10 Performances and optimizations

> Premature optimization is the root of all evil.
>
> --- Donald Knuth *Structured Programming with go to Statements*

10.1 Data structures

Most computer problems can be solved in an elegant and simple manner, provided that you use the right data structures – and Python provides many data structures to choose from.

Often, there is a temptation to code your own custom data structures – this is invariably a vain, useless, doomed idea. Python almost always has better data structures and code to offer – learn to use them.

For example, everybody uses `dict`, but how many times have you seen code like this:

```
def get_fruits(basket, fruit):
    # A variation is to use "if fruit in basket:"
    try:
        return basket[fruit]
    except KeyError:
```

```
        return set()
```

It's much more easy to use the `get` method already provided by the `dict` structure:

```
def get_fruits(basket, fruit):
    return basket.get(fruit, set())
```

It's not uncommon for people to use basic Python data structures without being aware of all the methods they provide. This is also true for sets – for example:

```
def has_invalid_fields(fields):
    for field in fields:
        if field not in ['foo', 'bar']:
            return True
    return False
```

This can be written without a loop:

```
def has_invalid_fields(fields):
    return bool(set(fields) - set(['foo', 'bar']))
```

The `set` data structures have methods which can solve many problems that would otherwise need to be addressed by writing nested for/if blocks.

There are also more advanced data structures that can greatly reduce the burden of code maintenance. For example, take a look at the following code:

```
def add_animal_in_family(species, animal, family):
    if family not in species:
        species[family] = set()
    species[family].add(animal)


species = {}
add_animal_in_family(species, 'cat', 'felidea')
```

Sure, this code is perfectly valid, but how many times will your program require a variation of the above? Tens? Hundreds?

Python provides the `collections.defaultdict` structure, which solves the problem in an elegant way.

```
import collections


def add_animal_in_family(species, animal, family):
    species[family].add(animal)


species = collections.defaultdict(set)
add_animal_in_family(species, 'cat', 'felidea')
```

Each time that you try to access a non-existent item from your *dict*, the `defaultdict` will use the function that was passed as argument to its constructor to build a new value – instead than raising a `KeyError`. In this case, the `set` function is used to build a new *set* each time we need it.

By the way, the `collections` module offers a few useful data structures that can solve other kinds of problems, such as `OrderedDict` or `Counter`.

It's really important to look for the right data structure in Python, as the correct choice will save you time, and lessen code maintenance.

10.2 Profiling

Python provides a few tools to profile your program. The standard one is `cProfile` and is easy enough to use.

Example 10.1 Using the `cProfile` module

```
$ python -m cProfile myscript.py
         343 function calls (342 primitive calls) in 0.000 seconds
```

```
   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 :0(_getframe)
        1    0.000    0.000    0.000    0.000 :0(len)
      104    0.000    0.000    0.000    0.000 :0(setattr)
        1    0.000    0.000    0.000    0.000 :0(setprofile)
        1    0.000    0.000    0.000    0.000 :0(startswith)
      2/1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 StringIO.py:30(<module>)
        1    0.000    0.000    0.000    0.000 StringIO.py:42(StringIO)
```

The results list indicates the number of calls each function was called, and the time spent on its execution. You can use the `-s` option to sort by other fields; e.g. `-s time` will sort by internal time.

If you've coded in C, as I did years ago, you probably already know the fantastic Valgrind tool, that – among other things – is able to provide profiling data for C programs. The data that it provides can then be visualized by another great tool named KCacheGrind.

You'll be happy to know that the profiling information generated by *cProfile* can easily be converted to a call tree that can be read by *KCacheGrind*. The `cProfile` module has a `-o` option that allows you to save the profiling data, and pyprof2calltree can convert from one format to the other.

Example 10.2 Using *KCacheGrind* to visualize Python profiling data

```
$ python -m cProfile -o myscript.cprof myscript.py
$ pyprof2calltree -k -i myscript.cprof
```

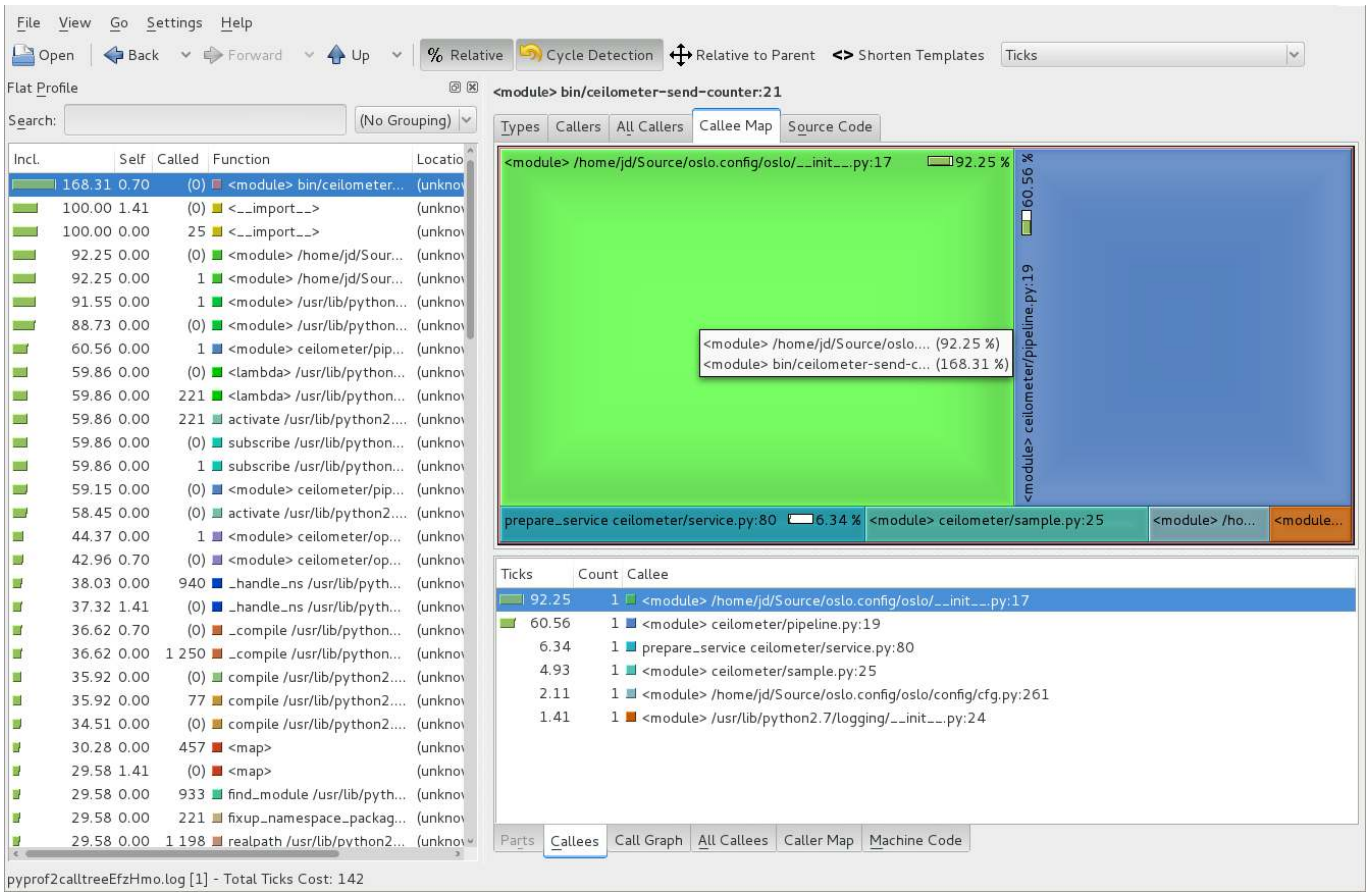

Figure 10.1: KCacheGrind example

This provides a lot of information that will allow you to determine what part of your program might be consuming too much resources.

While this clearly works well for a macroscopic view of your program, it sometimes helps to have a microscopic view of some part of the code. In such a context, I find it better to rely on the `dis` module to find out what's going on behind the scenes. The `dis` module is a disassembler of Python byte code. It's simple enough to use:

```
>>> def x():
...     return 42
...
>>> import dis
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
```

```
              3 RETURN_VALUE
```

The `dis.dis` function disassembles the function that you passed as a parameter, and prints the list of bytecode instructions that are run by the function. It can be useful to understand what's really behind each line of code that you write, in order to be able to properly optimize your code.

The following code defines two functions, each of which does the same thing – concatenates three letters:

```
abc = ('a', 'b', 'c')

def concat_a_1():
    for letter in abc:
            abc[0] + letter

def concat_a_2():
    a = abc[0]
    for letter in abc:
            a + letter
```

Both appear to do exactly the same thing, but if we disassemble them, we'll see that the generated bytecode is a bit different:

```
>>> dis.dis(concat_a_1)
  2           0 SETUP_LOOP              26 (to 29)
              3 LOAD_GLOBAL              0 (abc)
              6 GET_ITER
        >>    7 FOR_ITER                18 (to 28)
             10 STORE_FAST               0 (letter)

  3          13 LOAD_GLOBAL              0 (abc)
             16 LOAD_CONST               1 (0)
```

```
             19 BINARY_SUBSCR
             20 LOAD_FAST                0 (letter)
             23 BINARY_ADD
             24 POP_TOP
             25 JUMP_ABSOLUTE            7
        >>   28 POP_BLOCK
        >>   29 LOAD_CONST               0 (None)
             32 RETURN_VALUE
```

```
>>> dis.dis(concat_a_2)
  2           0 LOAD_GLOBAL              0 (abc)
              3 LOAD_CONST               1 (0)
              6 BINARY_SUBSCR
              7 STORE_FAST               0 (a)

  3          10 SETUP_LOOP              22 (to 35)
             13 LOAD_GLOBAL              0 (abc)
             16 GET_ITER
        >>   17 FOR_ITER                14 (to 34)
             20 STORE_FAST               1 (letter)

  4          23 LOAD_FAST                0 (a)
             26 LOAD_FAST                1 (letter)
             29 BINARY_ADD
             30 POP_TOP
             31 JUMP_ABSOLUTE           17
        >>   34 POP_BLOCK
        >>   35 LOAD_CONST               0 (None)
             38 RETURN_VALUE
```

As you can see, in the second version we store `abc[0]` in a temporary variable be-

fore running the loop. This makes the bytecode executed inside the loop a little smaller, as we avoid having to do the `abc[0]` lookup for each iteration. Measured using `timeit`, the second version is 10% faster than the first one; it takes a whole microsecond less to execute! Obviously this microsecond is not worth the optimization unless you call this function millions of times – but this is kind of insight that the `dis` module can provide.

Whether you should need to rely on such "tricks" as storing the value outside the loop is debatable – ultimately, it should be the compiler's work to optimize this kind of thing. On the other hand, as the language is heavily dynamic, it's difficult for the compiler to be sure that optimization wouldn't result in negative side effects. So be careful when writing your code!

Another wrong habit I've often encountered when reviewing code is the defining of functions inside functions for no reason. This has a cost – as the function is going to be redefined over and over for no reason.

Example 10.3 A function defined in a function, disassembled

```
>> import dis
>>> def x():
...     return 42
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 RETURN_VALUE
>>> def x():
...     def y():
...             return 42
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (<code object y at 0x100ce7e30,  ↩
```

```
    file "<stdin>", line 2>)
              3 MAKE_FUNCTION            0
              6 STORE_FAST               0 (y)

  4           9 LOAD_FAST                0 (y)
             12 CALL_FUNCTION            0
             15 RETURN_VALUE
```

We can see here that it is needlessly complicated, calling `MAKE_FUNCTION`, `STORE_FAST`, `LOAD_FAST` and `CALL_FUNCTION` instead of just `LOAD_CONST`. That requires many more opcodes for no good reason – and function calling in Python is already inefficient.

The only case in which it is required to define a function within a function is when building a function closure, and this is a perfectly identified use case in Python's opcodes.

Example 10.4 Disassembling a closure

```
>>> def x():
...     a = 42
...     def y():
...             return a
...     return y()
...
>>> dis.dis(x)
  2           0 LOAD_CONST               1 (42)
              3 STORE_DEREF              0 (a)

  3           6 LOAD_CLOSURE             0 (a)
              9 BUILD_TUPLE              1
             12 LOAD_CONST               2 (<code object y at 0x100d139b0,  ↩
```

```
                file "<stdin>", line 3>)
             15 MAKE_CLOSURE             0
             18 STORE_FAST               0 (y)

  5          21 LOAD_FAST                0 (y)
             24 CALL_FUNCTION            0
             27 RETURN_VALUE
```

10.3 Ordered list and bisect

When manipulating large lists, the use of sorted lists has a few advantages over non-sorted lists – for example, sorted lists have a retrieve time of *O(log n)*.

A couple of times, however, I've seen people trying to implement their own data structures and algorithms to handle such cases. This is a bad idea – you shouldn't spend time on problems already solved.

Firstly, Python provides a `bisect` module which contains a bisection algorithm. It's easy enough to use:

Example 10.5 Usage of `bisect`

```
>>> farm = sorted(['haystack', 'needle', 'cow', 'pig'])
>>> bisect.bisect(farm, 'needle')
3
>>> bisect.bisect_left(farm, 'needle')
2
>>> bisect.bisect(farm, 'chicken')
0
>>> bisect.bisect_left(farm, 'chicken')
0
>>> bisect.bisect(farm, 'eggs')
```

```
1
>>> bisect.bisect_left(farm, 'eggs')
1
```

The `bisect` function allows you to retrieve the index where a new list element should be inserted, while keeping the list sorted.

If you wish to insert the element immediately, the `bisect` module provides the `insort_left` and `insort_right` functions that do exactly that.

Example 10.6 Usage of `bisect.insort`

```
>>> farm
['cow', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'eggs')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig']
>>> bisect.insort(farm, 'turkey')
>>> farm
['cow', 'eggs', 'haystack', 'needle', 'pig', 'turkey']
```

You can then use these functions to create a list that is always sorted:

Example 10.7 A SortedList implementation

```
import bisect


class SortedList(list):
    def __init__(self, iterable):
        super(SortedList, self).__init__(sorted(iterable))

    def insort(self, item):
        bisect.insort(self, item)
```

```
    def index(self, value, start=None, stop=None):
        place = bisect.bisect_left(self[start:stop], value)
        if start:
            place += start
        end = stop or len(self)
        if place < end and self[place] == value:
            return place
        raise ValueError("%s is not in list" % value)
```

Obviously, one shouldn't use the direct functions `append` or `extend` on this list – or the list will no longer be sorted.

Many Python libraries exist which implement various versions of the above code – and many more data types, such as binary or red-black tree structures. The `blist` and `bintree` Python packages contain code that you can be use for these purposes, rather than implementing and debugging your own version.

10.4 Namedtuple and slots

Sometimes it's useful to have the ability to create very simple objects which only possess a few fixed attributes. A simple implementation would be something along these lines:

```
class Point(object):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

This definitely gets the job done – however, there is a downside to this approach: it creates a class which inherits from object. In using this *Point* class, you be instantiating objects.

One property of such *objects* in Python, is that they store all of their attributes inside a dictionary; this dictionary is itself stored in the `__dict__` attribute:

```
>>> p = Point(1, 2)
>>> p.__dict__
{'y': 2, 'x': 1}
>>> p.z = 42
>>> p.z
42
>>> p.__dict__
{'y': 2, 'x': 1, 'z': 42}
```

The advantage is that you can add as many attributes as you want to an object. The drawback, however, is that using a dictionary to store these attributes is quite expensive in terms of memory – you need to store the object, the keys, the value references, etc. It's slow to create and slow to manipulate, with a high memory cost. Consider the following simple class:

```
[source,python]
class Foobar(object):
    def __init__(self, x):
        self.x = x
```

Let's check the memory usage using the `memory_profiler` Python package:

```
$ python -m memory_profiler object.py
Filename: object.py

Line #    Mem usage    Increment   Line Contents
================================================
     5                             @profile
     6     9.879 MB     0.000 MB   def main():
```

```
     7    50.289 MB    40.410 MB       f = [ Foobar(42) for i in range ↩
        (100000) ]
```

Therefore, it exists a way to use objects without this default behaviour. Classes in Python can define a `__slots__` attribute that will list the only attributes allowed for instances of this class. The power of this is that instead of allocating a whole dictionary object to store all of the object attributes, they can now be stored in a `list` object. If you go through the CPython source code and take a look at the `Objects/typeobject.c` file, it is quite easy to understand what Python does in this case. Here is a cut down version of the function which handles this:

```
static PyObject *
type_new(PyTypeObject *metatype, PyObject *args, PyObject *kwds)
{
    […]
    /* Check for a __slots__ sequence variable in dict, and count it */
    slots = _PyDict_GetItemId(dict, &PyId___slots__);
    nslots = 0;
    if (slots == NULL) {
        if (may_add_dict)
            add_dict++;
        if (may_add_weak)
            add_weak++;
    }
    else {
        /* Have slots */
        /* Make it into a tuple */
        if (PyUnicode_Check(slots))
            slots = PyTuple_Pack(1, slots);
        else
            slots = PySequence_Tuple(slots);
```

```
        /* Are slots allowed? */
        nslots = PyTuple_GET_SIZE(slots);
        if (nslots > 0 && base->tp_itemsize != 0) {
            PyErr_Format(PyExc_TypeError,
                         "nonempty __slots__ "
                         "not supported for subtype of '%s'",
                         base->tp_name);
            goto error;
        }
        /* Copy slots into a list, mangle names and sort them.
           Sorted names are needed for __class__ assignment.
           Convert them back to tuple at the end.a
        */
        newslots = PyList_New(nslots - add_dict - add_weak);
        if (newslots == NULL)
            goto error;
        if (PyList_Sort(newslots) == -1) {
            Py_DECREF(newslots);
            goto error;
        }
        slots = PyList_AsTuple(newslots);
        Py_DECREF(newslots);
        if (slots == NULL)
            goto error;
    }
    /* Allocate the type object */
    type = (PyTypeObject *)metatype->tp_alloc(metatype, nslots);
    […]
    /* Keep name and slots alive in the extended type object */
    et = (PyHeapTypeObject *)type;
```

```
    Py_INCREF(name);
    et->ht_name = name;
    et->ht_slots = slots;
    slots = NULL;
    […]
    return (PyObject *)type;
```

As you can see, Python converts the content of `__slots__` into a tuple, then a `list` that it builds and sorts, before converting it back into a tuple to use and store it in the class. This way, Python can retrieve the values quickly, without having to allocate and use an entire dictionary.

It's easy enough to declare such a class:

Example 10.8 A class declaration using `__slots__`

```
class Foobar(object):
    __slots__ = 'x'

    def __init__(self, x):
        self.x = x
```

We can easily compare the memory usage of the two approaches using the `memory_profiler` Python package:

Example 10.9 Memory usage of objects using `__slots__`

```
% python -m memory_profiler slots.py
Filename: slots.py

Line #    Mem usage    Increment   Line Contents
================================================
     7                             @profile
     8     9.879 MB     0.000 MB   def main():
```

```
     9    21.609 MB    11.730 MB       f = [ Foobar(42) for i in range ↩
        (100000) ]
```

So it seems that by using the `__slots__` attribute of Python classes, we can halve our memory usage – this means that when creating a large amount of simple objects, the `__slots__` attribute is an effective and efficient choice. However, the technique shouldn't be misused in order to perform static typing or the like. This isn't in the spirit of Python programs.

Due to the fixed nature of the attribute list, it's easy enough to imagine classes where the attributes listed would always have a value, and where the fields would always be sorted in some way.

That's exactly the nature of the **`namedtuple`** class from the **`collection`** module. It allows us to dynamically create a class that will inherit from **`tuple`**, therefore sharing its characteristics – such as being immutable, and having a fixed number of entries. What `namedtuple` provides is the ability to retrieve the tuple elements by referencing a named attribute, rather than just referencing by index.

Example 10.10 Declaring a class using **`namedtuple`**

```
>>> import collections
>>> Foobar = collections.namedtuple('Foobar', ['x'])
>>> Foobar = collections.namedtuple('Foobar', ['x', 'y'])
>>> Foobar(42, 43)
Foobar(x=42, y=43)
>>> Foobar(42, 43).x
42
>>> Foobar(42, 43).x = 44
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
>>> Foobar(42, 43).z = 0
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Foobar' object has no attribute 'z'
>>> list(Foobar(42, 43))
[42, 43]
```

Since a class like this would inherit from `tuple`, we can easily convert it to a list. We can't change or add any attributes on objects of this class, because on one hand it inherits from `tuple`, and also because the `__slots__` value is set to an empty tuple – thereby avoiding the creating of the `__dict__`.

Example 10.11 Memory usage of a class built from `collections.namedtuple`

```
% python -m memory_profiler namedtuple.py
Filename: namedtuple.py

Line #    Mem usage    Increment   Line Contents
================================================
     4                             @profile
     5     9.895 MB     0.000 MB   def main():
     6    23.184 MB    13.289 MB       f = [ Foobar(42) for i in range ↩
        (100000) ]
```

Therefore, usage of the `namedtuple` class factory is as almost as efficient as using an object with `__slots__`, the only difference being that it is compatible with the `tuple` class. It can therefore be passed to many native Python functions and libraries that expect an iterable type as an argument. It also enjoys the various optimizations that exist for tuples [1].

`namedtuple` also provides a few extra methods that, even if prefixed by an underscore, are actually intended to be public. `_asdict` can convert the `namedtuple` to

[1] For example, tuples smaller than `PyTuple_MAXSAVESIZE` (20 by default) will use a faster memory allocator in CPython

a `dict` instance, `_make` allows us to convert an existing iterable object to this class, and `_replace` returns a new instance of the object with some fields replaced.

10.5 Memoization

Memoization is a technique used to speed up function calls by caching their result. The results can be cached only if the function is pure – meaning that it has no side effects or outputs, and that it does not depend on any global state.

A trivial function that can be memoized is the sine function `sin`.

Example 10.12 A basic memoization technique

```
>>> import math
>>> _SIN_MEMOIZED_VALUES = {}
>>> def memoized_sin(x):
...     if x not in _SIN_MEMOIZED_VALUES:
...         _SIN_MEMOIZED_VALUES[x] = math.sin(x)
...     return _SIN_MEMOIZED_VALUES[x]
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965}
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin(2)
0.9092974268256817
>>> _SIN_MEMOIZED_VALUES
{1: 0.8414709848078965, 2: 0.9092974268256817}
>>> memoized_sin(1)
0.8414709848078965
>>> _SIN_MEMOIZED_VALUES
```

```
{1: 0.8414709848078965, 2: 0.9092974268256817}
```

The first time that `memoized_sin` is called with an argument that is not stored in `_SIN_MEMOIZED_VALUES`, the value will be computed and stored in this dictionary. Later on, if we call the function with the same value again, the result will be retrieved from the dictionary rather than computed again. While `sin` is a function which computes very quickly, this may not be true of some advanced functions which involve more complicated computations.

If you've already read about decorators (if not, go to Section 7.1), you must be thinking that there is a perfect opportunity to use them here – and you'd be right. PyPI lists a few implementations of memoization through decorators, from very simple cases to the most complex and complete.

Starting with Python 3.3, the **`functools`** module provides a LRU (Least-Recently-Used) cache decorator. This provides the same functionality as the memoization described here, but with the benefit that it limits the number of entries in the cache, removing the least recently used one when the cache size reaches its maximum size.

The module also provides statistics on cache hits, misses, etc. In my opinion, these are a must-haves when implementing such a cache. There's no point in using memoization – or any caching technique – if you are unable to meter its usage and usefulness.

Here's an example of the `memoized_sin` function above, using `functools.lru_cache`:

Example 10.13 Using `functools.lru_cache`

```
>>> import functools
>>> import math
>>> @functools.lru_cache(maxsize=2)
... def memoized_sin(x):
...     return math.sin(x)
```

```
...
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(2)
0.9092974268256817
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=1, maxsize=2, currsize=1)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=2, maxsize=2, currsize=2)
>>> memoized_sin(4)
-0.7568024953079282
>>> memoized_sin.cache_info()
CacheInfo(hits=1, misses=3, maxsize=2, currsize=2)
>>> memoized_sin(3)
0.1411200080598672
>>> memoized_sin.cache_info()
CacheInfo(hits=2, misses=3, maxsize=2, currsize=2)
>>> memoized_sin.cache_clear()
>>> memoized_sin.cache_info()
CacheInfo(hits=0, misses=0, maxsize=2, currsize=0)
```

10.6 PyPy

PyPy is an efficient implementation of the Python language which complies with standards. Indeed, the canonical implementation of Python, CPython – so called

because it's written in C – can be very slow. The idea behind PyPy was to write a Python interpreter in Python itself. In time it evolved to be written in RPython, which is a restricted subset of the Python language.

RPython places constraints on the Python language in such a way that a variable's type can be inferred at compile time. The RPython code is translated to C code that is compiled to build the interpreter – RPython could of course be used to implement other languages than Python.

What's interesting in PyPy, besides the technical challenge, is that it is now at a stage where it can act as a faster replacement for CPython. PyPy has a JIT (Just-In-Time) compiler built-in – long story short, it allows the code to be run in a faster way by combining the speed of compiled code with the flexibility of interpretation.

How fast? That depends, but for pure algorithmic code it is **much** faster. For more general code, PyPy claims to achieve 3 times the speed, most of the time. Though don't start dreaming too much about it yet – PyPy also has some of the CPython limitations, such as the hated GIL. [2]

While not being a strict optimization technique, targeting PyPy as one of your supported Python implementations is probably a good idea. Achieving this goal requires the same kind of coding policy that is required for support of other Python versions – basically, you need to make sure that you are testing your software under PyPy like you do under CPython. *tox* (see Section 6.7) supports the building of virtual environments using PyPy, just as it does for CPython 2 or CPython 3, so it should be pretty straightforward to put this in place.

Doing so at the beginning of the project will make sure that there's not too much work to do at a later stage if you wish to be able to run your software with PyPy.

[2] Global Interpreter Lock

Note

For the Hy project, we successfully adopted such a strategy from the beginning. Hy always has supported PyPy and all CPython versions without much trouble. On the other hand, we failed to do so in all of our OpenStack projects, and we are now blocked by various code paths and dependencies that don't work on PyPy for various reasons, as they weren't fully tested in the early stages.

PyPy is compatible with Python 2.7, and its JIT compiler works on 32- and 64-bit, x86 and ARM architectures, and under various operating systems (Linux, Windows, Mac OS X…). Support for Python 3 is underway.

10.7 Achieving zero copy with the buffer protocol

Often programs have to deal with a huge amount of data in the form of large arrays of bytes. Handling such a large amount of data in strings can be very ineffective once you start manipulating it by copying, slicing, and modifying them.

Let's consider a small program which reads a large file of binary data, and copies it partially into another file. To examine out our memory usage, we will use memory_profiler, a nice Python package that allows us to see the memory usage of a program line by line.

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = content[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
```

```
        target.write(content_to_write)


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

We then run the above program using *memory_profiler*:

```
$ python -m memory_profiler memoryview/copy.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy.py

Mem usage    Increment   Line Contents
================================================
                         @profile
 9.883 MB     0.000 MB   def read_random():
 9.887 MB     0.004 MB       with open("/dev/urandom", "rb") as source:
19.656 MB     9.770 MB           content = source.read(1024 * 10000) ❶
29.422 MB     9.766 MB           content_to_write = content[1024:] ❷
29.422 MB     0.000 MB       print("Content length: %d, content to write  ↩
   length %d" %
29.434 MB     0.012 MB             (len(content), len(content_to_write)))
29.434 MB     0.000 MB       with open("/dev/null", "wb") as target:
29.434 MB     0.000 MB           target.write(content_to_write)
```

❶ We are reading 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB of memory to store this data as a string.

❷ We copy the entire block of data minus the first KB – because we won't be writing to that first KB to the target file.

What's interesting in this example is that, as you can see, the memory usage of the program is increased by about 10 MB when building the variable *content_to_write*.

In fact, the slice operator is copying the entirety of *content*, minus the first KB, into a new string object.

When dealing with large data, performing this kind of operation on large byte arrays is going to be a disaster. If you happen to have written C code already, you know that using *memcpy()* has a significant cost, both in term of memory usage and in terms of general performance: copying memory is slow.

But as a C programmer you'll also know that strings are arrays of characters, and that nothing stops you from looking at only part of this array without copying it, through the use of basic pointer arithmetic [3].

This is possible in Python using objects which implement the **buffer protocol**. The buffer protocol is defined in PEP 3118, which explains the C API used to provide this protocol to various types, such as strings.

When an object implements this protocol, you can use the **memoryview** class constructor on it to build a new *memoryview* object that will reference the original object memory.

Here's an example:

```
>>> s = b"abcdefgh"
>>> view = memoryview(s)
>>> view[1]
98 ❶
>>> limited = view[1:3]
<memory at 0x7fca18b8d460>
>>> bytes(view[1:3])
b'bc'
```

❶ This is the ASCII code for the letter *b*.

[3] Assuming that the entire string is in a contiguous memory area.

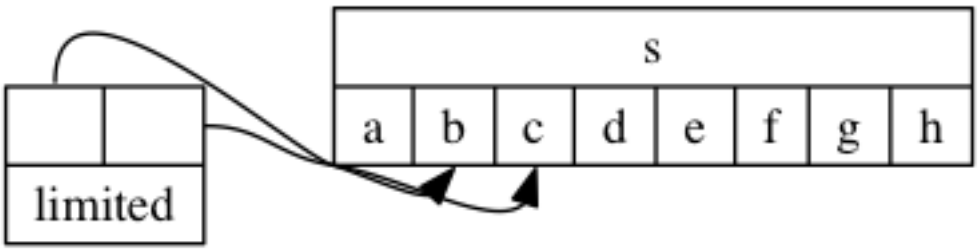

Figure 10.2: Using slice on *memoryview* objects

In this case, we are going to make use of the fact that the `memoryview` object's slice operator itself returns a `memoryview` object. That means it does *not* copy any data, but merely references a particular slice of it.

With this in mind, we now can rewrite the program, this time referencing the data we want to write using a *memoryview* object.

```
@profile
def read_random():
    with open("/dev/urandom", "rb") as source:
        content = source.read(1024 * 10000)
        content_to_write = memoryview(content)[1024:]
    print("Content length: %d, content to write length %d" %
          (len(content), len(content_to_write)))
    with open("/dev/null", "wb") as target:
        target.write(content_to_write)

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

And this program will have half the memory usage of the first version:

```
$ python -m memory_profiler memoryview/copy-memoryview.py
Content length: 10240000, content to write length 10238976
Filename: memoryview/copy-memoryview.py
```

```
Mem usage    Increment   Line Contents
================================================
                         @profile
 9.887 MB     0.000 MB   def read_random():
 9.891 MB     0.004 MB       with open("/dev/urandom", "rb") as source:
19.660 MB     9.770 MB           content = source.read(1024 * 10000) ❶
19.660 MB     0.000 MB           content_to_write = memoryview(content) ↩
   [1024:] ❷
19.660 MB     0.000 MB       print("Content length: %d, content to write  ↩
   length %d" %
19.672 MB     0.012 MB             (len(content), len(content_to_write)))
19.672 MB     0.000 MB       with open("/dev/null", "wb") as target:
19.672 MB     0.000 MB           target.write(content_to_write)
```

❶ We are reading 10 MB from */dev/urandom* and not doing much with it. Python needs to allocate around 10 MB of memory to store this data as a string.

❷ We reference the entire block of data minus the first KB – because we won't be writing to that first KB to the target file. No copying means that no more memory is used!

This kind of trick is especially useful when dealing with sockets. As you may know, when data is sent over a socket, it might not send all the data in a single call. A simple implementation would be to write:

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000) ❶
while data:
```

```
    sent = s.send(data)
    data = data[sent:] ❷
```

❶ Build a bytes object with more than 100 millions times the letter *a*.

❷ Remove the first *sent* bytes sent.

Obviously, using such a mechanism, you are going to copy the data over and over until the socket has sent everything. Using *memoryview*, we can achieve the same functionality without copying data – hence, zero copy:

```
import socket
s = socket.socket(…)
s.connect(…)
data = b"a" * (1024 * 100000) ❶
mv = memoryview(data)
while mv:
    sent = s.send(mv)
    mv = mv[sent:] ❷
```

❶ Build a bytes object with more than 100 millions times the letter *a*.

❷ Build a new memoryview object pointing to the data which remains to be sent.

This won't copy anything, and won't use any more memory than the 100 MB initially needed for our *data* variable.

We've now seen memoryview objects used to write data efficiently, but the same method can also be used to **read** data. Most I/O operations in Python know how to deal with objects implementing the buffer protocol. They can read from it, but also write to it. In this case, we don't need *memoryview* objects – we can just ask an I/O function to write into our pre-allocated object:

```
>>> ba = bytearray(8)
>>> ba
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00')
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba)
...
8
>>> ba
bytearray(b'`m.z\x8d\x0fp\xa1')
```

With such techniques, it's easy to pre-allocate a buffer (as you would do in C to mitigate the number of calls to *malloc()*) and fill it at your convenience. Using *memoryview*, you can even place data at any point in the memory area:

```
>>> ba = bytearray(8)
>>> ba_at_4 = memoryview(ba)[4:] ❶
>>> with open("/dev/urandom", "rb") as source:
...     source.readinto(ba_at_4) ❷
...
4
>>> ba
bytearray(b'\x00\x00\x00\x00\x0b\x19\xae\xb2')
```

❶ We reference the *bytearray* from offset 4 to its end.

❷ We write the content of /dev/urandom from offset 4 to the end of the *bytearray*, effectively reading 4 bytes only.

Tip

Both the objects in the *array* module and the functions in the *struct* module can handle the buffer protocol correctly, and can therefore perform efficiently when targeting zero copy.

10.8 Interview with Victor Stinner

Victor is a long time Python hacker, a core contributor and the author of many Python modules. He recently authored PEP 454, which proposes a new `tracemalloc` module to trace memory block allocation inside Python, and also wrote a simple AST optimizer.

What's a good starting strategy to optimize Python code?

Well, the strategy is the same in Python as in other languages. First you need a well-defined use case, in order to get a stable and reproducible benchmark. Without a reliable benchmark, trying different optimizations may result in a wasting time and premature optimizations. Useless optimizations may make the code worse, less readable, or even slower. A useful optimization must speed the program up by at least 5%.

If a specific part of the code is identified as being "slow", a benchmark should be prepared on this code. A benchmark on a short function is usually called a "micro-benchmark". The speedup should be at least 20%, maybe 25%, to justify an optimization on a micro-benchmark.

It may be interesting to run a benchmark on different computers, different operating systems, different compilers. For example, performances of *realloc()* may vary between Linux and Windows. Even if it should be avoided, sometimes, the implementation may depend on the platform.

There's a lot of different tools around for profiling or optimizing Python code; what are your weapons of choice?

Python 3.3 has a new `time.perf_counter()` function to measure elapsed time for a benchmark. It has the best resolution available.

A test should be run more than once; 3 times is a minimum, 5 may be enough. Repeating a test fills disk cache and CPU caches. I prefer to keep the minimum timing, other developers prefer the geometric mean.

For micro-benchmarks, the timeit module is easy to use and gives results quickly, but the results are not reliable using default parameters. Tests should be repeated manually to get stable results.

Optimizing can take a lot of time, so it's better to focus on functions which use the most CPU power. To find these functions, Python has `cProfile` and profile modules which record the amount of time spent in each function.

What are the interesting Python tricks to know that could improve performance?

The standard library should be reused as much as possible – it's well tested, and also usually efficient. Python built-in types are implemented in C and have good performance. Use the correct container to get the best performance; Python provides many different kind of containers – dict, list, deque, set, etc.

There are some hacks to optimize Python, but they should be avoided because they make the code less readable in exchange for only a minor speed-up.

The Zen of Python (PEP 20) says "There should be one – and preferably only one – obvious way to do it." In practice, there are different ways to write Python code, and performances are not the same. Only trust benchmarks on your use case.

In which areas does Python have poor performance? Which areas should

be used with care?

In general, I prefer not to worry about performance while developing a new application. Premature optimization is the root of all evil. When slow functions are identified, the algorithm should be changed. If the algorithm and the container types are well chosen, it's possible to rewrite short functions in C to get best performances.

A bottleneck in CPython is the Global Interpreter Lock known as the "GIL". Two threads cannot execute Python bytecode at the same time. However, this limitation only matters if two threads are executing pure Python code. If most processing time is spent in function calls, and these functions release the GIL, then the GIL is not the bottleneck. For example, most I/O functions release the GIL.

The multiprocessing module can easily be used to workaround the GIL. Another option, more complex to implement, is to write asynchronous code. Twisted, Tornado and Tulip projects, which are network-oriented libraries, make use of this technique.

What "mistakes" that contribute to poor performance do you see most often?

When Python is not well understood, inefficient code can be written. For example, I have seen `copy.deepcopy()` misused, when no copy was required.

Another performance-killer is an inefficient data structure. With less than one hundred items, the container type has no impact on performance. With more items, the complexity of each operation (add, get, delete) and it's effects must be known.