filter programs are very common on the UNIX command-line. Programs like grep(1), cat(1), head(1), tail(1), sed(1) are examples which most UNIX shell users will be familiar with. What these programs have in common is that they are run, specifying a list of filenames, as their main arguments (not including options). Filenames are processed from left to right. If no filename is supplied then standard input (stdin) is read, making them suitable to be used in pipelines.
The Python standard library includes **`fileinput`**, which gives us similar behaviour to create our own filter programs in Python. It is actually a little more powerful, in that the filenames do not actually need to come from the command line - they come from `sys.argv`, which is a normal mutable list and so can be manipulated in the program itself.
The module includes a comprehensive set of attributes and methods, only the most common are shown. See the Python documentation for a complete list. An example follows.



Example program: searching input files

Python 3 versi

```
import sys
import fileinput
import re
import glob

pattern = sys.argv.pop(1)

if len(sys.argv) > 1:
    if sys.platform == "win32":
       sys.argv[1:] = glob.glob(sys.argv[1])

    more_files = len(sys.argv[1:])
else:
    more_files = 0

for line in fileinput.input():
   m = re.search(pattern, line)

   if m :
      if more_files > 1:
         print(fileinput.filename(), ": ", end="")
      print(line, end="")
```

What happens if no filenames are given?

```
$ egrep.py move *.py
mmove.py: # Multiple-move program.
```

This small example program emulates the UNIX egrep program (not grep, since Python uses Extended REs). It shows how regular expressions are usually applied for filter-like utilities.
Note the empty line terminator after each print statement. This suppresses the newline, which should already be present at the end of each line. Some files might not have a newline at the end of file. While this is usually seen as an error in the file, it can be prudent to allow for the error by either testing the final line or printing an additional newline regardless.
By the way, if you are going to pass Regular Expressions on the command line be sure to quote them (on UNIX use single quotes, on Windows double quotes), otherwise the shell might confuse them for wild-cards.

Command-line option parsing - argparse

Introduced in Python 2.7 and 3.2

- If on an earlier release, use optparse

Steps:

- Create a parser object
- Define arguments

→ Supports short (e.g. `-h`) and long (e.g. `--help`) options

→ Actions include store, store_true, append, count, help, version

→ Type checking, argument counting, mandatory arguments, etc.

→ Mandatory parameters (not options) are also supported

- Parse the arguments

→ By default from sys.argv, but can be from elsewhere

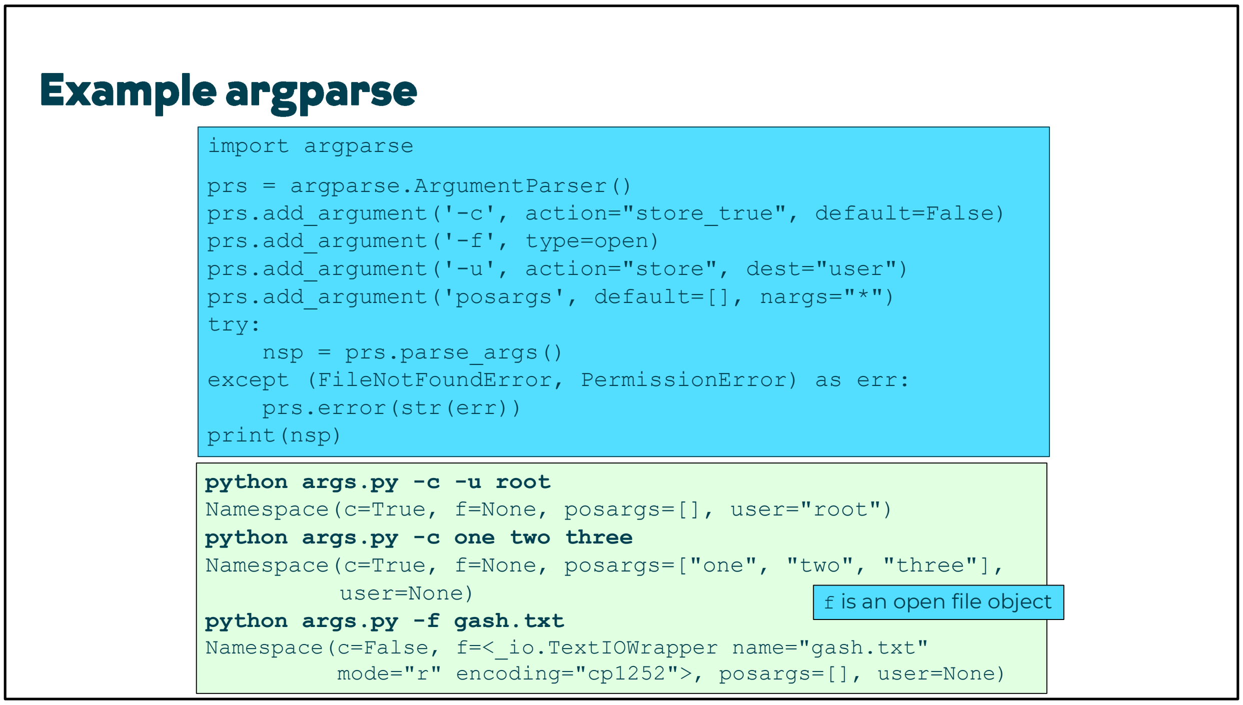
→ Returns a namespace containing arguments

Includes help text and error checking

Many other options, including file handing

The `argparse` module was introduced at Python 2.7 and 3.2. Older modules exist which you might come across: `getopt` (the legacy mechanism) and `optparse` (deprecated). See PEP389 for the justifications for this module over the others.
Options on UNIX type platforms (including Linux and OS X) are normally prefixed with single (short) or double (long) hyphens. Windows options are usually prefixed **/**, which `argparse` supports on that platform. The prefixes can be specified programmatically as well.
Not everything need be options, the module also supports positional parameters which may be stored in a list, and custom callbacks for validation.
This module is extensive, and probably offers more features than you will ever need. We suggest that you use it for simple applications first, and then explore the other features as you need them. We cannot show more than a hand-full of the features here (an example follows). We could easily have an entire chapter on `argparse` by itself!
There is a tutorial as part of the standard library documentation, and a cookbook available at https://mkaz.com/2014/07/26/python-

argparse-cookbook/.



Example argparse

```
import argparse
prs = argparse.ArgumentParser()
prs.add_argument('-c', action="store_true", default=False)
prs.add_argument('-f', type=open)
prs.add_argument('-u', action="store", dest="user")
prs.add_argument('posargs', default=[], nargs="*")
try:
    nsp = prs.parse_args()
except (FileNotFoundError, PermissionError) as err:
    prs.error(str(err))
print(nsp)
```

```
python args.py -c -u root
Namespace(c=True, f=None, posargs=[], user="root")
python args.py -c one two three
Namespace(c=True, f=None, posargs=["one", "two", "three"],
         user=None)
python args.py -f gash.txt
Namespace(c=False, f=<_io.TextIOWrapper name="gash.txt"
         mode="r" encoding="cp1252">, posargs=[], user=None)
```

f is an open file object

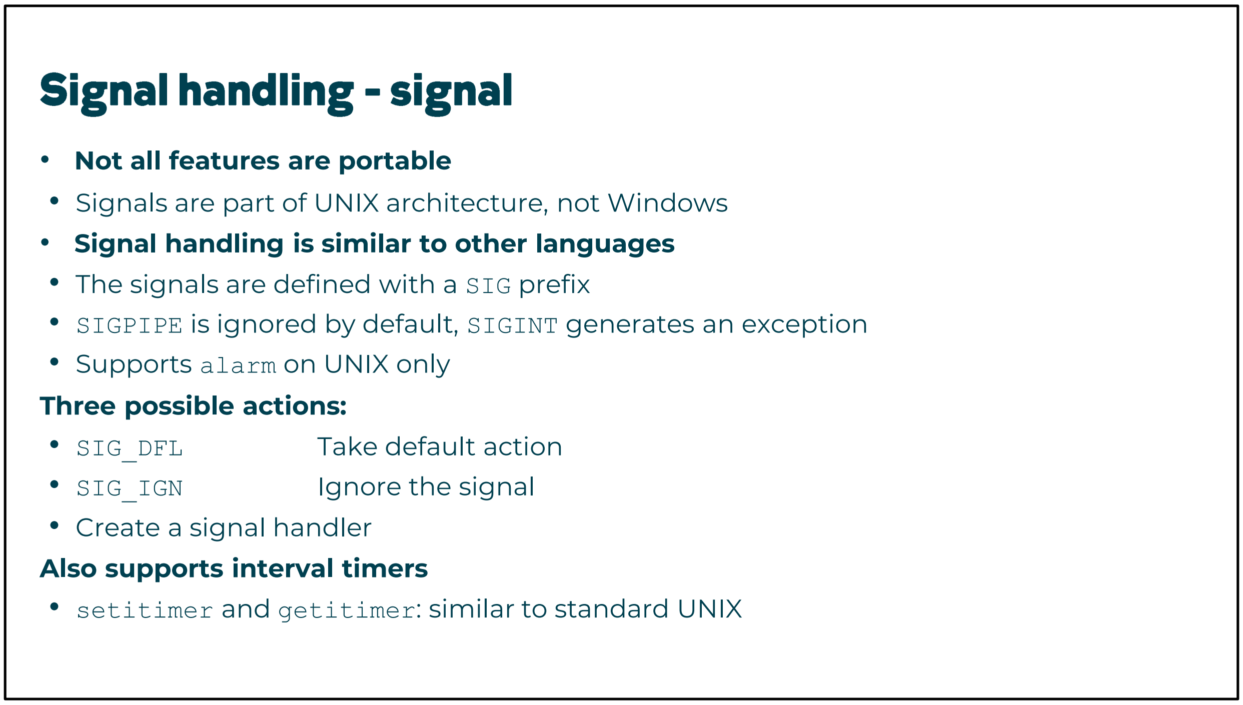
Only short options are shown, long options are handled in the same way. Default help options and text is provided:

```
python args.py --help
usage: args.py [-h] [-c] [-f F] [-u USER]
[posargs[posargs ...]]
positional arguments:
  posargs
optional arguments:
  -h, --help  show this help message and exit
  -c
  -f F
  -u USER
```

In the example, `prs` is the parser object, `nsp` is the Namespace object. Any type can be specified, including built-in types like `int` or `float` (`open` is shown, which returns an `io` object), the default is a string. Files can also be opened for write or append.
For an invalid filename, the error handling would give (for example):

```
python args.py -f xy.txt
usage: args.py [-h] [-c] [-f F] [-u USER]
```

```
     [posargs[posargs ...]]
     args.py: error: [Errno 2] No such file or
     directory: 'xy.txt'
```
The items in the Namespace are accessed as object attributes, for example:
```
     if nsp.f is None: nsp.f = sys.stdin
     for line in nsp.f: print line,
```



Signal handling - signal

- **Not all features are portable**
- Signals are part of UNIX architecture, not Windows
- **Signal handling is similar to other languages**
- The signals are defined with a `SIG` prefix
- `SIGPIPE` is ignored by default, `SIGINT` generates an exception
- Supports `alarm` on UNIX only

Three possible actions:

`SIG_DFL`	Take default action
`SIG_IGN`	Ignore the signal
Create a signal handler	

Also supports interval timers

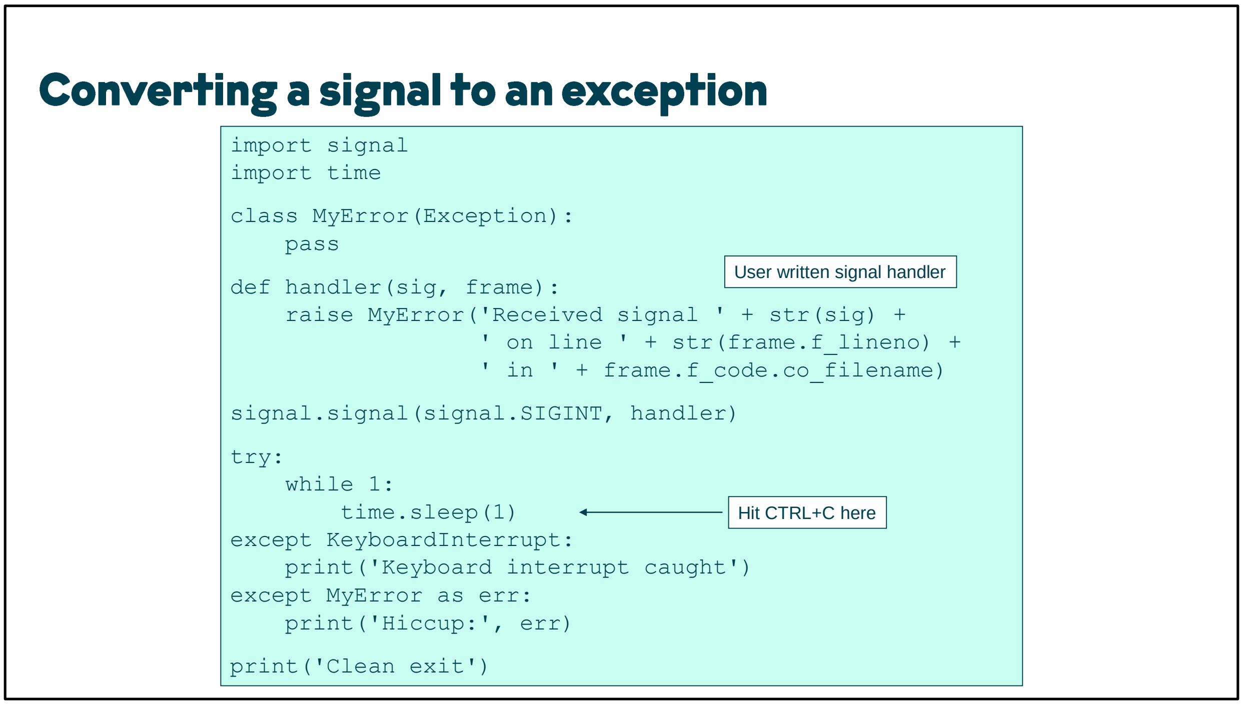
- `setitimer` and `getitimer`: similar to standard UNIX

The standard signal module handles signals in a familiar way. It does not expose the signal mask, but presents an interface similar to the old and deprecated ISO-C interface. Signals are an essential part of UNIX architecture, but do not sit well on Windows, so unless portability is of primary concern, then it is probably best to stick to Win32 objects, such as events, on that OS.
The UNIX architecture supports three possible actions on a signal: take the default action, which is usually to kill the process (SIGCHLD being an important exception); ignore the signal (SIGKILL cannot be ignored); or execute a handler. Handlers are not (*cannot be*) inherited, however ignoring a signal is, so setting:

```
signal.signal(SIGHUP,signal.SIG_IGN)
```

is useful when running background jobs on UNIX.
The **`alarm`** function is supported on UNIX only (on Windows use Waitable Timers) and has similar functionality to the POSIX C routine of the same name.
As in C, signals which interrupt a system call with set `errno.EINTR` and could raise an `IOError`.
In multi-threaded applications, only the main thread can create a signal handler. Signals were designed to communicate with

processes, not threads - threads and signals do not mix well (regardless of the language).

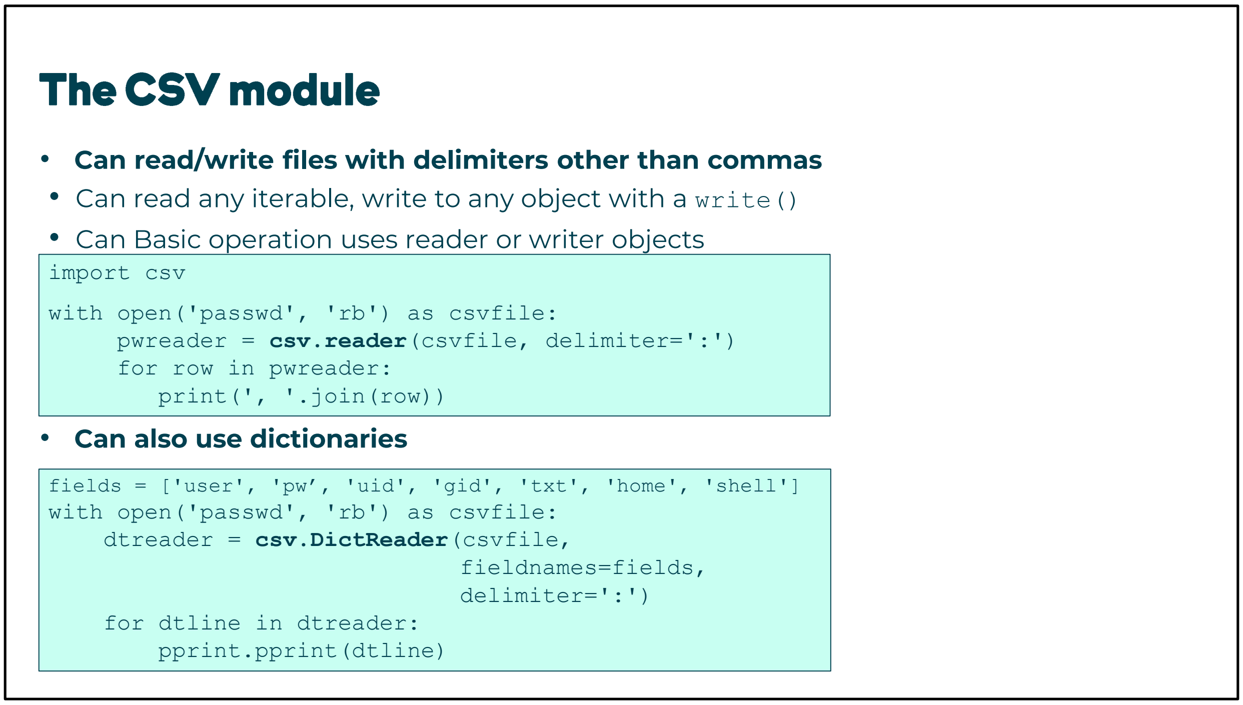


Converting a signal to an exception

```
import signal
import time

class MyError(Exception):
    pass

def handler(sig, frame):
    raise MyError('Received signal ' + str(sig) +
                  ' on line ' + str(frame.f_lineno) +
                  ' in ' + frame.f_code.co_filename)

signal.signal(signal.SIGINT, handler)

try:
    while 1:
        time.sleep(1)
except KeyboardInterrupt:
    print('Keyboard interrupt caught')
except MyError as err:
    print('Hiccup:', err)

print('Clean exit')
```

Setting a signal handler enables us to carry out other actions, which might include raising an exception.
The signal handler (**`handler`** in the example shown) takes two parameters, the signal number (unfortunately, the module does not support `psignal`) and a *frame object*. A frame object is used in trace backs, and contains, among other things, the *code object* (`frame.f_code`). A code object contains just about everything about a chunk of code, we have just used the filename here, but we could inspect variables as well.
Hitting CTRL+C during the `sleep` gives the following:

```
Hiccup: Received signal 2 on line 20 in
C:\QA\notes\sigs.py
Clean exit
```



The CSV module

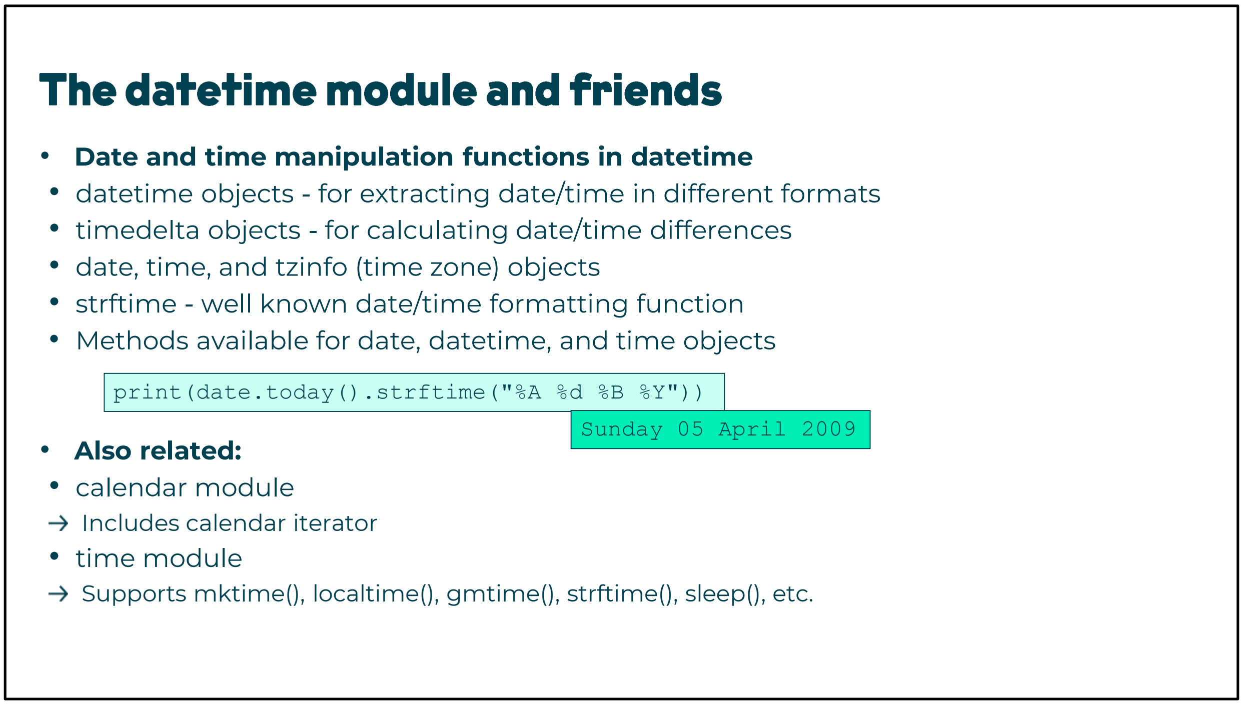
- **Can read/write files with delimiters other than commas**
- Can read any iterable, write to any object with a `write()`
- Can Basic operation uses reader or writer objects

```
import csv

with open('passwd', 'rb') as csvfile:
     pwreader = csv.reader(csvfile, delimiter=':')
     for row in pwreader:
        print(', '.join(row))
```

- **Can also use dictionaries**

```
fields = ['user', 'pw’, 'uid', 'gid', 'txt', 'home', 'shell']
with open('passwd', 'rb') as csvfile:
    dtreader = csv.DictReader(csvfile,
                              fieldnames=fields,
                              delimiter=':')
    for dtline in dtreader:
        pprint.pprint(dtline)
```

Files must be opened as binary.
CSV files come in several dialects, use `csv.list_dialects()` to find those supported, which might only be `['excel-tab', 'excel']`. It is possible to create your own dialect, or let the module figure out its own, using `sniff()`.



The datetime module and friends

- **Date and time manipulation functions in datetime**
  - datetime objects - for extracting date/time in different formats
  - timedelta objects - for calculating date/time differences
  - date, time, and tzinfo (time zone) objects
  - strftime - well known date/time formatting function
  - Methods available for date, datetime, and time objects

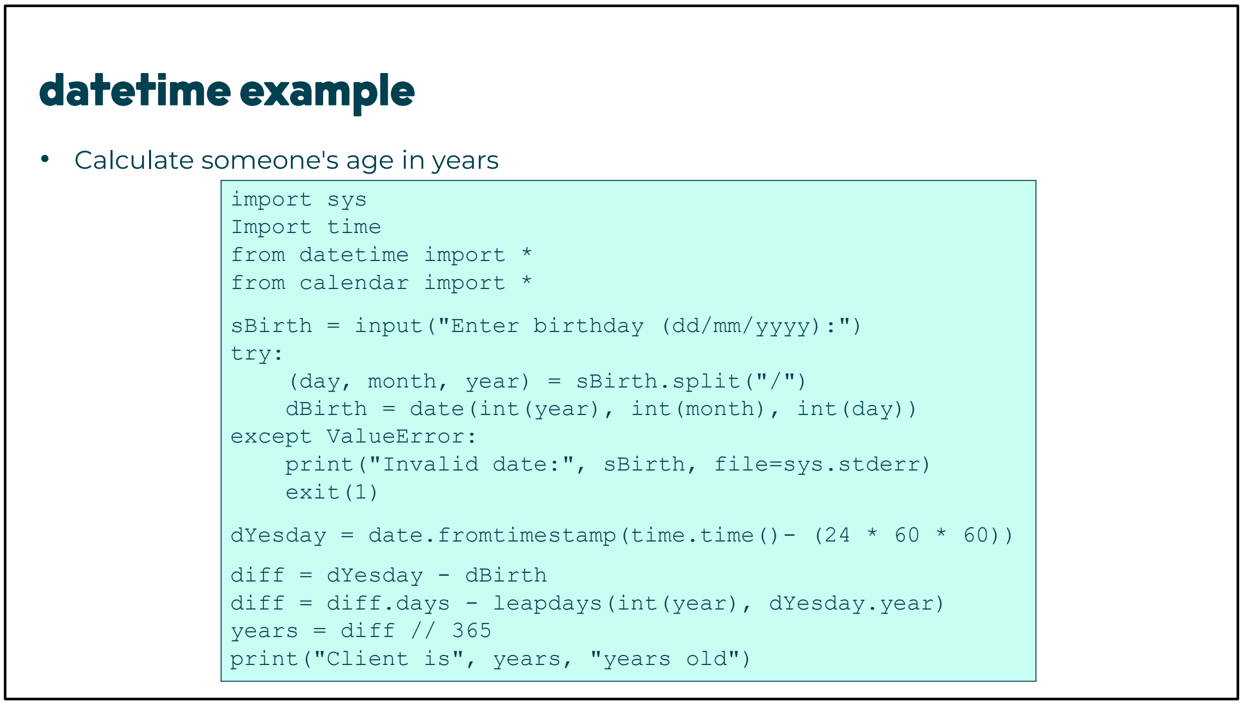
```
print(date.today().strftime("%A %d %B %Y"))
```

```
Sunday 05 April 2009
```

- **Also related:**
  - calendar module
    → Includes calendar iterator
  - time module
    → Supports mktime(), localtime(), gmtime(), strftime(), sleep(), etc.

Date and time manipulation is a common task in many programs, and there are three related modules in the Standard Library which help.
The **`datetime`** module includes most of the operations you may require, including **`timedelta`**, which handles differences between date and time objects. It also includes timestamp handling functions, useful for handling file timestamps, as used by Microsoft's NTFS and most UNIX file systems.
The calendar module is useful for stepping through days in a calendar (an iterator), and for generating calendar displays.
The **`time`** module includes the functions from POSIX which you may be familiar with. They are useful if you are porting code from other languages. One of the most commonly used methods from the time module is **`time.sleep`**`(`*seconds*`)`.
A method which many of these modules and objects share is **`strftime()`**, which comes from C but is supported by many other languages. It enables the date/time to be displayed in a variety of formats by specifying a format character prefixed by a % sign. There are a large number of formats available, see the online documentation for a list.

These modules are not suitable for timing operations, there is a **`timeit`** module for that, and **`os.times()`**.

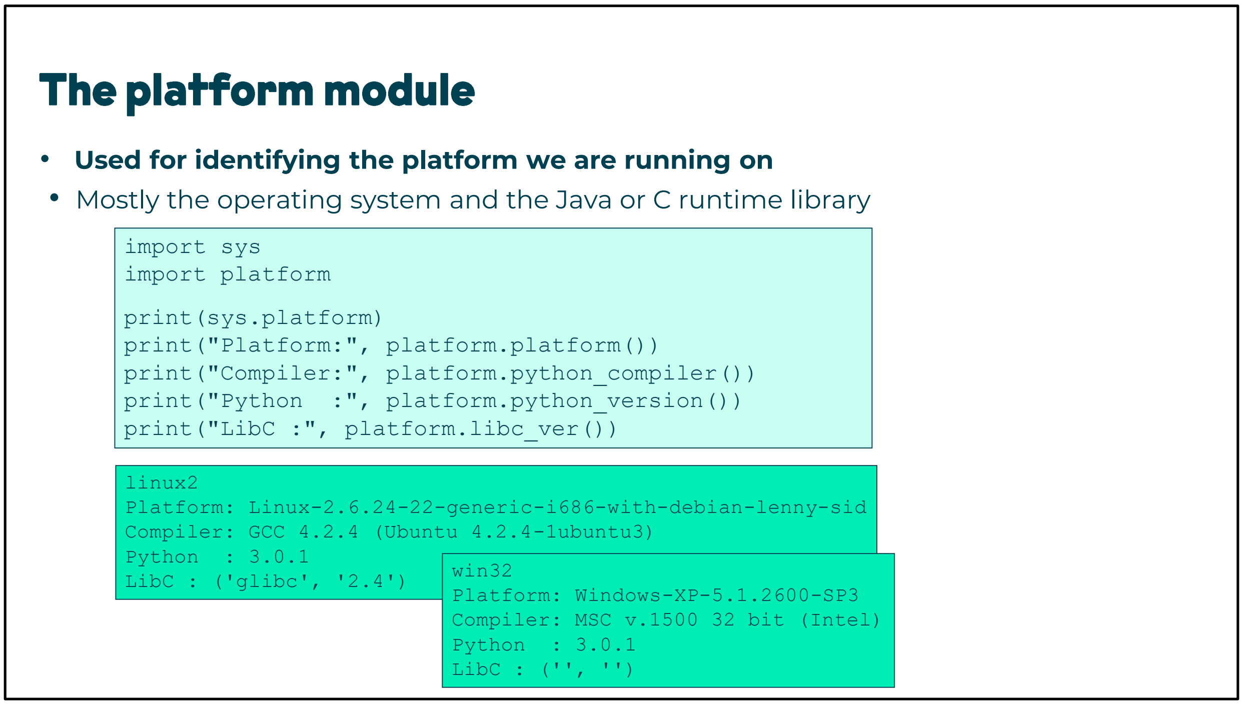


datetime example

- Calculate someone's age in years

```
import sys
Import time
from datetime import *
from calendar import *

sBirth = input("Enter birthday (dd/mm/yyyy):")
try:
    (day, month, year) = sBirth.split("/")
    dBirth = date(int(year), int(month), int(day))
except ValueError:
    print("Invalid date:", sBirth, file=sys.stderr)
    exit(1)

dYesday = date.fromtimestamp(time.time()- (24 * 60 * 60))

diff = dYesday - dBirth
diff = diff.days - leapdays(int(year), dYesday.year)
years = diff // 365
print("Client is", years, "years old")
```

This example reads a date of birth and calculates the person's age in years.
We are explicitly trapping **`ValueError`** here, because this exception could be raised by **`split()`** if there are no '/' characters in the string, or could be raised by **`date`** for an invalid date.
We should not include today, it is not yet over!
The **`diff`** variable is an example of a **`timedelta`** object (from **`datetime`** module), which has an attribute of the number of days.
By the way, in case you forgot, the **`//`** operator carries out an integer division. That is, it ignores any value in the result after the decimal point.



The platform module

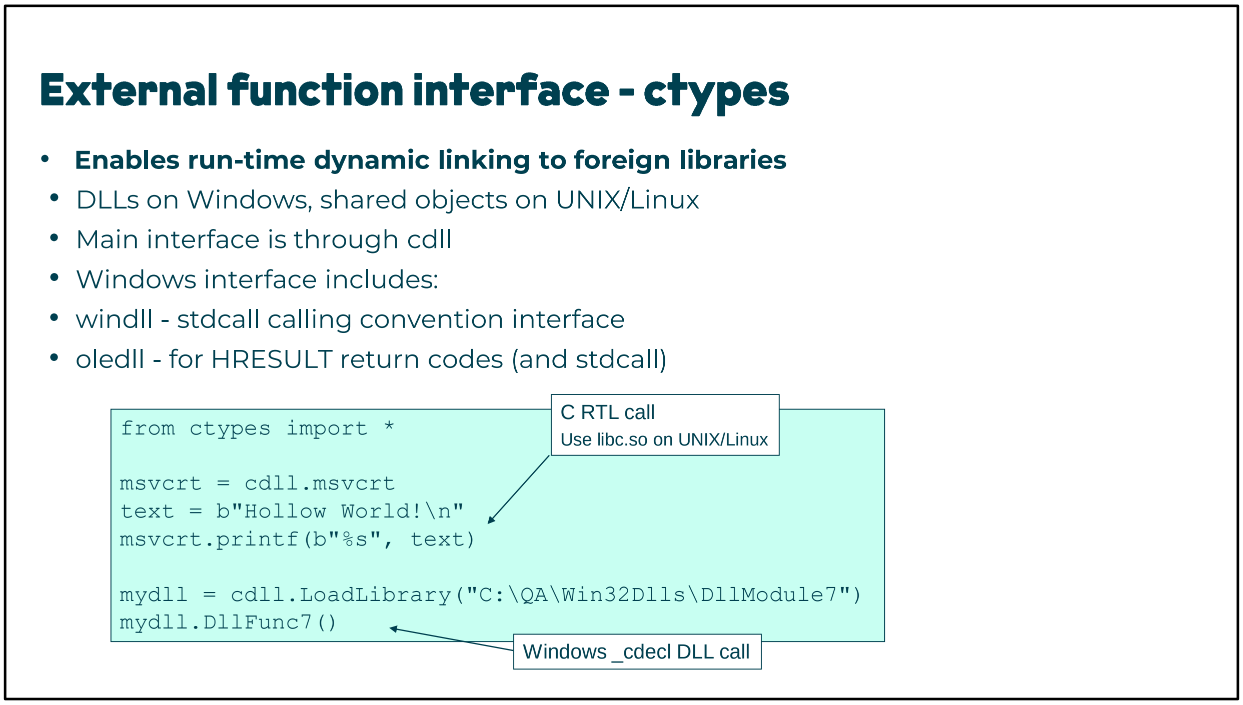
- **Used for identifying the platform we are running on**
- Mostly the operating system and the Java or C runtime library

```
import sys
import platform

print(sys.platform)
print("Platform:", platform.platform())
print("Compiler:", platform.python_compiler())
print("Python  :", platform.python_version())
print("LibC :", platform.libc_ver())
```

```
linux2
Platform: Linux-2.6.24-22-generic-i686-with-debian-lenny-sid
Compiler: GCC 4.2.4 (Ubuntu 4.2.4-1ubuntu3)
Python  : 3.0.1
LibC : ('glibc', '2.4')
```

```
win32
Platform: Windows-XP-5.1.2600-SP3
Compiler: MSC v.1500 32 bit (Intel)
Python  : 3.0.1
LibC : ('', '')
```

This module has nothing to do with railways! This module is safer, and more efficient, than calling a command like `uname(1)`, which even varies between different versions of UNIX. There are many more methods available in platform, even a version of `uname`. It can overlap in functionality with other modules, for example `sys.platform` is often used when only general platform details are required, and there is an `os.uname()`, but only on UNIX.



External function interface - ctypes

- **Enables run-time dynamic linking to foreign libraries**
- DLLs on Windows, shared objects on UNIX/Linux
- Main interface is through cdll
- Windows interface includes:
- windll - stdcall calling convention interface
- oledll - for HRESULT return codes (and stdcall)

```
from ctypes import *

msvcrt = cdll.msvcrt
text = b"Hollow World!\n"
msvcrt.printf(b"%s", text)

mydll = cdll.LoadLibrary("C:\QA\Win32Dlls\DllModule7")
mydll.DllFunc7()
```

C RTL call
Use libc.so on UNIX/Linux

Windows _cdecl DLL call

The **`ctypes`** library module is an easy way to interface Python with C/C++ written shared libraries. There is a slight performance penalty over a pure C interface in that each call: every variable and value has to have a Python wrapper built around it, however that is more than compensated by its ease of use. You do have to know quite a lot about C to be able to use this module, not to mention the shared library you are calling.
The code shown is for Windows, on Linux you can also load the C runtime:

```
libc = cdll.LoadLibrary("libc.so.6")
libc.printf("Hello world\n")
```

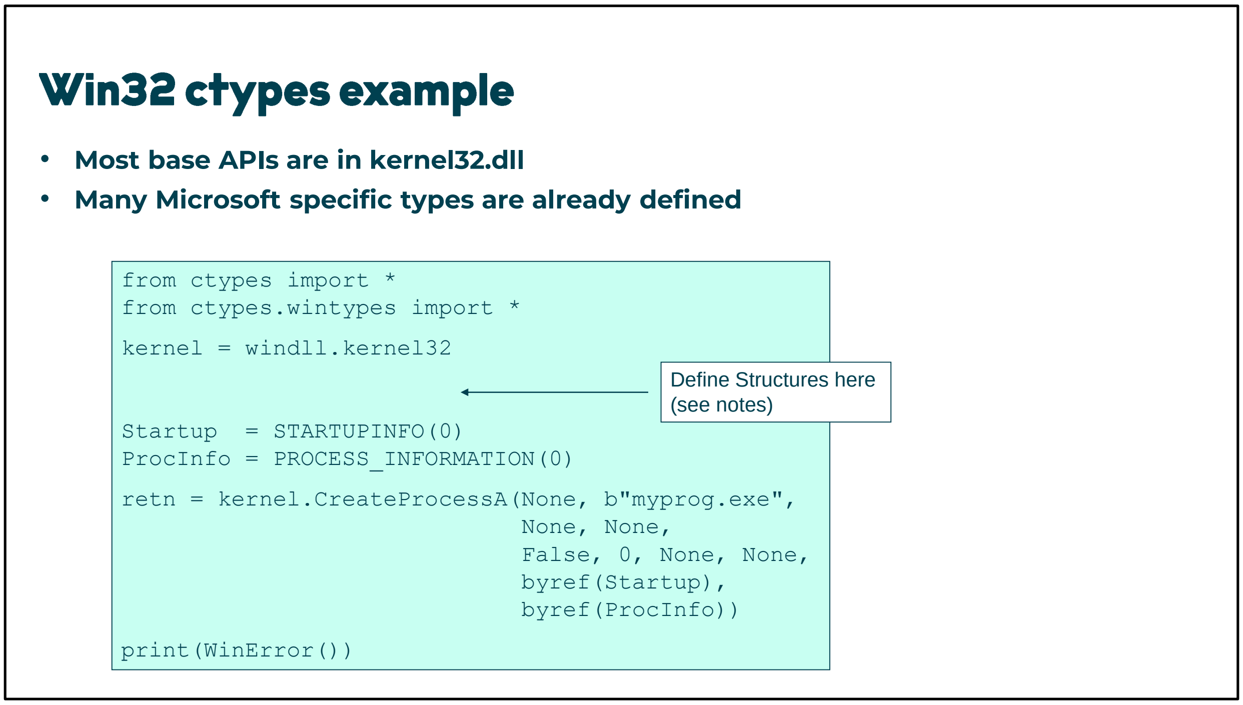
The example on the next slide calls the Win32 API **`CreateProcess`**.
Notice:

> `CreateProcess` itself is not called, that is because it is actually a macro to **`CreateProcessA`** or `CreateProcessW` depending on the character set used for the call (A - ASCII, W - Wide characters). We have elected to use the ASCII interface here, which means we need to use Python byte-strings (we would not have to supply the b" " decoration in Python 2).

We use **`windll`** rather than `cdll` because most Win32 APIs use the Windows `__stdcall` calling convention (the exceptions are those in the C runtime library, such as `_beginthreadex`).

We are using `WinError` to find the error message, if any. This calls (by default) `GetLastError()` and `FormatMessage()`.

`ctypes` is an extensive module with many features - see the documentation for further details.



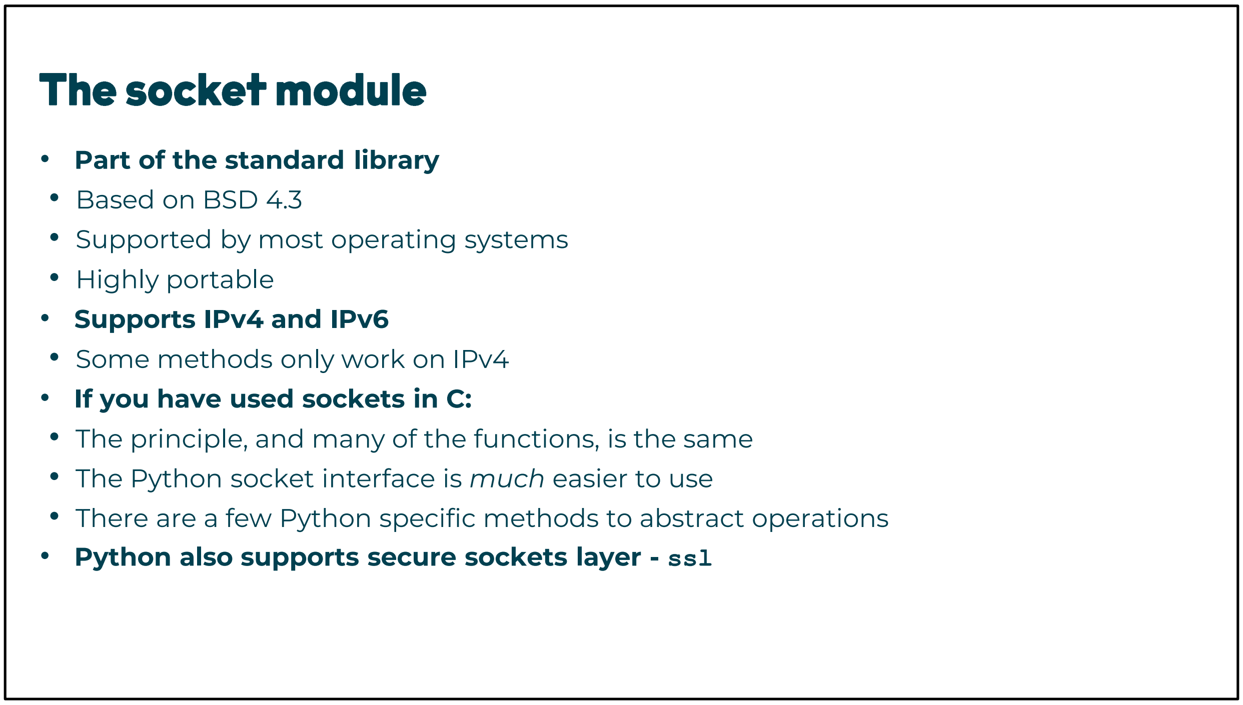
Win32 ctypes example

- Most base APIs are in kernel32.dll
- Many Microsoft specific types are already defined

```
from ctypes import *
from ctypes.wintypes import *

kernel = windll.kernel32

                                  ←   Define Structures here
                                      (see notes)

Startup  = STARTUPINFO(0)
ProcInfo = PROCESS_INFORMATION(0)

retn = kernel.CreateProcessA(None, b"myprog.exe",
                             None, None,
                             False, 0, None, None,
                             byref(Startup),
                             byref(ProcInfo))

print(WinError())
```

For an explanation of the example, see the notes for the previous slide.

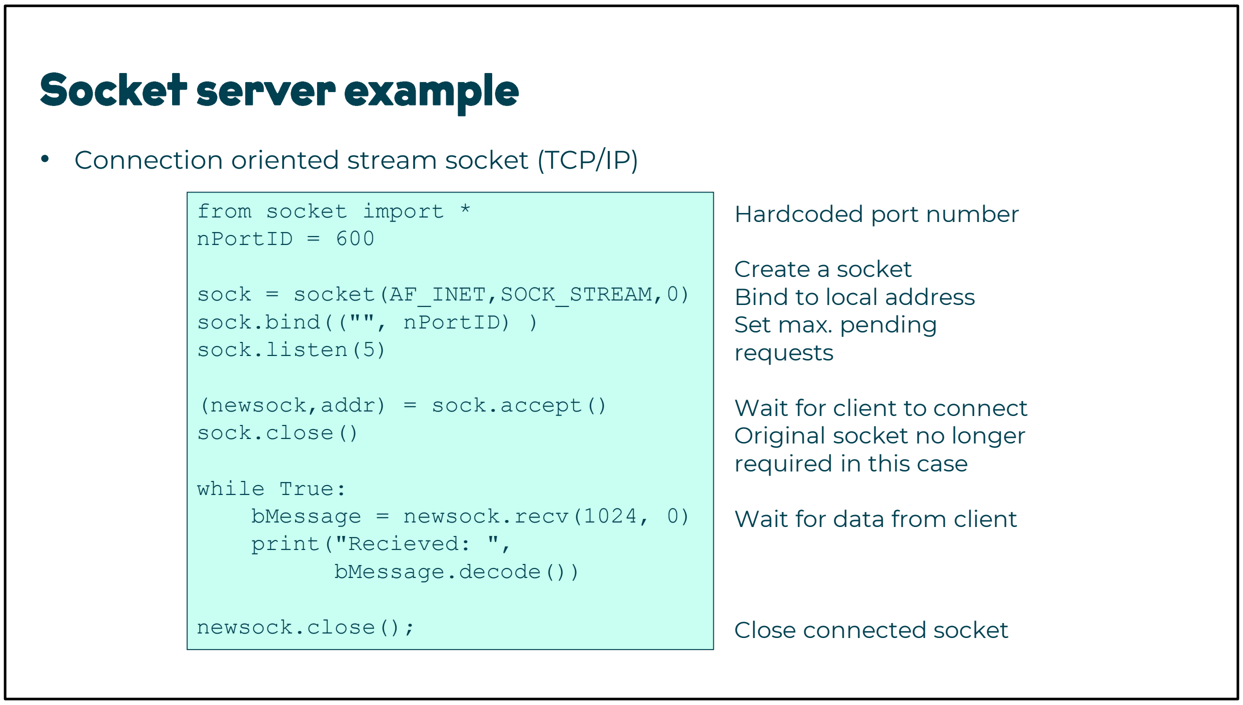
```
class STARTUPINFO(Structure):
    _fields_ = [
        ("cb",                DWORD),
        ("lpReserved",        LPSTR),
        ("lpDesktop",         LPSTR),
        ("lpTitle",           LPSTR),
        ("dwX",               DWORD),
        ("dwY",               DWORD),
        ("dwXSize",           DWORD),
        ("dwYSize",           DWORD),
        ("dwXCountChars",     DWORD),
        ("dwYCountChars",     DWORD),
        ("dwFillAttribute",   DWORD),
        ("dwFlags",           DWORD),
        ("wShowWindow",       WORD),
        ("cbReserved2",       WORD),
        ("lpReserved2",       c_char_p),  #LPBYTE
        ("hStdInput",         HANDLE),
        ("hStdOutput",        HANDLE),
        ("hStdError",         HANDLE)
    ]

class PROCESS_INFORMATION(Structure):
    _fields_ = [
        ("hProcess",    HANDLE),
        ("hThread",     HANDLE),
        ("dwProcessId", DWORD),
        ("dwThreadId",  DWORD)
    ]
```



The socket module

- **Part of the standard library**
  - Based on BSD 4.3
  - Supported by most operating systems
  - Highly portable
- **Supports IPv4 and IPv6**
  - Some methods only work on IPv4
- **If you have used sockets in C:**
  - The principle, and many of the functions, is the same
  - The Python socket interface is *much* easier to use
  - There are a few Python specific methods to abstract operations
- **Python also supports secure sockets layer - `ssl`**

The socket module is a network programming interface which is based on the Berkeley Sockets programming model and is consistent with release 4.3 of the Berkeley Software Distribution. The API also contains a set of Python-specific extensions.
The module provides a single API which abstracts the networking software below and to which developers can program. It closely follows the traditional C SPI, but is considerably abstracted and easier to use. Code written using this module is portable, for example, code runs on Windows (using WinSock) and Linux without modification.
The conventional BSD socket API only supported IPv4, not IPv6. Modern operating systems can support IPv6, including Windows from XP onwards, but some data structures and some parameters are different. The Microsoft utility Checkv4.exe is available as a porting tool for conversion to IPv4 to IPv6.
The Python standard library ssl provides a secure socket class derived from socket.



Socket server example

- Connection oriented stream socket (TCP/IP)

```
from socket import *
nPortID = 600

sock = socket(AF_INET,SOCK_STREAM,0)
sock.bind(("", nPortID) )
sock.listen(5)

(newsock,addr) = sock.accept()
sock.close()

while True:
    bMessage = newsock.recv(1024, 0)
    print("Recieved: ",
          bMessage.decode())

newsock.close();
```

Hardcoded port number

Create a socket
Bind to local address
Set max. pending
requests

Wait for client to connect
Original socket no longer
required in this case

Wait for data from client

Close connected socket

The socket calls shown will be familiar to anyone who has written similar code in C, or other languages. The main difference though is that it is much easier!
In particular, the **`bind()`** statement can be, um, a bind. Specifying the address in another language can be horribly complex. Here we just supply a two element tuple, the first being the IP address or hostname - the Python module does all the nastiness for us. The default hostname, shown, is `INADDR_ANY`, which for a server means we accept incoming requests from any machine.
The **`listen()`** call makes this a listening socket (it does not block) and the historic value of 5 maximum pending requests is usually used (in an old version of BSD sockets, 5 was the limit).
The **`accept()`** is where we wait for a client to connect. It returns a new socket which is now used for the conversation with the client. In addition to that socket, we also get the address details of the client, which we can use to check its IP address, write to an audit trail, and so on.
With a multi-client system, we would normally loop around the **`accept()`** call, creating a thread or another process to handle the client-server conversation. In this case, there is only one

connection, and so we can close the original socket.
The example shows IPv4, but this module also supports IPv6 and very few changes are required to support it, mostly in the way that the address is specified.

Other modules

There are many others in the Standard Library

http://docs.python.org/3.4/library/index.html

See also http://www.doughellmann.com/PyMOTW

Other modules are available

For example, Win32 module versions:

http://www.lfd.uci.edu/~gohlke/pythonlibs

PyPI - Python Package Index

http://pypi.python.org/pypi - note Python 3 sidebar link

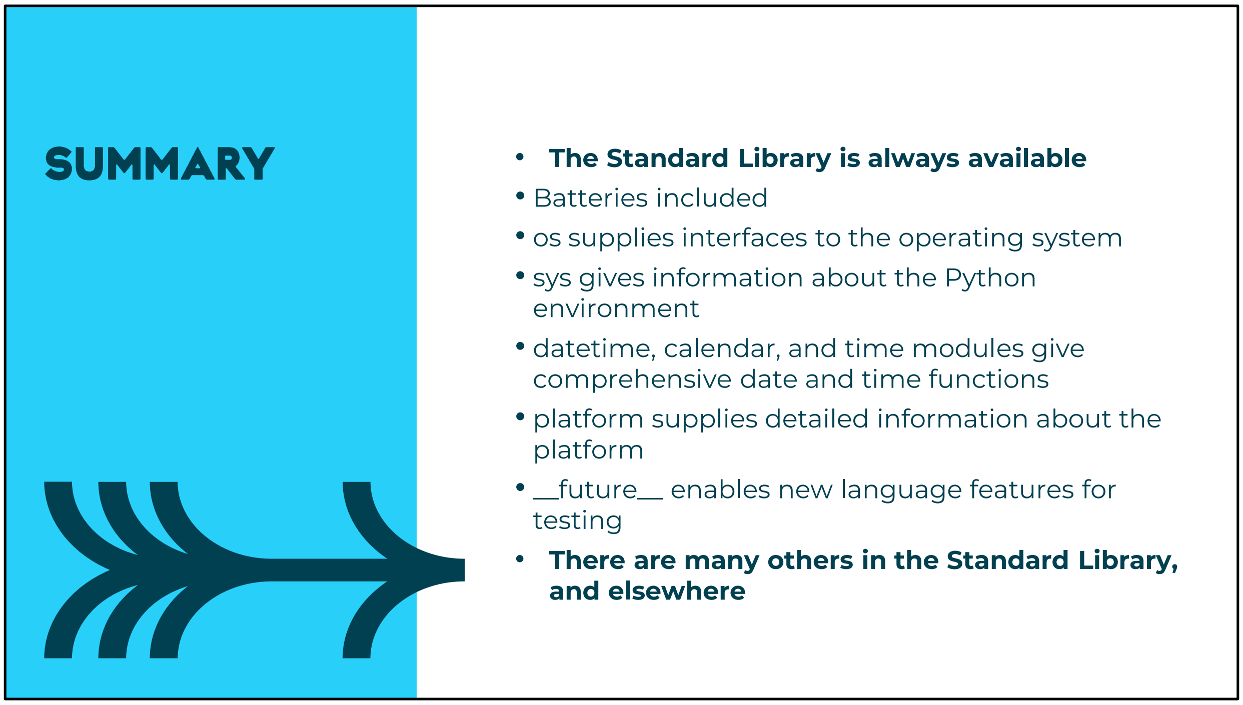
Also known as "The cheese shop"

`pip` is the preferred installation route for most

Easy Install is now obsolete

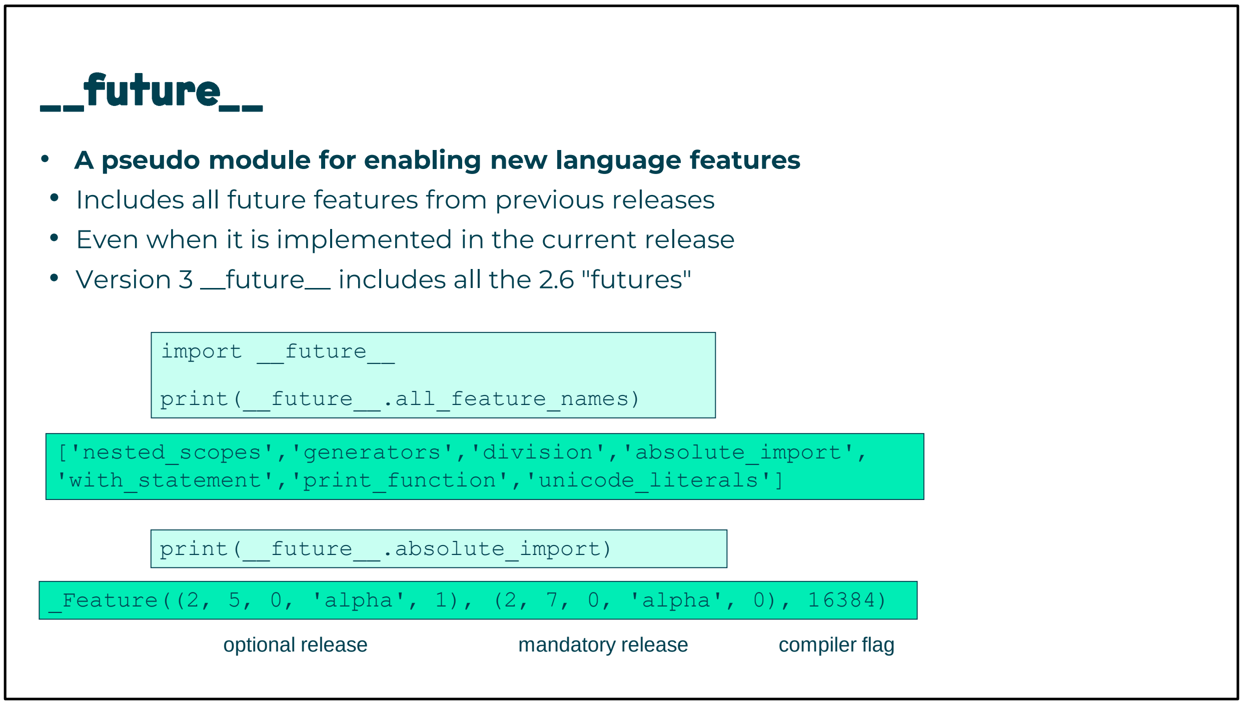
We have seen several standard library modules throughout this course, but many more are available. See Doug Hellmann's PyMOTW (Python Module Of The Week) blog, or his " The Python Standard Library By Example" book.
The number of modules available for Python 3 has increased considerably and is expected to continue to rise. Get at them:
http://pypi.python.org/pypi?:action=browse&c=533&show=all
See also, the CheeseShop:
http://cheeseshop.python.org [python.org] - a repository of links and downloads for Python projects.

There is already no need to manually download and build most Python packages, and pip can scan web pages for links to Python-built Windows installers, source tarballs, and its own "egg" (zip-based) and newer "wheel" binary format, as well as deal with Subversion URLs and Sourceforge mirrors. Many packages can in fact be installed and run in zipped form without ever extracting the files to disk, using less space and speeding up imports.



SUMMARY

- **The Standard Library is always available**
- Batteries included
- os supplies interfaces to the operating system
- sys gives information about the Python environment
- datetime, calendar, and time modules give comprehensive date and time functions
- platform supplies detailed information about the platform
- __future__ enables new language features for testing
- **There are many others in the Standard Library, and elsewhere**



__future__

- **A pseudo module for enabling new language features**
- Includes all future features from previous releases
- Even when it is implemented in the current release
- Version 3 __future__ includes all the 2.6 "futures"

```
import __future__
print(__future__.all_feature_names)
```

```
['nested_scopes','generators','division','absolute_import',
'with_statement','print_function','unicode_literals']
```

```
print(__future__.absolute_import)
```

```
_Feature((2, 5, 0, 'alpha', 1), (2, 7, 0, 'alpha', 0), 16384)
```

optional release mandatory release compiler flag

Planned new features in Python are often made available early. To make them available to your program, import the __future__ module. The module does not actually contain the new features, it just allows them to be used, and supplies some information about them. The mechanism enables existing programs to test to see if their local functions are impacted with the new names, and to test the impact of new syntax.
Information supplied for each feature name is the release number at which it is first present (the optional release), followed by the release number at which the feature is mandatory.
Future features are not removed from this module, so **`all_feature_names`** may include features that were implemented several versions ago.
In November 2009, a moratorium on further language changes was made for Python 3, to be lifted on June 2011. This does not mean there cannot be improvements to Python in that time, just no more language changes. See PEP 3003 for further details.