# NumPy & co

This lesson discusses a couple of other useful Python packages that you can use depending on your requirements.

WE'LL COVER THE FOLLOWING



Beyond NumPy, there are several other Python packages that are worth a look because they address similar yet different class of problems using different technology (compilation, virtual machine, just in time compilation, GPU, compression, etc.). Depending on your specific problem and your hardware, one package may be better than the other.

Let's illustrate their usage using a very simple example where we want to compute an expression based on two float vectors:

```
import numpy as np
a = np.random.uniform(0, 1, 1000).astype(np.float32) #stores samples from a uniform distribut
b = np.random.uniform(0, 1, 1000).astype(np.float32) #stores samples from a uniform distribut
c = 2*a + 3*b
```

## NumExpr #

The numexpr package supplies routines for the fast evaluation of array expressions element-wise by using a vector-based virtual machine. It's comparable to SciPy's weave package, but doesn't require a separate compile step of C or C++ code.

```
import numpy as np
import numexpr as ne

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
c = ne.evaluate("2*a + 3*b")
```

## Cython

Cython is an optimizing static compiler for both the Python programming language and the extended Cython programming language (based on Pyrex). It makes writing C extensions for Python as easy as Python itself.

```
import numpy as np

def evaluate(a,b):
  cdef int i
  cdef np.ndarray c = np.zeros_like(a)
  for i in range(a.size):
    c[i] = 2*a[i] + 3*b[i]
  return c

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
a = np.ndarray
b = np.ndarray
c = evaluate(a, b)
```

## Numba

Numba gives you the power to speed up your applications with high-performance functions written directly in Python. With a few annotations, array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C, C++ and Fortran, without having to switch languages or Python interpreters.

```
from numba import jit
import numpy as np

@jit
def evaluate(np.ndarray a, np.ndarray b):
  c = np.zeros_like(a)
  for i in range(a.size):
    c[i] = 2*a[i] + 3*b[i]
  return c

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
c = evaluate(a, b)
```

## Theano

Theano is a Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently. Theano features tight integration with NumPy, transparent use of a GPU, efficient symbolic differentiation, speed and stability optimizations, dynamic C code generation and extensive unit-testing and self-verification.

```
import numpy as np
import theano.tensor as T

x = T.fvector('x')
y = T.fvector('y')
z = 2*x + 3*y
f = function([x, y], z)

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
c = f(a, b)
```

## PyCUDA

PyCUDA lets you access Nvidia's CUDA parallel computation API from Python.

```
import numpy as np
import pycuda.autoinit
import pycuda.driver as drv
from pycuda.compiler import SourceModule

mod = SourceModule("""
  __global__ void evaluate(float *c, float *a, float *b)
  {
    const int i = threadIdx.x;
    c[i] = 2*a[i] + 3*b[i];
  }
""")

evaluate = mod.get_function("evaluate")

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
c = np.zeros_like(a)

evaluate(drv.Out(c), drv.In(a), drv.In(b), block=(400,1,1), grid=(1,1))
```

## PyOpenCL

PyOpenCL lets you access GPUs and other massively parallel computing

PyOpenCL lets you access GPUs and other massively parallel computing devices from Python.

```
import numpy as np
import pyopencl as cl

a = np.random.uniform(0, 1, 1000).astype(np.float32)
b = np.random.uniform(0, 1, 1000).astype(np.float32)
c = np.empty_like(a)

ctx = cl.create_some_context()
queue = cl.CommandQueue(ctx)

mf = cl.mem_flags
gpu_a = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=a)
gpu_b = cl.Buffer(ctx, mf.READ_ONLY | mf.COPY_HOST_PTR, hostbuf=b)

evaluate = cl.Program(ctx, """
  __kernel void evaluate(__global const float *gpu_a;
                         __global const float *gpu_b;
                         __global       float *gpu_c)
  {
    int gid = get_global_id(0);
    gpu_c[gid] = 2*gpu_a[gid] + 3*gpu_b[gid];
  }
""").build()

gpu_c = cl.Buffer(ctx, mf.WRITE_ONLY, a.nbytes)
evaluate.evaluate(queue, a.shape, None, gpu_a, gpu_b, gpu_c)
cl.enqueue_copy(queue, c, gpu_c)
```

Solve this Quiz!

1 Which of the following supplies routines for faster evaluation of array expressions?

COMPLETED 0%