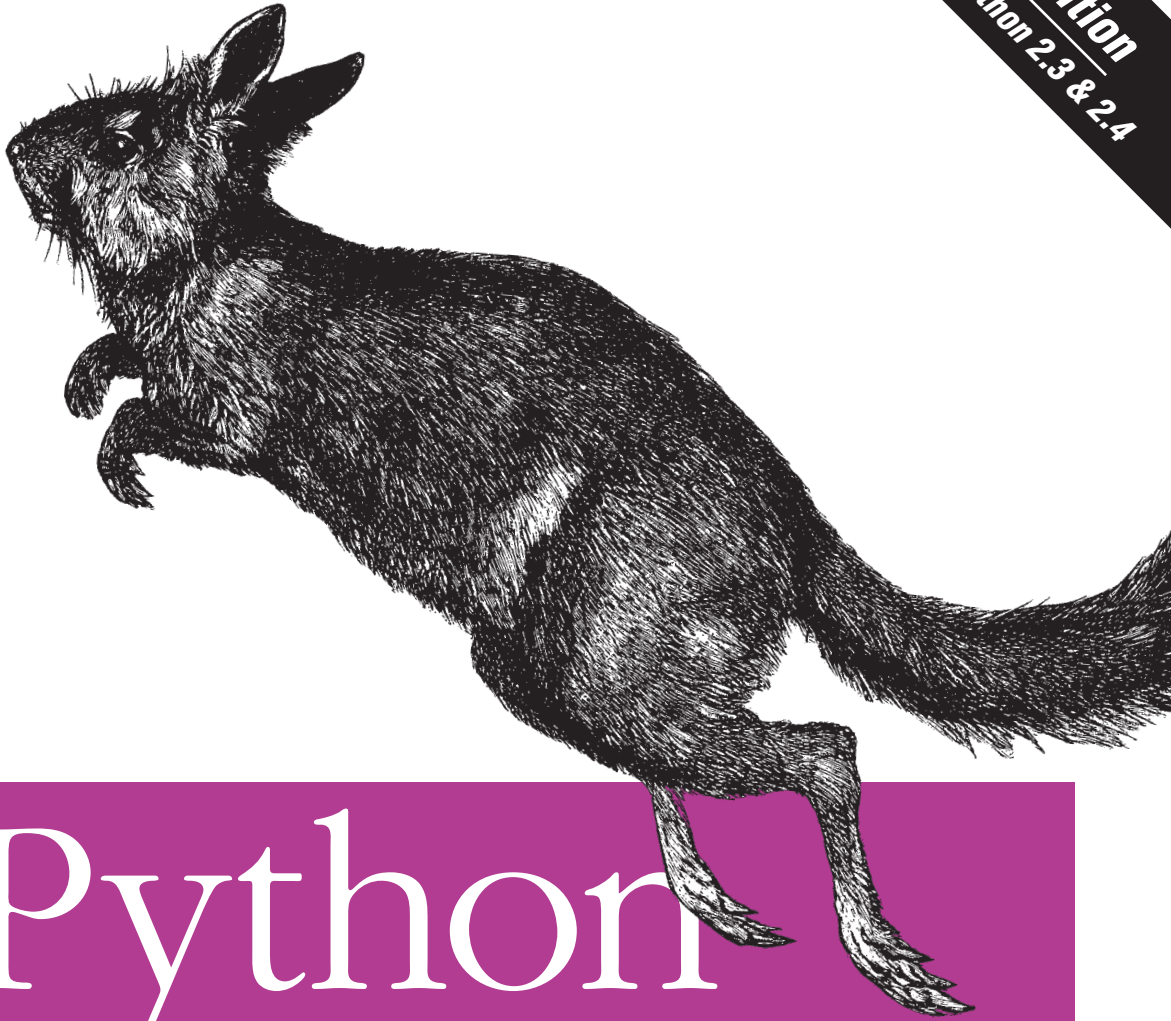


Recipes from the Python Community

2nd Edition
Covers Python 2.3 & 2.4



Python Cookbook™

O'REILLY®

Alex Martelli, Anna Ravenscroft & David Ascher

Time and Money

3.0 Introduction

Credit: Gustavo Niemeyer, Facundo Batista

Today, last weekend, next year. These terms sound so common. You have probably wondered, at least once, about how deeply our lives are involved in the very idea of time. The concept of time surrounds us, and, as a consequence, it's also present in the vast majority of software projects. Even very simple programs may have to deal with timestamps, delays, timeouts, speed gauges, calendars, and so on. As befits a general-purpose language that is proud to come with “batteries included,” Python's standard library offers solid support for these application needs, and more support yet comes from third-party modules and packages.

Computing tasks involving money are another interesting topic that catches our attention because it's so closely related to our daily lives. Python 2.4 introduced support for decimal numbers (and you can retrofit that support into 2.3, see http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html), making Python a good option even for computations where you must avoid using binary floats, as ones involving money so often are.

This chapter covers exactly these two topics, money and time. According to the old saying, maybe we should claim the chapter is really about a *single* topic, since after all, as everybody knows—time *is* money!

The time Module

Python Standard Library's `time` module lets Python applications access a good portion of the time-related functionality offered by the platform Python is running on. Your platform's documentation for the equivalent functions in the C library will therefore be useful, and some peculiarities of different platforms will affect Python as well.

One of the most used functions from module `time` is the one that obtains the current time—`time.time`. This function’s return value may be a little cryptic for the uninitiated: it’s a floating-point number that corresponds to the number of seconds passed since a fixed instant called the *epoch*, which may change depending on your platform but is usually midnight of January 1, 1970.

To check which epoch your platform uses, try, at any Python interactive interpreter prompt:

```
>>> import time
>>> print time.asctime(time.gmtime(0))
```

Notice we’re passing 0 (meaning 0 seconds after the epoch) to the `time.gmtime` function. `time.gmtime` converts any timestamp (in seconds since the epoch) into a tuple that represents that precise instant of time in human terms, without applying any kind of time zone conversion (GMT stands for “Greenwich mean time”, an old but colorful way to refer to what is now known as UTC, for “Coordinated Universal Time”). You can also pass a timestamp (in seconds since the epoch) to `time.localtime`, which applies the current local notion of time zone.

It’s important to understand the difference, since, if you have a timestamp that is *already* offset to represent a local time, passing it to the `time.localtime` function will *not* yield the expected result—unless you’re so lucky that your local time zone happens to coincide with the UTC time zone, of course!

Here is a way to unpack a tuple representing the current local time:

```
year, month, mday, hour, minute, second, wday, yday = time.localtime()
```

While valid, this code is not elegant, and it would certainly not be practical to use it often. This kind of construct may be completely avoided, since the tuples returned by the time functions let you access their elements via meaningful attribute names. Obtaining the current month then becomes a simple and elegant expression:

```
time.localtime().tm_mon
```

Note that we omitted passing any argument to `localtime`. When we call `localtime`, `gmtime`, or `asctime` without an argument, each of them conveniently defaults to using the current time.

Two very useful functions in module `time` are `strftime`, which lets you build a string from a time tuple, and `strptime`, which goes the other way, parsing a string and producing a time tuple. Each of these two functions accepts a format string that lets you specify exactly what you want in the resulting string (or, respectively, what you expect from the string you’re parsing) in excruciating detail. For all the formatting specifications that you can use in the format strings you pass to these functions, see <http://docs.python.org/lib/module-time.html>.

One last important function in module `time` is the `time.sleep` function, which lets you introduce delays in Python programs. Even though this function’s POSIX coun-

terpart accepts only an integer parameter, the Python equivalent supports a float and allows sub-second delays to be achieved. For instance:

```
for i in range(10):
    time.sleep(0.5)
    print "Tick!"
```

This snippet will take about 5 seconds to execute, emitting Tick! approximately twice per second.

Time and Date Objects

While module `time` is quite useful, the Python Standard Library also includes the `datetime` module, which supplies types that provide better abstractions for the concepts of dates and times—namely, the types `time`, `date`, and `datetime`. Constructing instances of those types is quite elegant:

```
today = datetime.date.today()
birthday = datetime.date(1977, 5, 4)      #May 4
currenttime = datetime.datetime.now().time()
lunchtime = datetime.time(12, 00)
now = datetime.datetime.now()
epoch = datetime.datetime(1970, 1, 1)
meeting = datetime.datetime(2005, 8, 3, 15, 30)
```

Further, as you'd expect, instances of these types offer comfortable information access and useful operations through their attributes and methods. The following statements create an instance of the `date` type, representing the current day, then obtain the same date in the next year, and finally print the result in a dotted format:

```
today = datetime.date.today()
next_year = today.replace(year=today.year+1).strftime("%Y.%m.%d")
print next_year
```

Notice how the year was incremented, using the `replace` method. Assigning to the attributes of `date` and `time` instances may sound tempting, but these instances are immutable (which is a good thing, because it means we can use the instances as members in a set, or keys in a dictionary!), so new instances must be created instead of changing existing ones.

Module `datetime` also provides basic support for *time deltas* (differences between instants of time; you can think of them as basically meaning *durations* in time), through the `timedelta` type. This type lets you change a given date by incrementing or decrementing the date by a given time slice, and it is also the result of taking the difference between times or dates.

```
>>> import datetime
>>> NewYearsDay = datetime.date(2005, 01, 01)
>>> NewYearsEve = datetime.date(2004, 12, 31)
>>> oneday = NewYearsDay - NewYearsEve
>>> print oneday
```

```
1 day, 0:00:00
>>>
```

A `timedelta` instance is internally represented by days, seconds, and microseconds, but you can construct `timedelta` instances by passing any of these arguments and also other multipliers, like minutes, hours and weeks. Support for other kinds of deltas, like months, and years, is not available—on purpose, since their meanings, and operation results, are debatable. (This feature is, however, offered by the third-party `dateutil` package—see <https://moin.conectiva.com.br/DateUtil>.)

`datetime` can be described as a *prudent* or *cautious* design. The decision of not implementing doubtful tasks, and tasks that may need many different implementations in different systems, reflects the strategy used to develop all of the module. This way, the module offers good interfaces for most use cases, and, even more importantly, a strong and coherent base for third-party modules to build upon.

Another area where this cautious design strategy for `datetime` shows starkly is the module's time zone support. Even though `datetime` offers nice ways to query and set time zone information, they're not really useful without an external source to provide nonabstract subclasses of the `tzinfo` type. At least two third-party packages provide time zone support for `datetime`: `dateutil`, mentioned previously, and `pyTZ`, available at <http://sourceforge.net/projects/pytz/>.

Decimal

`decimal` is a Python Standard Library module, new in Python 2.4, which finally brings decimal arithmetic to Python. Thanks to `decimal`, we now have a decimal numeric data type, with bounded precision and floating point. Let's look at each of these three little phrases in more detail:

Decimal numeric data type

The number is not stored in binary, but rather, as a sequence of decimal digits.

With bounded precision

The number of digits each number stores is fixed. (It is a fixed parameter of each decimal number object, but different decimal number objects can be set to use different numbers of digits.)

Floating point

The decimal point does not have a fixed place. (To put it another way: while the number has a fixed amount of digits *in total*, it does not have a fixed amount of digits *after the decimal point*. If it did, it would be a *fixed-point*, rather than *floating-point*, numeric data type).

Such a data type has many uses (the big use case is as the basis for money computations), particularly because `decimal.Decimal` offers many other advantages over standard binary `float`. The main advantage is that all of the decimal numbers that the user can enter (which is to say, all the decimal numbers with a finite number of digits

) can be represented exactly (in contrast, some of those numbers do not have an exact representation in binary floating point):

```
>>> import decimal
>>> 1.1
1.1000000000000001
>>> 2.3
2.2999999999999998
>>> decimal.Decimal("1.1")
Decimal("1.1")
>>> decimal.Decimal("2.3")
Decimal("2.3")
```

The exactness of the representation carries over into arithmetic. In binary floating point, for example:

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.5511151231257827e-17
```

Such differences are very close to zero, and yet they prevent reliable equality testing; moreover, even tiny differences can accumulate. For this reason, `decimal` should be preferred to binary floats in accounting applications that have strict equality requirements:

```
>>> d1 = decimal.Decimal("0.1")
>>> d3 = decimal.Decimal("0.3")
>>> d1 + d1 + d1 - d3
Decimal("0.0")
```

`decimal.Decimal` instances can be constructed from integers, strings, or tuples. To create a `decimal.Decimal` from a float, first convert the float to a string. This necessary step serves as an explicit reminder of the details of the conversion, including representation error. Decimal numbers include special values such as NaN (which stands for “not a number”), positive and negative Infinity, and `-0`. Once constructed, a `decimal.Decimal` object is immutable, just like any other number in Python.

The `decimal` module essentially implements the rules of arithmetic that are taught in school. Up to a given working precision, exact, unrounded results are given whenever possible:

```
>>> 0.9 / 10
0.08999999999999997
>>> decimal.Decimal("0.9") / decimal.Decimal(10)
Decimal("0.09")
```

Where the number of digits in a result exceeds the working precision, the number is rounded according to the current rounding method. Several rounding methods are available; the default is round-half-even.

The `decimal` module incorporates the notion of *significant digits*, so that, for example, `1.30+1.20` is `2.50`. The trailing zero is kept to indicate significance. This is the

usual representation for monetary applications. For multiplication, the “school-book” approach uses all the figures in the multiplicands:

```
>>> decimal.Decimal("1.3") * decimal.Decimal("1.2")
Decimal("1.56")
>>> decimal.Decimal("1.30") * decimal.Decimal("1.20")
Decimal("1.5600")
```

In addition to the standard numeric properties that decimal objects share with other built-in number types, such as float and int, decimal objects also have several specialized methods. Check the docs for all of the methods, with details and examples.

The decimal data type works within a *context*, where some configuration aspects are set. Each thread has its own current context (having a separate context per thread means that each thread may make changes without interfering with other threads); the current thread’s current context is accessed or changed using functions `getcontext` and `setcontext` from the decimal module.

Unlike hardware-based binary floating point, the precision of the decimal module can be set by users (defaulting to 28 places). It can be set to be as large as needed for a given problem:

```
>>> decimal.getcontext().prec = 6          # set the precision to 6...
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857")
>>> decimal.getcontext().prec = 60        # ...and to 60 digits
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857142857142857142857142857142857142857142857142857")
```

Not everything in decimal can be as simple and elementary as shown so far, of course. Essentially, decimal implements the standards for general decimal arithmetic which you can study in detail at <http://www2.hursley.ibm.com/decimal/>. In particular, this means that decimal supports the concept of *signals*. Signals represent abnormal conditions arising from computations (e.g., 1/0, 0/0, Infinity/Infinity). Depending on the needs of each specific application, signals may be ignored, considered as informational, or treated as exceptions. For each signal, there is a flag and a trap enabler. When a signal is encountered, its flag is incremented from zero, and then, if the trap enabler is set to one, an exception is raised. This gives programmers a great deal of power and flexibility in configuring decimal to meet their exact needs.

Given all of these advantages for decimal, why would someone want to stick with float? Indeed, is there any reason why Python (like just about every other widespread language, with Cobol and Rexx the two major exceptions that easily come to mind) originally adopted floating-point binary numbers as its default (or only) non-integer data type? Of course—many reasons can be provided, and they’re all spelled *speed!* Consider:

```
$ python -mtimeit -s'from decimal import Decimal as D' 'D("1.2")+D("3.4")'
10000 loops, best of 3: 191 usec per loop
```

```
$ python -mtimeit -s'from decimal import Decimal as D' '1.2+3.4'  
1000000 loops, best of 3: 0.339 usec per loop
```

This basically translates to: on this machine (an old Athlon 1.2 GHz PC running Linux), Python can perform almost 3 million sums per second on floats (using the PC's arithmetic hardware), but only a bit more than 5 thousand sums per second on Decimals (all done in software and with all the niceties shown previously).

Essentially, if your application must sum many tens of millions of noninteger numbers, you had better stick with float! When an average machine was a thousand times slower than it is today (and it wasn't *all* that long ago!), such limitations hit even applications doing relatively small amounts of computation, if the applications ran on reasonably cheap machines (again, we see time and money both playing a role!). Rexx and Cobol were born on mainframes that were not quite as fast as today's cheapest PCs but thousands of times more expensive. Purchasers of such mainframes could *afford* nice and friendly decimal arithmetic, but most other languages, born on more reasonably priced machines (or meant for computationally intensive tasks), just couldn't.

Fortunately, relatively few applications actually need to perform so much arithmetic on non-integers as to give any observable performance problems on today's typical machines. Thus, today, most applications can actually take advantage of decimal's many beneficial aspects, including applications that must continue to use Python 2.3, even though decimal is in the Python Standard Library only since version 2.4. To learn how you can easily integrate decimal into Python 2.3, see http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html.

3.1 Calculating Yesterday and Tomorrow

Credit: Andrea Cavalcanti

Problem

You want to get today's date, then calculate yesterday's or tomorrow's.

Solution

Whenever you have to deal with a “change” or “difference” in time, think `timedelta`:

```
import datetime  
today = datetime.date.today()  
yesterday = today - datetime.timedelta(days=1)  
tomorrow = today + datetime.timedelta(days=1)  
print yesterday, today, tomorrow  
#emits: 2004-11-17 2004-11-18 2004-11-19
```

Discussion

This recipe's Problem has been a fairly frequent question on Python mailing lists since the `datetime` module arrived. When first confronted with this task, it's quite

common for people to try to code it as `yesterday = today - 1`, which gives a `TypeError: unsupported operand type(s) for -: 'datetime.date' and 'int'`.

Some people have called this a bug, implying that Python should guess what they mean. However, one of the guiding principles that gives Python its simplicity and power is: “in the face of ambiguity, refuse the temptation to guess.” Trying to guess would clutter `datetime` with heuristics meant to guess that you “really meant 1 day”, rather than 1 second (which `timedelta` also supports), or 1 year.

Rather than trying to *guess* what you mean, Python, as usual, expects you to make your meaning explicit. If you want to subtract a time difference of one day, you code that explicitly. If, instead, you want to add a time difference of one second, you can use `timedelta` with a `datetime.datetime` object, and then you code the operation using exactly the same syntax. This way, for each task you might want to perform, there’s only one obvious way of doing it. This approach also allows a fair amount of flexibility, without added complexity. Consider the following interactive snippet:

```
>>> anniversary = today + datetime.timedelta(days=365)           # add 1 year
>>> print anniversary
2005-11-18
>>> t = datetime.datetime.today()                                # get right now
>>> t
datetime.datetime(2004, 11, 19, 10, 12, 43, 801000)
>>> t2 = t + datetime.timedelta(seconds=1)                       # add 1 second
>>> t2
datetime.datetime(2004, 11, 19, 10, 12, 44, 801000)
>>> t3 = t + datetime.timedelta(seconds=3600)                   # add 1 hour
>>> t3
datetime.datetime(2004, 11, 19, 11, 12, 43, 801000)
```

Keep in mind that, if you want fancier control over date and time arithmetic, third-party packages, such as `dateutil` (which works together with the built-in `datetime`) and the classic `mx.DateTime`, are available. For example:

```
from dateutil import relativedelta
nextweek = today + relativedelta.relativedelta(weeks=1)
print nextweek
#emits: 2004-11-25
```

However, “always do the simplest thing that can possibly work.” For simple, straightforward tasks such as the ones in this recipe, `datetime.timedelta` works just fine.

See Also

`dateutil` documentation at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>, and `datetime` documentation in the *Library Reference*. `mx.DateTime` can be found at <http://www.egenix.com/files/python/mxDateTime.html>. `mx.DateTime` can be found at <http://www.egenix.com/files/python/mxDateTime.html>.

3.2 Finding Last Friday

Credit: Kent Johnson, Danny Yoo, Jonathan Gennick, Michael Wener

Problem

You want to find the date of last Friday (or today, if today is Friday) and print it in a specified format.

Solution

You can use the `datetime` module from Python’s standard library to easily achieve this:

```
import datetime, calendar
lastFriday = datetime.date.today()
oneday = datetime.timedelta(days=1)
while lastFriday.weekday() != calendar.FRIDAY:
    lastFriday -= oneday
print lastFriday.strftime('%A, %d-%b-%Y')
# emits, e.g.: Friday, 10-Dec-2004
```

Discussion

The handy little snippet of code in this recipe lets us find a previous weekday and print the properly formatted date, regardless of whether that weekday is in the same month, or even the same year. In this example, we’re looking for the last Friday (or today, if today is Friday). Friday’s integer representation is 4, but to avoid depending on this “magical number,” we just import the Python Standard Library `calendar` module and rely instead on its `calendar.FRIDAY` attribute (which, sure enough, is the number 4). We set a variable called `lastFriday` to today’s date and work backward until we have reached a date with the desired weekday value of 4.

Once we have the date we desire, formatting the date in any way we like is easily achieved with the “string formatting” method `strftime` of the `datetime.date` class.

An alternative, slightly more terse solution uses the built-in constant `datetime.date.resolution` instead of explicitly building the `datetime.timedelta` instance to represent one day’s duration:

```
import datetime, calendar
lastFriday = datetime.date.today()
while lastFriday.weekday() != calendar.FRIDAY:
    lastFriday -= datetime.date.resolution
print lastFriday.strftime('%d-%b-%Y')
```

The `datetime.date.resolution` class attribute has exactly the same value as the `oneday` variable in the recipe’s Solution—the time interval of one day. However, `resolution` can trip you up. The value of the class attribute `resolution` varies among various classes of the `datetime` module—for the `date` class it’s `timedelta(days=1)`, but for the

time and datetime classes, it's `timedelta(microseconds=1)`. You *could* mix-and-match (e.g., add `datetime.date.resolution` to a `datetime.datetime` instance), but it's easy to get confused doing so. The version in this recipe's Solution, using the explicitly named and defined `oneday` variable, is just as general, more explicit, and less confusing. Thus, all in all, that version is more Pythonic (which is why it's presented as the "official" one!).

A more important enhancement is that we don't really need to loop, decrementing a date by one at each step through the loop: we can, in fact, get to the desired target in one fell swoop, computing the number of days to subtract thanks to the wonders of modular arithmetic:

```
import datetime, calendar
today = datetime.date.today()
targetDay = calendar.FRIDAY
thisDay = today.weekday()
deltaToTarget = (thisDay - targetDay) % 7
lastFriday = today - datetime.timedelta(days=deltaToTarget)
print lastFriday.strftime('%d-%b-%Y')
```

If you don't follow why this works, you may want to brush up on modular arithmetic, for example at <http://www.cut-the-knot.org/blue/Modulo.shtml>.

Use the approach you find clearest, without worrying about performance. Remember Hoare's dictum (often misattributed to Knuth, who was in fact quoting Hoare): "premature optimization is the root of all evil in programming." Let's see why thinking of optimization *would* be premature here.

Net of the common parts (computing today's date, and formatting and emitting the result) on a four-year-old PC, with Linux and Python 2.4, the slowest approach (the one chosen for presentation as the "Solution" because it's probably the clearest and most obvious one) takes 18.4 microseconds; the fastest approach (the one avoiding the loop, with some further tweaks to really get *pedal to the metal*) takes 10.1 microseconds.

You're not going to compute last Friday's date often enough, in your life, to be able to tell the difference at 8 microseconds a pop (much less if you use recent hardware rather than a box that's four years old). If you consider the time needed to compute today's date and to format the result, you need to add 37 microseconds to each timing, even net of the I/O time for the `print` statement; so, the range of performance is roughly between 55 microseconds for the slowest and clearest form, and 47 microseconds for the fastest and tersest one—clearly not worth worrying about.

See Also

`datetime` module and `strftime` documentation in the *Library Reference* (currently at <http://www.python.org/doc/lib/module-datetime.html> and <http://www.python.org/doc/current/lib/node208.html>).

3.3 Calculating Time Periods in a Date Range

Credit: Andrea Cavalcanti

Problem

Given two dates, you want to calculate the number of weeks between them.

Solution

Once again, the standard `datetime` and third-party `dateutil` modules (particularly `dateutil`'s `rrule.count` method) come in quite handy. After importing the appropriate modules, it's a really simple job:

```
from dateutil import rrule
import datetime
def weeks_between(start_date, end_date):
    weeks = rrule.rrule(rrule.WEEKLY, dtstart=start_date, until=end_date)
    return weeks.count()
```

Discussion

Function `weeks_between` takes the starting and ending dates as arguments, instantiates a rule to recur weekly between them, and returns the result of the rule's `count` method—faster to code than to describe. This method will return only an integer (it won't return “half” weeks). For example, eight days is considered two weeks. It's easy to code a test for this:

```
if __name__ == '__main__':
    starts = [datetime.date(2005, 01, 04), datetime.date(2005, 01, 03)]
    end = datetime.date(2005, 01, 10)
    for s in starts:
        days = rrule.rrule(rrule.DAILY, dtstart=s, until=end).count()
        print "%d days shows as %d weeks" % (days, weeks_between(s, end))
```

This test emits the following output:

```
7 days shows as 1 weeks
8 days shows as 2 weeks
```

It's not necessary to give a name to a recurrence rule, if you don't want to—changing the function's body, for example, to the single statement:

```
return rrule.rrule(rrule.WEEKLY, dtstart=start_date, until=end_date).count()
```

works just as well. I prefer to name recurrence rules because (frankly) I still find them a bit weird, even though they're so incredibly useful I doubt I could do without them!

See Also

Refer to the `dateutil` module's documentation available at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>, `datetime` documentation in the *Library Reference*.

3.4 Summing Durations of Songs

Credit: Anna Martelli Ravenscroft

Problem

You want to find out the total duration of a playlist of songs.

Solution

Use the `datetime` standard module and the built-in function `sum` to handle this task:

```
import datetime
def totaltimer(times):
    td = datetime.timedelta(0)    # initial value of sum (must be a timedelta)
    duration = sum([
        datetime.timedelta(minutes=m, seconds=s) for m, s in times],
        td)
    return duration
if __name__ == '__main__':      # test when module run as main script
    times1 = [(2, 36),         # list containing tuples (minutes, seconds)
              (3, 35),
              (3, 45),]
    times2 = [(3, 0),
              (5, 13),
              (4, 12),
              (1, 10),]
    assert totaltimer(times1) == datetime.timedelta(0, 596)
    assert totaltimer(times2) == datetime.timedelta(0, 815)
    print ("Tests passed.\n"
          "First test total: %s\n"
          "Second test total: %s" % (
            totaltimer(times1), totaltimer(times2)))
```

Discussion

I have a large playlist of songs I listen to during workouts. I wanted to create a select list but wanted to know the total duration of the selected songs, without having to create the new playlist first. I wrote this little script to handle the task.

A `datetime.timedelta` is normally what's returned when calculating the difference between two `datetime` objects. However, you can create your own `timedelta` instance to represent any given *duration* of time (while other classes of the `datetime` module,

such as class `datetime`, have instances that represent a *point* in time). Here, we need to sum durations, so, clearly, it's exactly `timedelta` that we need.

`datetime.timedelta` takes a variety of optional arguments: days, seconds, microseconds, milliseconds, minutes, hours, weeks. So, to create an instance, you really should pass named arguments when you call the class to avoid confusion. If you simply call `datetime.timedelta(m, n)`, without naming the arguments, the class uses positional notation and treats `m` and `n` as days and seconds, which produces really strange results. (I found this out the hard way . . . a good demonstration of the need to *test!*)

To use the built-in function `sum` on a list of objects such as `timedeltas`, you have to pass to `sum` a second argument to use as the initial value—otherwise, the default initial value is 0, integer zero, and you get an error as soon as you try to sum a `timedelta` with that `int`. All objects in the iterable that you pass as `sum`'s first argument should be able to support *numeric* addition. (Strings are *specifically* disallowed, but, take my earnest advice: *don't* use `sum` for concatenating a lot of lists either!) In Python 2.4, instead of a list comprehension for `sum`'s first argument, we could use a generator expression by replacing the square brackets, [and], with parentheses, (and)—which might be handy if you're trying to handle a playlist of several thousand songs.

For the test case, I manually created a list of tuples with the durations of the songs in minutes and seconds. The script could be enhanced to parse the times in different formats (such as `mm:ss`) or to read the information from a file or directly from your music library.

See Also

Library Reference on `sum` and `datetime`.

3.5 Calculating the Number of Weekdays Between Two Dates

Credit: Anna Martelli Ravenscroft

Problem

You want to calculate the number of weekdays (working days), as opposed to calendar days, that fall between two dates.

Solution

Since weekends and other “days off” vary by country, by region, even sometimes within a single company, there is no built-in way to perform this task. However,

using `dateutil` along with `datetime` objects, it's reasonably simple to code a solution:

```
from dateutil import rrule
import datetime
def workdays(start, end, holidays=0, days_off=None):
    if days_off is None:
        days_off = 5, 6          # default to: saturdays and sundays
    workdays = [x for x in range(7) if x not in days_off]
    days = rrule.rrule(rrule.DAILY, dtstart=start, until=end,
                      byweekday=workdays)
    return days.count() - holidays
if __name__ == '__main__':
    # test when run as main script
    testdates = [ (datetime.date(2004, 9, 1), datetime.date(2004, 11, 14), 2),
                  (datetime.date(2003, 2, 28), datetime.date(2003, 3, 3), 1), ]
    def test(testdates, days_off=None):
        for s, e, h in testdates:
            print 'total workdays from %s to %s is %s with %s holidays' % (
                s, e, workdays(s, e, h, days_off), h)
    test(testdates)
    test(testdates, days_off=[6])
```

Discussion

This project was my very first one in Python: I needed to know the number of actual days in training of our trainees, given a start date and end date (inclusive). This problem was a bit trickier back in Python 2.2; today, the `datetime` module and the `dateutil` third-party package make the problem much simpler to solve.

Function `workdays` starts by assigning a reasonable default value to variable `days_off` (unless an explicit value was passed for it as an argument), which is a sequence of the weekday numbers of our normal days off. In my company, weekly days off varied among individuals but were usually fewer than the workdays, so it was easier to track and modify the days off rather than the workdays. I made this an argument to the function so that I can easily pass a different value for `days_off` if and when I have different needs. Then, the function uses a list comprehension to create a list of actual weekly workdays, which are all weekdays not in `days_off`. Now the function is ready to do its calculations.

The workhorse in this recipe is an instance, named `days`, of `dateutil`'s `rrule` (recurrence rule) class. Class `rrule` may be instantiated with various parameters to produce a rule object. In this example, I pass a frequency (`rrule.DAILY`), a beginning date and an ending date—both of which must be `datetime.date` objects—and which weekdays to include (`workdays`). Then, I simply call method `days.count` to count the number of occurrences generated by the rule. (See recipe 3.3 “Calculating Time Periods in a Date Range” for other uses for the `count` method of `rrule`.)

You can easily set your own definition of *weekend*: just pass as `days_off` whichever values you need. In this recipe, the default value is set to the standard U.S. weekend

of Saturday and Sunday. However, if your company normally works a four-day week, say, Tuesday through Friday, you would pass `days_off=(5, 6, 0)`. Just be sure to pass the `days_off` value as an iterable, such as a list or tuple, even if, as in the second test, you only have a single day in that container.

A simple but useful enhancement might be to automatically check whether your start and end dates are weekends (for weekend-shift workers), and use an `if/else` to handle the weekend shifts, with appropriate changes to `days_off`. Further enhancements would be to add the ability to enter sick days, or to perform a call to an automatic holiday lookup function, rather than passing the number of holidays directly, as I do in this recipe. See recipe 3.6 “Looking up Holidays Automatically” for a simple implementation of a holidays list for this purpose.

See Also

Refer to the `dateutil` documentation, which is available at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>, `datetime` documentation in the *Library Reference*; recipe 3.3 “Calculating Time Periods in a Date Range” for another use of `rrule.count`; recipe 3.6 “Looking up Holidays Automatically” for automatic holiday lookup.

3.6 Looking up Holidays Automatically

Credit: Anna Martelli Ravenscroft, Alex Martelli

Problem

Holidays vary by country, by region, even by union within the same company. You want an automatic way to determine the number of holidays that fall between two given dates.

Solution

Between two dates, there may be movable holidays, such as Easter and Labor Day (U.S.); holidays that are based on Easter, such as Boxing Day; holidays with a fixed date, such as Christmas; holidays that your company has designated (the CEO’s birthday). You can deal with all of them using `datetime` and the third-party module `dateutil`.

A very flexible architecture is to factor out the various possibilities into separate functions to be called as appropriate:

```
import datetime
from dateutil import rrule, easter
try: set
except NameError: from sets import Set as set
def all_easter(start, end):
    # return the list of Easter dates within start..end
```

```

    easters = [easter.easter(y)
               for y in xrange(start.year, end.year+1)]
    return [d for d in easters if start<=d<=end]
def all_boxing(start, end):
    # return the list of Boxing Day dates within start..end
    one_day = datetime.timedelta(days=1)
    boxings = [easter.easter(y)+one_day
               for y in xrange(start.year, end.year+1)]
    return [d for d in boxings if start<=d<=end]
def all_christmas(start, end):
    # return the list of Christmas Day dates within start..end
    christmases = [datetime.date(y, 12, 25)
                   for y in xrange(start.year, end.year+1)]
    return [d for d in christmases if start<=d<=end]
def all_labor(start, end):
    # return the list of Labor Day dates within start..end
    labors = rrule.rrule(rrule.YEARLY, bymonth=9, byweekday=rrule.MO(1),
                        dtstart=start, until=end)
    return [d.date() for d in labors] # no need to test for in-between here
def read_holidays(start, end, holidays_file='holidays.txt'):
    # return the list of dates from holidays_file within start..end
    try:
        holidays_file = open(holidays_file)
    except IOError, err:
        print 'cannot read holidays (%r):' % (holidays_file,), err
        return []
    holidays = []
    for line in holidays_file:
        # skip blank lines and comments
        if line.isspace() or line.startswith('#'):
            continue
        # try to parse the format: YYYY, M, D
        try:
            y, m, d = [int(x.strip()) for x in line.split(',')]
            date = datetime.date(y, m, d)
        except ValueError:
            # diagnose invalid line and just go on
            print "Invalid line %r in holidays file %r" % (
                line, holidays_file)
            continue
        if start<=date<=end:
            holidays.append(date)
    holidays_file.close()
    return holidays
holidays_by_country = {
    # map each country code to a sequence of functions
    'US': (all_easter, all_christmas, all_labor),
    'IT': (all_easter, all_boxing, all_christmas),
}
def holidays(cc, start, end, holidays_file='holidays.txt'):
    # read applicable holidays from the file
    all_holidays = read_holidays(start, end, holidays_file)
    # add all holidays computed by applicable functions
    functions = holidays_by_country.get(cc, ())

```

```

    for function in functions:
        all_holidays += function(start, end)
    # eliminate duplicates
    all_holidays = list(set(all_holidays))
    # uncomment the following 2 lines to return a sorted list:
    # all_holidays.sort()
    # return all_holidays
    return len(all_holidays)    # comment this out if returning list
if __name__ == '__main__':
    test_file = open('test_holidays.txt', 'w')
    test_file.write('2004, 9, 6\n')
    test_file.close()
    testdates = [ (datetime.date(2004, 8, 1), datetime.date(2004, 11, 14)),
                  (datetime.date(2003, 2, 28), datetime.date(2003, 5, 30)),
                  (datetime.date(2004, 2, 28), datetime.date(2004, 5, 30)),
                  ]
    def test(cc, testdates, expected):
        for (s, e), expect in zip(testdates, expected):
            print 'total holidays in %s from %s to %s is %d (exp %d)' % (
                cc, s, e, holidays(cc, s, e, test_file.name), expect)
        print
    test('US', testdates, (1,1,1) )
    test('IT', testdates, (1,2,2) )
    import os
    os.remove(test_file.name)

```

Discussion

In one company I worked for, there were three different unions, and holidays varied among the unions by contract. In addition, we had to track any snow days or other release days in the same way as “official” holidays. To deal with all the potential variations in holidays, it’s easiest to factor out the calculation of standard holidays into their own functions, as we did in the preceding example for `all_easter`, `all_labor`, and so on. Examples of different types of calculations are provided so it’s easy to roll your own as needed.

Although half-open intervals (with the lower bound included but the upper one excluded) are the norm in Python (and for good reasons, since they’re arithmetically more malleable and tend to induce fewer bugs in your computations!), this recipe deals with closed intervals instead (both lower and upper bounds included). Unfortunately, that’s how specifications in terms of date intervals tend to be given, and `dateutil` also works that way, so the choice was essentially obvious.

Each function is responsible for ensuring that it only returns results that meet our criteria: lists of `datetime.date` instances that lie between the dates (inclusive) passed to the function. For example, in `all_labor`, we coerce the `datetime.datetime` results returned by `dateutil`’s `rrule` into `datetime.date` instances with the `date` method.

A company may choose to set a specific date as a holiday (such as a snow day) “just this once,” and a text file may be used to hold such unique instances. In our exam-

ple, the `read_holidays` function handles the task of reading and processing a text file, with one date per line, each in the format year, month, day. You could also choose to refactor this function to use a “fuzzy” date parser, as shown in recipe 3.7 “Fuzzy Parsing of Dates.”

If you need to look up holidays many times within a single run of your program, you may apply the optimization of reading and parsing the text file just once, then using the list of dates parsed from its contents each time that data is needed. However, “premature optimization is the root of all evil in programming,” as Knuth said, quoting Hoare: by avoiding even this “obvious” optimization, we gain clarity and flexibility. Imagine these functions being used in an interactive environment, where the text file containing holidays may be edited between one computation and the next: by rereading the file each time, there is no need for any special check about whether the file was changed since you last read it!

Since countries often celebrate different holidays, the recipe provides a rudimentary `holidays_by_country` dictionary. You can consult plenty of web sites that list holidays by country to flesh out the dictionary for your needs. The important part is that this dictionary allows a different group of holidays-generating functions to be called, depending on which country code is passed to the `holidays` function. If your company has multiple unions, you could easily create a union-based dictionary, passing the union-code instead of (or for multinationals, in addition to) a country code to `holidays`. The `holidays` function calls the appropriate functions (including, unconditionally, `read_holidays`), concatenates the results, eliminates duplicates, and returns the length of the list. If you prefer, of course, you can return the list instead, by simply uncommenting two lines as indicated in the code.

See Also

Recipe 3.7 “Fuzzy Parsing of Dates” for fuzzy parsing; `dateutil` documentation at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>, `datetime` documentation in the *Library Reference*.

3.7 Fuzzy Parsing of Dates

Credit: Andrea Cavalcanti

Problem

Your program needs to read and accept dates that don’t conform to the `datetime` standard format of “`yyyy, mm, dd`”.

Solution

The third-party `dateutil.parser` module provides a simple answer:

```
import datetime
import dateutil.parser
def tryparse(date):
    # dateutil.parser needs a string argument: let's make one from our
    # `date` argument, according to a few reasonable conventions...:
    kwargs = {} # assume no named-args
    if isinstance(date, (tuple, list)):
        date = ' '.join([str(x) for x in date]) # join up sequences
    elif isinstance(date, int):
        date = str(date) # stringify integers
    elif isinstance(date, dict):
        kwargs = date # accept named-args dicts
        date = kwargs.pop('date') # with a 'date' str
    try:
        try:
            parsedate = dateutil.parser.parse(date, **kwargs)
            print 'Sharp %r -> %s' % (date, parsedate)
        except ValueError:
            parsedate = dateutil.parser.parse(date, fuzzy=True, **kwargs)
            print 'Fuzzy %r -> %s' % (date, parsedate)
        except Exception, err:
            print 'Try as I may, I cannot parse %r (%s)' % (date, err)
    if __name__ == "__main__":
        tests = (
            "January 3, 2003", # a string
            (5, "Oct", 55), # a tuple
            "Thursday, November 18", # longer string without year
            "7/24/04", # a string with slashes
            "24-7-2004", # European-format string
            {'date':"5-10-1955", "dayfirst":True}, # a dict including the kwarg
            "5-10-1955", # dayfirst, no kwarg
            19950317, # not a string
            "11AM on the 11th day of 11th month, in the year of our Lord 1945",
        )
        for test in tests:
            tryparse(test) # testing date formats
            # try to parse
```

Discussion

`dateutil.parser`'s `parse` function works on a variety of date formats. This recipe demonstrates a few of them. The parser can handle English-language month-names and two- or four-digit years (with some constraints). When you call `parse` without named arguments, its default is to first try parsing the string argument in the following order: `mm-dd-yy`. If that does not make logical sense, as, for example, it doesn't for the `'24-7-2004'` string in the recipe, `parse` then tries `dd-mm-yy`. Lastly, it tries `yy-mm-dd`. If a "keyword" such as `dayfirst` or `yearfirst` is passed (as we do in one test), `parse` attempts to parse based on that keyword.

The recipe tests define a few *edge cases* that a date parser might encounter, such as trying to pass the date as a tuple, an integer (ISO-formatted without spaces), and even a phrase. To allow testing of the keyword arguments, the `tryparse` function in the recipe also accepts a dictionary argument, expecting, in this case, to find in it the value of the string to be parsed in correspondence to key `'date'`, and passing the rest on to `dateutil`'s parser as keyword arguments.

`dateutil`'s parser can provide a pretty good level of “fuzzy” parsing, given *some* hints to let it know which piece is, for example, the hour (such as the AM in the test phrase in this recipe). For production code, you should avoid relying on fuzzy parsing, and either do some kind of preprocessing, or at least provide some kind of mechanism for checking the accuracy of the parsed date.

See Also

For more on date-parsing algorithms, see `dateutil` documentation at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>; for date handling, see the `datetime` documentation in the *Library Reference*.

3.8 Checking Whether Daylight Saving Time Is Currently in Effect

Credit: Doug Fort

Problem

You want to know whether daylight saving time is in effect in your local time zone today.

Solution

It's a natural temptation to check `time.daylight` for this purpose, but that doesn't work. Instead you need:

```
import time
def is_dst():
    return bool(time.localtime().tm_isdst)
```

Discussion

In my location (as in most others nowadays), `time.daylight` is always 1 because `time.daylight` means that this time zone has daylight saving time (DST) at *some* time during the year, whether or not DST is in effect *today*.

The very last item in the pseudo-tuple you get by calling `time.localtime`, on the other hand, is 1 only when DST is currently in effect, otherwise it's 0—which, in my experience, is exactly the information one usually needs to check. This recipe wraps

this check into a function, calling built-in type `bool` to ensure the result is an elegant `True` or `False` rather than a rougher `1` or `0`—optional refinements, but nice ones, I think. You could alternatively access the relevant item as `time.localtime()[-1]`, but using attribute-access syntax with the `tm_isdst` attribute name is more readable.

See Also

Library Reference and *Python in a Nutshell* about module `time`.

3.9 Converting Time Zones

Credit: Gustavo Niemeyer

Problem

You are in Spain and want to get the correct local (Spanish) time for an event in China.

Solution

Time zone support for `datetime` is available in the third-party `dateutil` package. Here's one way to set the local time zone, then print the current time to check that it worked properly:

```
from dateutil import tz
import datetime
posixstr = "CET-1CEST-2,M3.5.0/02:00,M10.5.0/03:00"
spaintz = tz.tzstr(posixstr)
print datetime.datetime.now(spaintz).ctime()
```

Conversion between different time zones is also possible, and often necessary in our expanding world. For instance, let's find out when the next Olympic Games will start, according to a Spanish clock:

```
chinatz = tz.tzoffset("China", 60*60*8)
olympicgames = datetime.datetime(2008, 8, 8, 20, 0, tzinfo=chinatz)
print olympicgames.astimezone(spaintz)
```

Discussion

The cryptic string named `posixstr` is a POSIX-style representation for the time zone currently being used in Spain. This string provides the standard and daylight saving time zone names (CST and CEST), their offsets (UTC+1 and UTC+2), and the day and hour when DST starts and ends (the last Sunday of March at 2 a.m., and the last Sunday of October at 3 a.m., respectively). We may check the DST zone bounds to ensure they are correct:

```
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 1, 59)) == "CET"
assert spaintz.tzname(datetime.datetime(2004, 03, 28, 2, 00)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 1, 59)) == "CEST"
assert spaintz.tzname(datetime.datetime(2004, 10, 31, 2, 00)) == "CET"
```

All of these asserts should pass silently, confirming that the time zone name switches between the right strings at the right times.

Observe that even though the return to the standard time zone is scheduled to 3a.m., the moment of the change is marked as 2 a.m. This happens because of a one-hour gap, between 2 a.m. and 3 a.m., that is ambiguous. That hour of time happens twice: once in the time zone CEST, and then again in the time zone CET. Currently, expressing this moment in an unambiguous way, using the standard Python date and time support, is not possible. This is why it is recommended that you store `datetime` instances in UTC, which is unambiguous, and only use time zone conversion for display purposes.

To do the conversion from China to Spain, we've used `tzoffset` to express the fact that China is eight hours ahead of UTC time (`tzoffset` is always compared to *UTC*, *not* to a particular time zone). Notice how the `datetime` instance is created with the time zone information. This is always necessary for converting between two different time zones, even if the given time is in the local time zone. If you don't create the instance with the time zone information, you'll get a `ValueError: astimezone() cannot be applied to a naive datetime`. `datetime` instances are always created *naive*—they ignore time zone issues entirely—unless you explicitly create them with a time zone. For this purpose, `dateutil` provides the `tzlocal` type, which creates instances representing the platform's idea of the local time zone.

Besides the types we have seen so far, `dateutil` also provides `tzutc`, which creates instances representing UTC; `tzfile`, which allows using standard binary time zone files; `tzical`, which creates instances representing iCalendar time zones; and many more besides.

See Also

Documentation about the `dateutil` module can be found at <https://moin.conectiva.com.br/DateUtil?action=highlight&value=DateUtil>, and `datetime` documentation in the *Library Reference*.

3.10 Running a Command Repeatedly

Credit: Philip Nunez

Problem

You need to run a command repeatedly, with arbitrary periodicity.

Solution

The `time.sleep` function offers a simple approach to this task:

```
import time, os, sys
def main(cmd, inc=60):
```

```

while True:
    os.system(cmd)
    time.sleep(inc)
if __name__ == '__main__':
    numargs = len(sys.argv) - 1
    if numargs < 1 or numargs > 2:
        print "usage: " + sys.argv[0] + " command [seconds_delay]"
        sys.exit(1)
    cmd = sys.argv[1]
    if numargs < 3:
        main(cmd)
    else:
        inc = int(sys.argv[2])
        main(cmd, inc)

```

Discussion

You can use this recipe with a command that periodically checks for something (e.g., polling), or with one that performs an endlessly repeating action, such as telling a browser to reload a URL whose contents change often, so as to always have a recent version of that URL for viewing. The recipe is structured into a function called `main` and a body that is preceded by the usual `if __name__=='__main__':` idiom, to execute only if the script runs as a main script. The body examines the command-line arguments you used with the script and calls `main` appropriately (or gives a usage message if there are too many or too few arguments). This is the best way to structure a script, to make its functionality also available to other scripts that may import it as a module.

The `main` function accepts a `cmd` string, which is a command you want to pass periodically to the operating system's shell, and, optionally, a period of time in seconds, with a default value of 60 (one minute). `main` loops forever, alternating between executing the command with `os.system` and waiting (without consuming resources) with `time.sleep`.

The script's body looks at the command-line arguments you used with the script, which it finds in `sys.argv`. The first argument, `sys.argv[0]`, is the name of the script, often useful when the script identifies itself as it prints out messages. The body checks that one or two other arguments, in addition to this name, are included. The first (and mandatory) is the command to be run. (You may need to enclose this command in quotes to preserve it from your shell's parsing: the important thing is that it must remain a single argument.) The second (and optional) argument is the delay in seconds between two runs of the command. If the second argument is missing, the body calls `main` with just the command argument, accepting the default delay (60 seconds).

Note that, if there is a second argument, the body transforms it from a string (all items in `sys.argv` are always strings) into an integer, which is done most simply by calling built-in type `int`:

```
inc = int(sys.argv[2])
```

If the second argument is a string that is not acceptable for transformation into an integer (in other words, if it's anything except a sequence of digits), this call to `int` raises an exception and terminates the script with appropriate error messages. As one of Python's design principles states, "errors should not pass silently, unless explicitly silenced." It would be bad design to let the script accept an arbitrary string as its second argument, silently taking a default action if that string was not a correct integer representation!

For a variant of this recipe that uses the standard Python library module `sched`, rather than explicit looping and sleeping, see recipe 3.11 "Scheduling Commands."

See Also

Documentation of the standard library modules `os`, `time`, and `sys` in the *Library Reference* and *Python in a Nutshell*; recipe 3.11 "Scheduling Commands."

3.11 Scheduling Commands

Credit: Peter Cogolo

Problem

You need to schedule commands for execution at certain times.

Solution

That's what the `sched` module of the standard library is for:

```
import time, os, sys, sched
schedule = sched.scheduler(time.time, time.sleep)
def perform_command(cmd, inc):
    schedule.enter(inc, 0, perform_command, (cmd, inc)) # re-scheduler
    os.system(cmd)
def main(cmd, inc=60):
    schedule.enter(0, 0, perform_command, (cmd, inc)) # 0==right now
    schedule.run()
if __name__ == '__main__':
    numargs = len(sys.argv) - 1
    if numargs < 1 or numargs > 2:
        print "usage: " + sys.argv[0] + " command [seconds_delay]"
        sys.exit(1)
    cmd = sys.argv[1]
    if numargs < 3:
        main(cmd)
    else:
```

```
inc = int(sys.argv[2])
main(cmd, inc)
```

Discussion

This recipe implements the same functionality as in the previous recipe 3.10 “Running a Command Repeatedly,” but instead of that recipe’s simpler roll-our-own approach, this one uses the standard library module `sched`.

`sched` is a reasonably simple, yet flexible and powerful, module for scheduling tasks that must take place at given times in the future. To use `sched`, you first instantiate a scheduler object, such as `Scheduler` (shown in this recipe’s Solution), with two arguments. The first argument is the function to call in order to find out what time it is—normally `time.time`, which returns the current time as a number of seconds from an arbitrary reference point known as the *epoch*. The second argument is the function to call to wait for some time—normally `time.sleep`. You can also pass functions that measure time in arbitrary artificial ways. For example, you can use `sched` for such tasks as simulation programs. However, measuring time in artificial ways is an advanced use of `sched` not covered in this recipe.

Once you have a `Scheduler` instance `s`, you schedule events by calling either `s.enter`, to schedule something at a relative time `n` seconds from now (you can pass `n` as 0 to schedule something for *right now*), or `s.enterabs`, to schedule something at a given absolute time. In either case, you pass the time (relative or absolute), a priority (if multiple events are scheduled for the same time, they execute in priority order, lowest-priority first), a function to call, and a tuple of arguments to call that function with. Each of these two methods return an *event identifier*, an arbitrary token that you may store somewhere and later use to cancel a scheduled event by passing the event’s token as the argument in a call to `s.cancel`—another advanced use which this recipe does not cover.

After scheduling some events, you call `s.run`, which keeps running until the queue of scheduled events is empty. In this recipe, we show how to schedule a periodic, recurring event: function `perform_command` reschedules itself for `inc` seconds later in the future as the first thing it does, before running the specified system command. In this way, the queue of scheduled events never empties, and function `perform_command` keeps getting called with regular periodicity. This self-rescheduling is an important idiom, not just in using `sched`, but any time you have a “one-shot” way to ask for something to happen in the future, and you need instead to have something happen in the future “periodically”. (Tkinter’s `after` method, e.g., also works in exactly this way, and thus is also often used with just this kind of self-rescheduling idiom.)

Even for a task as simple as the one handled by this recipe, `sched` still has a small advantage over the simpler roll-your-own approach used previously in recipe 3.10 “Running a Command Repeatedly.” In recipe 3.10, the specified delay occurs between the *end* of one execution of `cmd` and the *beginning* of the next execution. If

the execution of `cmd` takes a highly variable amount of time (as is often the case, e.g., for commands that must wait for the network, or some busy server, etc.), then the command is not really being run periodically. In this recipe, the delay occurs between *beginning* successive runs of `cmd`, so that periodicity is indeed guaranteed. If a certain run of `cmd` takes longer than `inc` seconds, the schedule temporarily falls behind, but it will eventually catch up again, as long as the *average* running time of `cmd` is less than `inc` seconds: `sched` never “skips” events. (If you *do* want an event to be skipped because it’s not relevant any more, you have to keep track of the event identifier token and use the `cancel` method.)

For a detailed explanation of this script’s structure and body, see recipe 3.10 “Running a Command Repeatedly.”

See Also

Recipe 3.10 “Running a Command Repeatedly”; documentation of the standard library modules `os`, `time`, `sys`, and `sched` in the *Library Reference* and *Python in a Nutshell*.

3.12 Doing Decimal Arithmetic

Credit: Anna Martelli Ravenscroft

Problem

You want to perform some simple arithmetic computations in Python 2.4, but you want decimal results, *not* the Python default of `float`.

Solution

To get the normal, expected results from plain, simple computations, use the `decimal` module introduced in Python 2.4:

```
>>> import decimal
>>> d1 = decimal.Decimal('0.3') # assign a decimal-number object
>>> d1/3 # try some division
Decimal("0.1")
>>> (d1/3)*3 # can we get back where we started?
Decimal("0.3")
```

Discussion

Newcomers to Python (particularly ones without experience with binary float calculations in other programming languages) are often surprised by the results of seemingly simple calculations. For example:

```
>>> f1 = .3 # assign a float
>>> f1/3 # try some division
0.09999999999999999
```

```
>>> (f1/3)*3                # can we get back where we started?
0.29999999999999999
```

Binary floating-point arithmetic is the default in Python for very good reasons. You can read all about them in the Python FAQ (Frequently Asked Questions) document at <http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate>, and even in the appendix to the *Python Tutorial* at <http://docs.python.org/tut/node15.html>.

Many people, however, were unsatisfied with binary floats being the *only* option—they wanted to be able to specify the precision, or wanted to use decimal arithmetic for monetary calculations with predictable results. Some of us just wanted the predictable results. (A True Numerical Analyst *does*, of course, find all results of binary floating-point computations to be perfectly predictable; if any of you three are reading this chapter, you can skip to the next recipe, thanks.)

The new `decimal` type affords a great deal of control over the *context* for your calculations, allowing you, for example, to set the precision and rounding method to use for the results. However, when all you want is to run simple arithmetical operations that return predictable results, `decimal`'s default context works just fine.

Just keep in mind a few points: you may pass a string, integer, tuple, or other decimal object to create a new `decimal` object, but if you have a float `n` that you want to make into a `decimal`, pass `str(n)`, *not* bare `n`. Also, `decimal` objects can interact (i.e., be subject to arithmetical operations) with integers, longs, and other `decimal` objects, but not with floats. These restrictions are anything but arbitrary. Decimal numbers have been added to Python exactly to provide the precision and predictability that float lacks: if it was allowed to build a decimal number from a float, or by operating with one, the whole purpose would be defeated. `decimal` objects, on the other hand, can be coerced into other numeric types such as float, long, and int, just as you would expect.

Keep in mind that `decimal` is still floating point, *not* fixed point. If you want fixed point, take a look at Tim Peter's `FixedPoint` at <http://fixedpoint.sourceforge.net/>. Also, no money data type is yet available in Python, although you can look at recipe 3.13 "Formatting Decimals as Currency" to learn how to roll-your-own money formatting on top of `decimal`. Last but not least, it is not obvious (at least not to me), when an intermediate computation produces more digits than the inputs, whether you should keep the extra digits for further intermediate computations, and round only when you're done computing a formula (and are about to display or store a result), or whether you should instead round at each step. Different textbooks suggest different answers. I tend to do the former, simply because it's more convenient.

If you're stuck with Python 2.3, you may still take advantage of the `decimal` module, by downloading and installing it as a third-party extension—see http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html.

See Also

The explanation of floating-point arithmetic in Appendix B of the Python Tutorial at <http://docs.python.org/tut/node15.html>; the Python FAQ at <http://www.python.org/doc/faq/general.html#why-are-floating-point-calculations-so-inaccurate>; Tim Peter's FixedPoint at <http://fixedpoint.sourceforge.net/>; using decimal as currency, see recipe 3.13 "Formatting Decimals as Currency"; decimal is documented in the Python 2.4 *Library Reference* and is available for download to use with 2.3 at <http://cvs.sourceforge.net/viewcvs.py/python/python/dist/src/Lib/decimal.py>; the decimal PEP (Python Enhancement Proposal), PEP 327, is at <http://www.python.org/peps/pep-0327.html>.

3.13 Formatting Decimals as Currency

Credit: Anna Martelli Ravenscroft, Alex Martelli, Raymond Hettinger

Problem

You want to do some tax calculations and display the result in a simple report as Euro currency.

Solution

Use the new decimal module, along with a modified `moneyfmt` function (the original, by Raymond Hettinger, is part of the Python library reference section about decimal):

```
import decimal
""" calculate Italian invoice taxes given a subtotal. """
def italfomat(value, places=2, curr='EUR', sep=',', dp=',', pos='', neg='-',
              overall=10):
    """ Convert Decimal ``value`` to a money-formatted string.
    places: required number of places after the decimal point
    curr:   optional currency symbol before the sign (may be blank)
    sep:    optional grouping separator (comma, period, or blank) every 3
    dp:     decimal point indicator (comma or period); only specify as
            blank when places is zero
    pos:    optional sign for positive numbers: "+", space or blank
    neg:    optional sign for negative numbers: "-", "(", space or blank
    overall: optional overall length of result, adds padding on the
            left, between the currency symbol and digits
    """
    q = decimal.Decimal((0, (1,), -places))          # 2 places --> '0.01'
    sign, digits, exp = value.quantize(q).as_tuple()
    result = []
    digits = map(str, digits)
    append, next = result.append, digits.pop
    for i in range(places):
        if digits:
```

```

        append(next())
    else:
        append('0')
append(dp)
i = 0
while digits:
    append(next())
    i += 1
    if i == 3 and digits:
        i = 0
        append(sep)
while len(result) < overall:
    append(' ')
append(curr)
if sign: append(neg)
else: append(pos)
result.reverse()
return ''.join(result)
# get the subtotal for use in calculations
def getsubtotal(subtin=None):
    if subtin == None:
        subtin = input("Enter the subtotal: ")
    subtotal = decimal.Decimal(str(subtin))
    print "\n subtotal:           ", italformat(subtotal)
    return subtotal
# specific Italian tax law functions
def cnpcalc(subtotal):
    contrib = subtotal * decimal.Decimal('.02')
    print "+ contributo integrativo 2%:      ", italformat(contrib, curr='')
    return contrib
def vatcalc(subtotal, cnp):
    vat = (subtotal+cnp) * decimal.Decimal('.20')
    print "+ IVA 20%:                          ", italformat(vat, curr='')
    return vat
def ritacalc(subtotal):
    rit = subtotal * decimal.Decimal('.20')
    print "-Ritenuta d'acconto 20%:          ", italformat(rit, curr='')
    return rit
def dototal(subtotal, cnp, iva=0, rit=0):
    totl = (subtotal+cnp+iva)-rit
    print "                               TOTALE: ", italformat(totl)
    return totl
# overall calculations report
def invoicer(subtotal=None, context=None):
    if context is None:
        decimal.getcontext().rounding="ROUND_HALF_UP"    # Euro rounding rules
    else:
        decimal.setcontext(context)                      # set to context arg
    subtot = getsubtotal(subtotal)
    contrib = cnpcalc(subtot)
    dototal(subtot, contrib, vatcalc(subtot, contrib), ritacalc(subtot))
if __name__=='__main__':
    print "Welcome to the invoice calculator"
    tests = [100, 1000.00, "10000", 555.55]

```

```

print "Euro context"
for test in tests:
    invoicer(test)
print "default context"
for test in tests:
    invoicer(test, context=decimal.DefaultContext)

```

Discussion

Italian tax calculations are somewhat complicated, more so than this recipe demonstrates. This recipe applies only to invoicing customers within Italy. I soon got tired of doing them by hand, so I wrote a simple Python script to do the calculations for me. I've currently refactored into the version shown in this recipe, using the new `decimal` module, just on the principle that money computations should never, but *never*, be done with binary floats.

How to best use the new `decimal` module for monetary calculations was not immediately obvious. While the decimal arithmetic is pretty straightforward, the options for displaying results were less clear. The `italformat` function in the recipe is based on Raymond Hettinger's `moneyfmt` recipe, found in the `decimal` module documentation available in the Python 2.4 *Library Reference*. Some minor modifications were helpful for my reporting purposes. The primary addition was the `overall` parameter. This parameter builds a decimal with a specific number of overall digits, with whitespace padding between the currency symbol (if any) and the digits. This eases alignment issues when the results are of a standard, predictable length.

Notice that I have coerced the subtotal input `subtin` to be a string in `subtotal = decimal.Decimal(str(subtin))`. This makes it possible to feed floats (as well as integers or strings) to `getsubtotal` without worry—without this, a float would raise an exception. If your program is likely to pass tuples, refactor the code to handle that. In my case, a float was a rather likely input to `getsubtotal`, but I didn't have to worry about tuples.

Of course, if you need to display using U.S. \$, or need to use other rounding rules, it's easy enough to modify things to suit your needs. For example, to display U.S. currency, you could change the `curr`, `sep`, and `dp` arguments' default values as follows:

```

def USformat(value, places=2, curr='$', sep=',', dp='.', pos='', neg='- ',
            overall=10):
    ...

```

If you regularly have to use multiple currency formats, you may choose to refactor the function so that it looks up the appropriate arguments in a dictionary, or you may want to find other ways to pass the appropriate arguments. In theory, the `locale` module in the Python Standard Library should be the standard way to let your code access locale-related preferences such as those connected to money

formatting, but in practice I've never had much luck using `locale` (for this or any other purpose), so that's one task that I'll gladly leave as an exercise to the reader.

Countries often have specific rules on rounding; `decimal` uses `ROUND_HALF_EVEN` as the default. However, the Euro rules specify `ROUND_HALF_UP`. To use different rounding rules, change the context, as shown in the recipe. The result of this change may or may not be obvious, but one should be aware that it *can* make a (small, but legally not negligible) difference.

You can also change the context more extensively, by creating and setting your own context class instance. A change in context, whether set by a simple `getcontext` attribution change, or with a custom context class instance passed to `setcontext(mycontext)`, continues to apply throughout the active thread, until you change it. If you are considering using `decimal` in production code (or even for your own home bookkeeping use), be sure to use the right context (in particular, the correct rounding rules) for your country's accounting practices.

See Also

Python 2.4's *Library Reference* on `decimal`, particularly the section on `decimal.context` and the “recipes” at the end of that section.

3.14 Using Python as a Simple Adding Machine

Credit: Brett Cannon

Problem

You want to use Python as a simple adding machine, with accurate decimal (not binary floating-point!) computations and a “tape” that shows the numbers in an uncluttered columnar view.

Solution

To perform the computations, we can rely on the `decimal` module. We accept input lines, each made up of a number followed by an arithmetic operator, an empty line to request the current total, and `q` to terminate the program:

```
import decimal, re, operator
parse_input = re.compile(r'''(?x) # allow comments and whitespace in the RE
    (\d+\.\d*) # number with optional decimal part
    \s* # optional whitespace
    ([+/*]) # operator
    '$''') # end-of-string
oper = { '+': operator.add, '-': operator.sub,
        '*': operator.mul, '/': operator.truediv,
        }
total = decimal.Decimal('0')
def print_total():
```

```

    print '====\n', total
print """Welcome to Adding Machine:
Enter a number and operator,
an empty line to see the current subtotal,
or q to quit: """
while True:
    try:
        tape_line = raw_input().strip()
    except EOFError:
        tape_line = 'q'
    if not tape_line:
        print_total()
        continue
    elif tape_line == 'q':
        print_total()
        break
    try:
        num_text, op = parse_input.match(tape_line).groups()
    except AttributeError:
        print 'Invalid entry: %r' % tape_line
        print 'Enter number and operator, empty line for total, q to quit'
        continue
    total = oper[op](total, decimal.Decimal(num_text))

```

Discussion

Python’s interactive interpreter is often a useful calculator, but a simpler “adding machine” also has its uses. For example, an expression such as `2345634+2894756-2345823` is not easy to read, so checking that you’re entering the right numbers for a computation is not all that simple. An adding machine’s tape shows numbers in a simple, uncluttered columnar view, making it easier to double check what you have entered. Moreover, the `decimal` module performs computations in the normal, decimal-based way we need in real life, rather than in the floating-point arithmetic preferred by scientists, engineers, and today’s computers.

When you run the script in this recipe from a normal command shell (this script is *not* meant to be run from within a Python interactive interpreter!), the script prompts you once, and then just sits there, waiting for input. Type a number (one or more digits, then optionally a decimal point, then optionally more digits), followed by an operator (`/`, `*`, `-`, or `+`—the four operator characters you find on the numeric keypad on your keyboard), and then press return. The script applies the number to the running total using the operator. To output the current total, just enter a blank line. To quit, enter the letter `q` and press return. This simple interface matches the input/output conventions of a typical simple adding machine, removing the need to have some other form of output.

The `decimal` package is part of Python’s standard library since version 2.4. If you’re still using Python 2.3, visit http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html and download and install the package in whatever form is most conve-

nient for you. `decimal` allows high-precision decimal arithmetic, which is more convenient for many uses (such as any computation involving money) than the binary floating-point computations that are faster on today’s computers and which Python uses by default. No more lost pennies due to hard-to-understand issues with binary floating point! As demonstrated in recipe 3.13 “Formatting Decimals as Currency,” you can even change the rounding rules from the default of `ROUND_HALF_EVEN`, if you really need to.

This recipe’s script is meant to be very simple, so many improvements are possible. A useful enhancement would be to keep the “tape” on disk for later checking. You can do that easily, by adding, just before the loop, a statement to open some appropriate text file for append:

```
tapefile = open('tapefile.txt', 'a')
```

and, just after the `try/except` statement that obtains a value for `tape_line`, a statement to write that value to the file:

```
tapefile.write(tape_line+'\n')
```

If you do want to make these additions, you will probably also want to enrich function `print_total` so that it writes to the “tape” file as well as to the command window, therefore, change the function to:

```
def print_total():
    print '====\n', total
    tapefile.write('====\n' + str(total) + '\n')
```

The `write` method of a file object accepts a string as its argument and does not implicitly terminate the line as the `print` statement does, so we need to explicitly call the `str` built-in function and explicitly add `'\n'` as needed. Alternatively, the second statement in this version of `print_total` could be coded in a way closer to the first one:

```
print >>tapefile, '====\n', total
```

Some people really dislike this `print >>somefile`, syntax, but it can come in handy in cases such as this one.

More ambitious improvements would be to remove the need to press Return after each operator (that would require performing unbuffered input and dealing with one character at a time, rather than using the handy but line-oriented built-in function `raw_input` as the recipe does—see recipe 2.23 “Reading an Unbuffered Character in a Cross-Platform Way” for a cross-platform way to get unbuffered input), to add a `clear` function (or clarify to users that inputting `0*` will zero out the “tape”), and even to add a GUI that looks like an adding machine. However, I’m leaving any such improvements as exercises for the reader.

One important point about the recipe’s implementation is the `oper` dictionary, which uses operator characters (`/`, `*`, `-`, `+`) as keys and the appropriate arithmetic functions

from the built-in module `operator`, as corresponding values. The same effect could be obtained, more verbosely, by a “tree” of `if/elif`, such as:

```
if op == '+':
    total = total + decimal.Decimal(num_text)
elif op == '-':
    total = total - decimal.Decimal(num_text)
elif op == '*':
    <line_annotation>... and so on ...</line_annotation>
```

However, Python dictionaries are very idiomatic and handy for such uses, and they lead to less repetitious and thus more maintainable code.

See Also

`decimal` is documented in the Python 2.4 *Library Reference*, and is available for download to use with 2.3 at http://www.taniquetil.com.ar/facundo/bdvfiles/get_decimal.html; you can read the decimal PEP 327 at <http://www.python.org/peps/pep-0327.html>.

3.15 Checking a Credit Card Checksum

Credit: David Shaw, Miika Keskinen

Problem

You need to check whether a credit card number respects the industry standard Luhn checksum algorithm.

Solution

Luhn mod 10 is the credit card industry’s standard for credit card checksums. It’s not built into Python, but it’s easy to roll our own computation for it:

```
def cardLuhnChecksumIsValid(card_number):
    """ checks to make sure that the card passes a luhn mod-10 checksum """
    sum = 0
    num_digits = len(card_number)
    oddeven = num_digits & 1
    for count in range(num_digits):
        digit = int(card_number[count])
        if not ((count & 1) ^ oddeven):
            digit = digit * 2
        if digit > 9:
            digit = digit - 9
        sum = sum + digit
    return (sum % 10) == 0
```

Discussion

This recipe was originally written for a now-defunct e-commerce application to be used within Zope.

It can save you time and money to apply this simple validation before trying to process a bad or miskeyed card with your credit card vendor, because you won't waste money trying to authorize a bad card number. The recipe has wider applicability because many government identification numbers also use the Luhn (i.e., modulus 10) algorithm.

A full suite of credit card validation methods is available at <http://david.theresistance.net/files/creditValidation.py>

If you're into cool one-liners rather than simplicity and clarity, (a) you're reading the wrong book (the *Perl Cookbook* is a great book that will make you much happier), (b) meanwhile, to keep you smiling while you go purchase a more appropriate oeuvre, try:

```
checksum = lambda a: (
    10 - sum([int(y)*[7,3,1][x%3] for x, y in enumerate(str(a)[::-1])])%10)%10
```

See Also

A good therapist, if you *do* prefer the one-line checksum version.

3.16 Watching Foreign Exchange Rates

Credit: Victor Yongwei Yang

Problem

You want to monitor periodically (with a Python script to be run by *crontab* or as a Windows scheduled task) an exchange rate between two currencies, obtained from the Web, and receive email alerts when the rate crosses a certain threshold.

Solution

This task is similar to other monitoring tasks that you could perform on numbers easily obtained from the Web, be they exchange rates, stock quotes, wind-chill factors, or whatever. Let's see specifically how to monitor the exchange rate between U.S. and Canadian dollars, as reported by the Bank of Canada web site (as a simple CSV (comma-separated values) feed that is easy to parse):

```
import httplib
import smtplib
# configure script's parameters here
thresholdRate = 1.30
smtpServer = 'smtp.freebie.com'
fromaddr = 'foo@bar.com'
```

```

toaddr = 'your@corp.com'
# end of configuration
url = '/en/financial_markets/csv/exchange_eng.csv'
conn = httpplib.HTTPConnection('www.bankofcanada.ca')
conn.request('GET', url)
response = conn.getresponse()
data = response.read()
start = data.index('United States Dollar')
line = data[start:data.index('\n', start)] # get the relevant line
rate = line.split(',')[ -1] # last field on the line
if float(rate) < thresholdRate:
    # send email
    msg = 'Subject: Bank of Canada exchange rate alert %s' % rate
    server = smtplib.SMTP(smtpServer)
    server.sendmail(fromaddr, toaddr, msg)
    server.quit()
conn.close()

```

Discussion

When working with foreign currencies, it is particularly useful to have an automated way of getting the conversions you need. This recipe provides this functionality in a quite simple, straightforward manner. When cron runs this script, the script goes to the site, and gets the CSV feed, which provides the daily noon exchange rates for the previous seven days:

```

Date (m/d/year),11/12/2004,11/15/2004, ... ,11/19/2004,11/22/2004
$Can/US closing rate,1.1927,1.2005,1.1956,1.1934,1.2058,1.1930,
United States Dollar,1.1925,1.2031,1.1934,1.1924,1.2074,1.1916,1.1844
...

```

The script then continues to find the specific currency ('United States Dollar') and reads the last field to find today's rate. If you're having trouble understanding how that works, it may be helpful to break it down:

```

US = data.find('United States Dollar') # find the index of the currency
endofUSline = data.index('\n', US) # find index for that line end
USline = data[US:endofUSline] # slice to make one string
rate = USline.split(',')[ -1] # split on ',' and return last field

```

The recipe provides an email alert when the rate falls below a particular threshold, which can be configured to whatever rate you prefer (e.g., you could change that statement to send you an alert whenever the rate changes outside a threshold range).

See Also

`httpplib`, `smtplib`, and `string` function are documented in the *Library Reference* and *Python in a Nutshell*.