Future

Future statement :is a real module, and serves

* purposes:

- To avoid confusing existing tools that analyze import statements and expect to find the modules they're importing.
- To ensure that future statements run under releases prior to 2.1 at least yield runtime exceptions To document when incompatible changes were introduced, and when they will be or weremade mandatory. This is a form of executable documentation, and can be inspected programmatically via importing future and examining its contents.
- To document when incompatible changes were introduced, and when they will be or were made mandatory. This is a form of executable documentation, and can be inspected programmatically via importing futureand examining its contents

Calendar

The functions and classes defined in this module use an idealized calendar, the current Gregorian calendar extended indefinitely in both directions.

Calendar instances have the following methods

iterweekdays

Return an iterator for the week day numbers that will be used for one week.

itermonthdates(year, month)

Return an iterator for the month month (1–12) in the year year. This iterator will return all days (as datetime.date objects) for the month and all days before the start of the month or after the end of the month that are required to get a complete week.

itermonthdays(year, month)

Return an iterator for the month month in the year year similar to itermonthdates(), but not restricted by the datetime.date range.

itermonthdays2(year, month)

Return an iterator for the month month in the year year similar to itermonthdates(), but not restricted by the datetime.date range.

Doctest

The doctest module searches for pieces of text that look like interactive Python sessions.

There are several common ways to use doctest:

To check that a module's docstrings are up-to-date by verifying that all interactive examples still work as documented

To perform regression testing by verifying that interactive examples from a test file or a test object work as expected

To write tutorial documentation for a package, liberally illustrated with input-output examples. Depending on whether the examples or the expository text are emphasized, this has the flavor of "literate testing" or "executable documentation"

Email

The email package is a library for managing email messages. It is specifically not designed to do any sending of email messages to SMTP , those are functions of modules such as smtplib and nntplib .

The overall structure of the email package can be divided into three major components, plus a fourth component that controls the behavior of the other components

The central component of the package is an "object model" that represents email messages. An application interacts with the package

primarily through the object model interface defined in the message sub-module

The other two major components of the package are the parser and the generator. The parser takes the serialized version of an email message (a stream of bytes) and converts it into a tree of EmailMessage objects. The generator takes an EmailMessage and turns it back into a serialized byte stream

Cross

Queries the given executable (defaults to the Python interpreter binary) for various architecture information

Returns a tuple (bits, linkage) which contain information about the bit architecture and the linkage format used for the executable. Both values are returned as strings

Values that cannot be determined are returned as given by the parameter presets. If bits is given as '', the sizeof(pointer) is used as indicator for the supported pointer size

platform.machine()

Returns the machine type, e.g. 'i386'. An empty string is returned if the .value cannot be determined

platform.node()

Returns the computer’s network name (may not be fully qualified!). An empty string is returned if the value cannot be determined