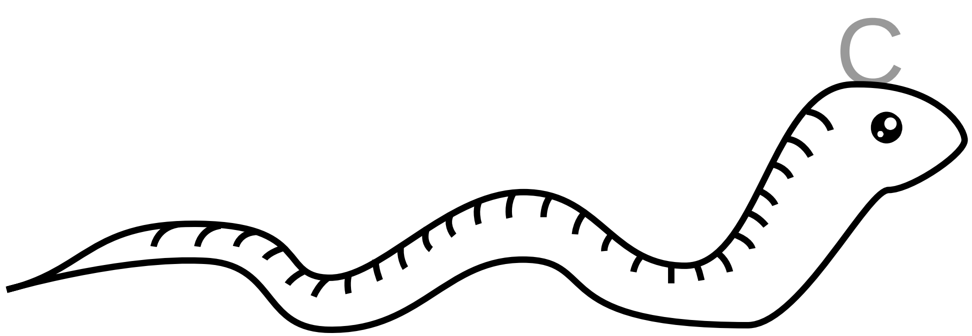
Python 3 for scientific computing

Lecture 8, 28.3.2018
High-performance computing (HPC) in Python

Juha Jeronen
juha.jeronen@tut.fi



Spring 2018, TUT, Tampere
RAK-19006 Various Topics of Civil Engineering

TAMPERE UNIVERSITY OF TECHNOLOGY

HPC in Python

- Python itself is a (very) high-level language, and (comparatively speaking) can be slow.
- Performance-critical computations require acceleration tools, such as:

  - ***NumPy***, ***SciPy***
    - *NumPy is already an acceleration tool* (much faster than Python loops), if your math is expressed in the tensor formalism. However, single-core only (unless multicore BLAS).
    - Similarly, *SciPy already provides excellent performance* for standard math problems, simply by using LAPACK et al.

  ⚠ Cython not to be confused with CPython, the de-facto standard Python interpreter.

☆ - ***Cython***: *C-Extensions for Python*
    - Statically compiled Python with extra syntax for easy embedding of fast C-like code.
    - Creates native extension modules that can be **import**ed like any Python modules.
    - For custom low-level code, and for creating Python interfaces to existing low-level libraries.

  - ***Numexpr***: *Fast numerical array expression evaluator*
    - Further accelerates NumPy. Optimizes memory use, uses multiple cores automatically.

  - ***Numba***: JIT compiler
    - *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.*

  - ***Theano***
    - *A Python library that allows you to define, optimize, and evaluate mathematical expressions involving multi-dimensional arrays efficiently.*
    - No longer developed.

Cython

- Compiled language.
  - A superset of Python; can use a Python program as a starting point.
  - However, looking at Cython as just a static compiler for Python is missing the point.
  - **Language basics** for new users.

Static compilation means *offline, before program execution*. Contrast *dynamic compilation*. See also *incremental comp.*.

- **Two main use cases**:

  - ***FFI*** (***foreign function interface***), to connect Python code to libraries written in C or C++.
    - See *simple-cython-example* and *scikit-sparse* for examples of Cython for FFI.
    - Other possibilities: CFFI (example in *sparseqr*), and *ctypes* in the standard library.
    - If you need a Fortran FFI instead, see F2PY.

☆ - **Acceleration**, by allowing low-level algorithms to be implemented and manually tuned.
    - The level of abstraction can be closer to the underlying hardware than in pure Python. In effect, Cython transforms Python into a *wide-spectrum language*.
    - Especially good for exploiting multiple cores: number-crunching code that needs only NumPy arrays does not need to hold the GIL.

- Compiles to C, which is then compiled to machine language via GCC.
  - The result is an extension module, which can be imported just like any Python module.
  - The compiled code skips the Python interpreter, calling directly into the Python C API. This is slightly faster, but not much.
  - Number-crunching loops accessing only NumPy arrays *may compile to native C code, skipping Python altogether*! This may be **up to two orders of magnitude faster**.
    - Ideal for inherently serial algorithms, or when (part of) an algorithm is much easier to express as a traditional loop over array elements rather than as a tensor operation.
  - The C-like extra syntax has some differences to the actual C language.

Cython

- **Advantages**:
  - Approachable to Python users: just Python, with some extra syntax.
  - Number-crunching loops compiled to machine language, can be very fast (if the algorithm is).
  - True parallelism for custom multithreaded number-crunching in Python.
    - Also extremely easy to leverage OpenMP for parallel loops over NumPy arrays.
  - Excellent integration with Python.
    - Just import your Cython module, no need to care it's written in another language.
  - Easy-ish tool for FFI. (So is CFFI.)

- **Drawbacks**:
  - Limited support in IDEs and text editors, which tend to focus on pure Python. Acceleration of Python code usually a concern only in the scientific computing community.
    - Spyder has Cython syntax highlighting, but no graphical debugger for Cython.
    - PyCharm has full Cython support, but only in the commercial edition.
    - Cython syntax highlighting data file for gedit (actually, for GtkSourceView that it uses).
  - Can be somewhat clumsy:
    - No REPL: must make a module, compile, and import it.
      - Not an inherent limitation of compiled languages; just a common choice in the C family. Python has a REPL, although it internally uses a compiler. Lisps have a REPL; some of them compile to machine code.
    - Need to create and maintain a compile script (*setup.py*).
    - Very basic static type system, just like in C; often gets in your way instead of being helpful.
    - Binaries are platform-dependent; the distribution process can be much more involved.
  - Smaller community, compared to pure Python.
    - May need to look at the generated C code to analyze what exactly is going wrong, instead of just asking the Internet.
      - All input Cython code is included as comments, though, so it's easy to find the right spot.
  - To do (or understand) some things, may need to know a bit of C.

Cython

- **Simple example**: a naive ddot, let's call it ***ddot.pyx***:

```
def ddot(double [::1] a, double [::1] b):    # vector-vector dot product, double precision
    cdef unsigned int k
    cdef unsigned int n = a.shape[0]    # memoryviews have a .shape, just like NumPy arrays
    cdef double out = 0.0

    for k in range(n):          # Becomes a C loop when compiled. Cython knows the datatypes
        out[k] += a[k] * b[k]   # and memory layouts, and no Python objects are used here.

    return out
```

Memoryviews don't support out[:] = a[:] * b[:], so we loop explicitly.

- New keywords **double**, **unsigned**, **int**, **cdef**, **nogil**
  - The IEEE-754 double precision float datatype is called **double**, like in C
  - **int** is a native integer, either 32 or 64 bits, depending on platform
    - **unsigned** means no sign bit is used; values are 0, 1, …, $2^m - 1$, where m is 32 or 64.
  - **cdef**, when used for variables, denotes a type declaration.
    - Optional. If a name is used without a **cdef**, the default type is general Python object.
    - Technically speaking, since *k*, *n* and *out* have a primitive datatype, they are ***variables*** – *names for fixed storage locations*, like in C. Now e.g. *k += 1* ***actually mutates*** the variable.

- Explicit memory layout: the pythonic syntax [::1] denotes a *contiguous* rank-1 array.
- *range()* now comes from Cython, not from Python, although it looks the same.
- The loop now runs at native speed, but doesn't yet release the GIL. (See next slide.)

- For more, see **Cython for NumPy users**.

Cython

- **Releasing the GIL.** A naive O(n^3) dgemm (**note**: *better algorithms exist!*):

```
import numpy as np    # just a usual Python import

def dgemm(double [:,::1] A, double [:,::1] B):    # matrix-matrix multiply, double precision
    cdef unsigned int i, j, k
    cdef unsigned int n = A.shape[0]
    cdef unsigned int m = B.shape[1]    # the datatype double [:,::1] ensures ndim=2.
    cdef unsigned int p = A.shape[1]
    cdef double [:,::1] out = np.zeros((n,m), dtype=np.float64, order='C')  # or empty(), ones(), ...

    if B.shape[0] != p:
        raise ValueError('Incompatible input shapes ({:d},[:d}), ({:d},{:d})'.format(n,p,B.shape[0],m))

    with nogil:
        for i in range(n):
            for j in range(m):
                for k in range(p):
                    out[n,m] += a[i,k] * b[k,j]

    return np.asanyarray(out)   # return np.array, not memoryview (with no extra copying of data)
```

- The “context manager” **with nogil**: releases the GIL for the dynamic extent of the **with** block.
- Almost always, *np.array* is a good choice for *dynamically allocated*, but immutable-size arrays.
  - Cython can use C primitives malloc/free (no need for GIL!), but very painful (just like in C).
- Memoryviews talk with np.arrays seamlessly, except at output (**return**) must be converted.
  - When creating the array, we use *order='C'* to be explicit that we want a *[:,::1]*.

Cython

- What exactly happens in the *np.array* ↔ *memoryview* conversion:

```
import numpy as np

def test():
    A = np.random.random((5,5))
    cdef double [:,::1] B = A
    C = np.asanyarray(B)
    print(A is C)                             # False; A and C are different np.array instances
    print([type(x) for x in (A,B,C)])         # B is a memoryview; A and C are np.arrays
    print(&A[0,0] == &B[0,0] == &C[0,0])      # memory address of element [0,0] remains the same
```

& is Cython (and C) for "take the memory address of".
Python-level code can use **id** and **is**, as usual.

It **must** work like that, because
this line does not refer to "A".

- We first create an np.array, and bind it to the name A, as a general Python object.

- Then, the line with the **cdef**:
  - We first declare a C-contiguous memoryview, with the name "B".
  - We attempt to bind the name "B" to point to the same object as "A", but "A" is an np.array.
  - Cython **implicitly converts** "A" to a memoryview.
    - ⚠ Recall that pure Python *makes no implicit conversions*. This is different in Cython.
  - Finally, this new memoryview instance is bound to the name "B".
  - We could put this on one line as **cdef double** [:,::1] B = np.random.random((5,5))
    - In that case, even though the original *np.array* disappears, the data inside it – strictly speaking, the exact same region of memory – lives on in the memoryview instance.

- *np.asanyarray()* reverses the conversion, wrapping the same memory back into a new *np.array.*
  - For returning a memoryview, an explicit conversion is needed, because Cython will **return** the exact object instance we tell it to. Alternatively, we can return "A", already an *np.array*.

Cython

- **C functions**. The previous examples are, in principle, usual Python functions, although they contain some C-like code and they are compiled into a native extension module.
- To avoid Python's high overhead for function calls, Cython provides ***C functions***, which are:
  - “called at the C level”, using the faster C calling conventions.
  - ***not visible to Python modules*** – can only be called from Cython code.
    - “Cython code” includes also any regular (**def**) functions in Cython modules.

```
cdef double cddot(double [::1] a, double [::1] b) nogil:    # keyword cdef makes a C function.
    cdef unsigned int k
    cdef unsigned int n = a.shape[0]
    cdef double out = 0.0

    for k in range(n):
        out[k] += a[k] * b[k]

    return out
```

- Note optional C-like type declaration for the return value, just before the function name.
- **nogil** at the end of the function signature means that this C function ***may be called also*** when the GIL is not held. (This implies the function cannot use any Python objects!)
  - *This does not release the GIL*; it allows calls **also** without the GIL.
- C functions, by default, ***cannot propagate Python exceptions*** to the caller, but they can tag a special ***error return value*** to add this functionality. Use an **except** keyword at the end of the function signature, just use exceptions as usual, and ***never manually return the tagged value***.
- See also Cython Function Declarations; and Early Binding for Speed.
- Beside **cdef**, there is also **cpdef**, which means “define this function for both C and Python”.

Cython

- **Summary, language basics**:

  - Cython significantly accelerates **for** loops that use only *np.array*s (≈100× over pure Python).
    - Needs static type information so that the loop can be compiled to C.
      - Must be provided by the programmer, in the form of type declarations (**cdef**).

  - To avoid Python's high overhead for function calls, Cython provides ***C functions*** (also **cdef**).
    - A C function can be declared callable without the GIL by appending **nogil** to the signature.

  - When you don't need Python objects except *np.array*s, you can **release the GIL with nogil**:

    - **Danger**: **with nogil**: is in effect *during the dynamic extent of the **with** block* – not lexically!
      - In other words, it behaves just like a context manager.
    - **Danger**: Trying to release the GIL when it is already released **will crash the program**.
      - **Design carefully**; combine with **nogil**. In each thread, use at most one **with nogil**: at a time; but it may safely call, also in a nested manner, any C functions which are **nogil**.

  - NumPy arrays in Cython:
    - A good choice for dynamically allocated, but immutable-size arrays. Garbage-collected.
    - At the C level, access via typed memoryviews. At the Python level, just use as usual.
    - *Memory layout* can be C-contiguous *[:,…,::1]*, Fortran-contiguous *[::1,…,:]*, or general.
      - General layouts are denoted by predefined constants; to use them, ***cimport*** *cython.view*

    - **Important**: A fast C loop will only be generated **if the array is contiguous, and Cython knows it** (from a datatype declaration)!
      - Access to a fully general memory layout cannot be accelerated (as much), because then the only way to find out where the data is, is to ask the memoryview for that information.

Cython

- **Parallel loops**:

```
import numpy as np
cimport cython.parallel    # note cimport instead of import (explanation on next slide)

def pdgemm(double [:,::1] A, double [:,::1] B):    # parallel dgemm
    cdef unsigned int i, j, k
    cdef unsigned int n = A.shape[0]
    cdef unsigned int m = B.shape[1]
    cdef unsigned int p = A.shape[1]
    cdef double [:,::1] out = np.zeros((n,m), dtype=np.float64, order='C')

    if B.shape[0] != p:
        raise ValueError('Incompatible input shapes ({:d},[:d}), ({:d},{:d})'.format(n,p,B.shape[0],m))

    with nogil:
        for i in cython.parallel.prange(n):    # ← parallel loop
            for j in range(m):
                for k in range(p):
                    out[n,m] += a[i,k] * b[k,j]      # arrays can be accessed as usual

    return np.asanyarray(out)
```

⚠ OpenMP not to be confused with Open MPI.

- Uses *OpenMP (Open Multi-Processing)*. Needs compiler and linker options to enable OpenMP.
- **Available only at the C level**; no access to Python objects in parallel code.
- *prange()* must either be inside a **with nogil**: block, or use the magic kwarg *nogil=**True***.
- May only be used from the main thread or parallel regions due to OpenMP restrictions.
- For more (e.g. thread-local variables), see **Parallelism** in the Cython documentation.

Cython

- **Cython's import system.** Both of these are valid, but each works for a different reason:

```
cimport cython.parallel

def do_stuff():
    cdef unsigned int i, n=10
    ...
    with nogil:
        for i in cython.parallel.prange(n):
            …
    …
```

```
import cython.parallel

def do_stuff():
    cdef unsigned int i, n=10
    ...
    for i in cython.parallel.prange(n, nogil=True):
        …
    …
```

- **cimport** is like C's #include <something.h>
- Brings in **C-level definitions**
- Just like **cdef**, seen only by Cython code

- **import** is just Python's **import**
- Brings in **Python modules or objects**, as usual
- Works just like in any Python code

- **Be precise.** For example, trying to ***import*** *cython.parallel* and then use it in a ***with nogil****:* block *is a compile-time error*, because **Python objects** cannot be used with the GIL released! But using a **C-level definition** with the GIL released is fine.
- *cython.view* must be **cimport**ed (not **import**ed!), because the things that are used to denote general memory layouts are defined as **C-level constants**.
- NumPy must be **import**ed (not **cimport**ed!), because the things we need from it (such as *np.empty()*, *np.asanyarray()*) are **Python functions**.

Cython

- **Definition files**:

  - Like in C, Cython code can be split to separate *definition* and *implementation* files.
    For example, for functions:
    - A *definition* states that a function with a certain name exists, and specifies its signature.
    - An *implementation* contains the actual code.

  - Definition files are used to export functions at the C level – i.e. to make them available for other Cython modules to **cimport**.
    - If a module is intended not to export anything at the C level, no definition file is needed.

  - **Implementation files** have the file extension ***.pyx***
  - **Definition files** have the file extension ***.pxd***; they play a role similar to C's header files (*.h*).

  - Roughly, in C terms a *declaration* corresponds to a Cython *definition*, and a C *definition* corresponds to a Cython *implementation*. (Edge cases slightly complicate matters.)

  - Contrast pure Python, which requires only an implementation. Separate definitions are typically a feature of statically compiled languages; functions are looked up statically instead of at runtime, as in Python.
    - Roughly speaking. We have ignored C++'s virtual methods, which use a limited form of runtime lookup to account for the object's *dynamic type*. (Recall lecture 3, slide 45.)
    - A declaration gives the *compiler* enough information to resolve the reference, whereas the definition provides what the *linker* needs to concretely fix the function to be called.

  - See **Sharing Declarations Between Cython Modules** in the Cython documentation.

Cython

- **Definition files – example**:

```
# mysum.pxd
cdef double sum(double[::1] a) nogil

# mysum.pyx
cdef double sum(double[::1] a) nogil:
    cdef unsigned int k, n = a.shape[0]
    cdef double out = 0.0
    for k in range(n):
        out += a[k]
    return out

# main.pyx
import numpy as np
cimport mysum

def main():
    lst = tuple(np.arange(10, dtype=np.float64))
    print(mysum.sum(lst))  # use just like any Python function (as long as the datatypes match)

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

- Cython must be able to find *mysum.pxd* when compiling *main.pyx*.
  - This can sometimes be a source of headaches, especially when writing libraries. In Python 3, libraries use relative **import**s, but Cython doesn't fully seem to understand that.
  - One possible solution is to use absolute(-ish) **cimport**s; see *setup-template-cython*.

Cython

- ***Extension types*** (a.k.a. ***cdef classes***):
  - On equal footing with Python's built-in types. Uses less memory and is faster than a class.
  - Uses a C struct to store fields and methods, instead of a Python dict.
    - Can store **arbitrary C types** in the fields without requiring a Python wrapper.
    - Can access fields and methods directly at the C level (without a dict lookup).
    - Cannot add or remove fields, or change their datatype, at runtime.
  - **Example**: creating a Python-level wrapper for a generic C pointer (a.k.a. *void pointer*, **void** *):

```
# ptrwrap.pxd
cdef class PointerWrapper:
    cdef void * data
    cdef set(self, void * p)

# ptrwrap.pyx
cdef class PointerWrapper:
    cdef set(self, void * p):
        self.data = p

# main.pyx
import numpy as np
from ptrwrap cimport PointerWrapper
def main():
    cdef double [:,::1] A = np.empty((3,5), dtype=np.float64, order='C')
    cdef PointerWrapper pw = PointerWrapper()
    pw.set(&A[0,0])
    cdef double [:,::1] B = <double [:3,:5:1]>pw.data  # specify size, a C pointer doesn't know!
main()
```

Cast to the right type to later access the data.

In Cython, the size is specified using the slice syntax.

Where is this needed? To Python, a *PointerWrapper* is a Python object; it can be passed into a function as an argument, returned from a function, stored into the usual Python containers such as *list*, etc.

This can be convenient if your Cython code needs to pass pointers around internally, but the function to be called is a regular **def** function (Python function).

In real code, if a *PointerWrapper* is passed/stored as a Python object, cast it explicitly to a *PointerWrapper* before trying to access its *data* field, because it's a C field, not a Python attribute! Code example.

Cython

- The **Cython and C compilation steps** are separate; different compilers, different languages.
  - Example: embedding SSE2 assembly (for the x86 family), via C FFI to SIMD intrinsics.
  - No ready-made *.pxd* for this exists. To declare (i.e. write Cython definitions for) the existing, external C functions we want to use, we must first tell Cython about the datatype *__m128d*, which is a pair of **double**s packed into one 128-bit item for SSE2-based SIMD processing:

```
cdef extern from "emmintrin.h":    # emmintrin for SSE2
    ctypedef double __m128d

    __m128d _mm_loadu_pd (double *__P) nogil # (__P[0], __P[1]) are the input doubles
    __m128d _mm_add_pd (__m128d __A, __m128d __B) nogil
    __m128d _mm_mul_pd (__m128d __A, __m128d __B) nogil
    void _mm_store_pd (double *__P, __m128d __A) nogil # result written to (__P[0], __P[1])
```

  - **Cython and GCC need different information**:
    - Cython must be told, within the constraints of its syntax, that *an __m128d behaves somewhat like a* ***double*** – so that it knows how to generate the C code correctly.
    - But the C compiler needs the exact definition – *an __m128d is not a* ***double****, but a pair of* ***double****s*. To get the Cython-generated C code to *#include* the actual original definition, so that the C compile step can work, we must **cdef extern from** the original header.
  - Full example on GitHub. (**Danger**: GCC-optimized C may be faster than manual assembly!)

Cython

- **Compiling**, or how to write a *setup.py*

  - Closely related to packaging and distributing Python projects; uses the same tools.
    - A large topic. See the Packaging and Distibuting Projects tutorial by PyPA.
      - Still a moving target; update your skills regularly.
      - As of early 2018, the recommended framework is *setuptools*.

    - Cython's documentation is based on the older *distutils*, but the relevant parts are somewhat compatible.
    - GCC compiler options for x86 may come in handy.

  - Examples too long to fit on a slide; see online:

    - A very minimal example for Cython compilation and distribution in *cython-sse-example.*

    - A template for number-crunching projects using Cython: *setup-template-cython*
      - For compile only, no distribution, see its test subfolder.

    - A template for FFI projects using Cython: *simple-cython-example*

    - Cf. a distribution script for a project using CFFI, but no Cython, in *sparseqr*.
    - Cf. a distribution script for a pure-Python project, no compile, in PyPA's *sampleproject*.

  - In simple cases, invoke with ***python setup.py build_ext --inplace***
    - *build_ext* compiles; *build* just copies .py files declared as belonging to packages.
  - If you know what you are doing, invoke with ***python setup.py build_ext***
    - The folder structure generated by this variant is very useful for making a package.

Cython

- **Performance**:

  - Some compiler directives can be used to enable a significant performance boost.
    - How to set them: per-function decorators, magic comments at start of file.
    - Default settings:
      - *boundscheck=True*: Array accesses are bounds-checked, like in Python.
        - Safe (no mysterious crashes or program state corruption), but slow.
        - Once sure that there are no indexing-past-end bugs, consider disabling.
      - *wraparound=True*: Negative indices are allowed, like in Python.
        - Convenient, but slow. Disable, if sure that no negative indices are used.
      - *cdivision=False*: The division operator uses Python semantics. Very slow!
        - Almost always, switch to C semantics.

  - If your code is running suspiciously slowly:

    - First, invoke ***cython -a mymodule.pyx***
      - This produces ***mymodule.html***, showing the relative efficiency of the generated code.
      - Open it in a web browser to see your source, annotated (-a) with the generated C code.
      - Click on any line with a "+" in front of it to expand.
      - The more yellow a line is, the more Python work it is doing.
        - Lines that translate to pure C appear white.

    - For more detail, look at the generated C code. See e.g. if one of your loops:
      - queries the array for its *strides*; if so, Cython doesn't know the array is contiguous. Likely the datatype declaration for the array variable is missing, or is missing the ::1.
      - calls any Python functions when manipulating the loop counter variable; if so, the datatype declaration for the loop counter is missing.

Cython

- **Final words**:
  - Regular Python modules can **import** Cython modules just fine – which is the point of Cython, i.e. providing acceleration selectively where needed. Usually there is a *main.py*, no *main.pyx*.
  - Our examples have used Cython only, just because we have concentrated on demonstrating Cython's C-level features.
  - We looked at **cdef extern from** to declare external functions when no ready-made *.pxd* definition file exists; but often there is one, such as *libc.math* or scipy.linalg.cython_lapack, so you can just **cimport** it. See Calling C functions, and includes provided with Cython.
  - In Cython modules, function signatures do not automatically appear in *help()*.
    - To work around this, manually copy the signature to the beginning of the function docstring – users will expect to see it.
    - (And remember to update the manual copy if you change the signature later.)
  - **Is it worth it**? If you need to write custom low-level code, **yes**.
    - Achieve true parallelism for CPU-bound workloads.
    - Achieve native speed – *the real deal, -O3 C* – for number-crunching loops.
    - Eliminate the cost of Python function calls.
    - …all while not deviating *that* far from Python.
  - But keep in mind the Pareto principle – very often, it happens that 80% of your run time comes from 20% of the code.
    - ***Identify that 20% before working too much on acceleration.***
    - "*Premature optimization is the root of all evil*" –Donald Knuth

Meta

Next time

- Introduction to software engineering
  - I.e. *how to develop correct and maintainable software quickly*
  - Tools of the trade:
    - Basics of version control with *git* and GitHub
    - Debuggers
    - Profilers (finding that 20%, reliably)
  - How to write informative comments
  - Assertions, ensuring internal consistency
  - Automated unit testing to avoid new bugs, and catch regressions

- See you next week!