ICHEC

Irish Centre for High-end Computing

PRACE

SUMMER OF HPC

Numba

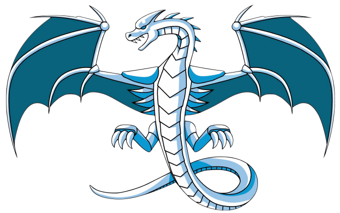
Introduction

`@numba.jit()`

- A **just-in-time** compiler for Python functions.
- From the types of the function argument it translates the function into a specialised, fast, **machine code** equivalent.
- Uses LLVM (https://llvm.org) compiler infrastructure for code generation.
- Get speeds similar to C/C++ and Fortran but without having to write any code in those languages! - doesn't require C/C++ compiler.
- Simple as applying one of the Numba **decorators** to your Python function
- A decorator is a function that takes in a function as an argument and spits out a function

https://numba.pydata.org/

https://numba.discourse.group/



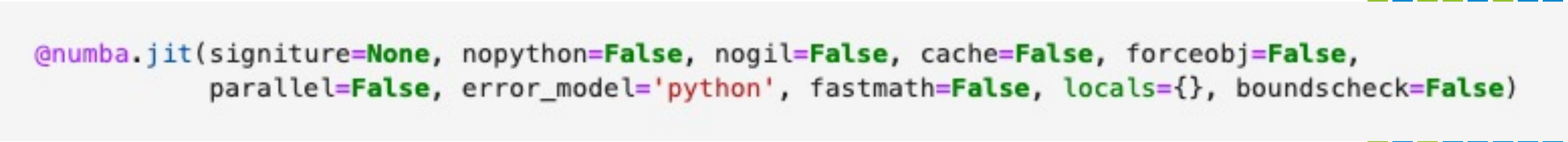
Introduction

- Designed to work well with **NumPy** arrays - very useful for scientific computing.
- Makes it easy to **parallelise** your code and use multiple threads.
- Does SIMD vectorisation to get most out of your CPU. Meaning a single instruction can be applied to multiple data elements in parallel. It will automatically translate some loops into vector instructions and will adapt to your CPU capabilities.
- Can run Numba code on GPU. (Not covering that in this course).
- **“Numba” = “NumPy”+ “Mamba”**

NumPy

Overview

- **Theory:** The **jit** decorator
- **Exercise 1**: Monte Carlo
- **Theory:** Compilation time
- **Demo -** modes
- **Theory: nopython** mode
- **Exercise 2**: Mandelbrot
- **Demo –** Mandelbrot
- **Demo -** is_prime function & toggles
- **Exercise 3**: Toggles
- **Theory:** Parallelising
- **Demo -** parallel
- **Exercise 4**: parallel=True
- **Theory:** Vectorising – ufuncs
- **Demo -** Vectorize
- **Exercise 5**: vectorize is_prime function
- **Wrap-up**



```
@numba.jit(signiture=None, nopython=False, nogil=False, cache=False, forceobj=False,
           parallel=False, error_model='python', fastmath=False, locals={}, boundscheck=False)
```

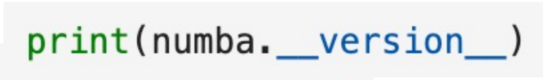
The Jit compiler options/toggles

- **signature** - The expected types and signatures of function arguments and return values **- Eager Compilation**
- Numba has two modes - **nopython, forcobj**. Numba will infer the argument types at call time, and generate optimized code based on this information.
- **nogil=True** - releases the gil inside the compiled function - only if in **nopython** mode.
- **cache=True** - enables a file-based cache to shorten compilation times when the function was already compiled in a previous invocation. Cannot be used in conjunction with **parallel=True**.

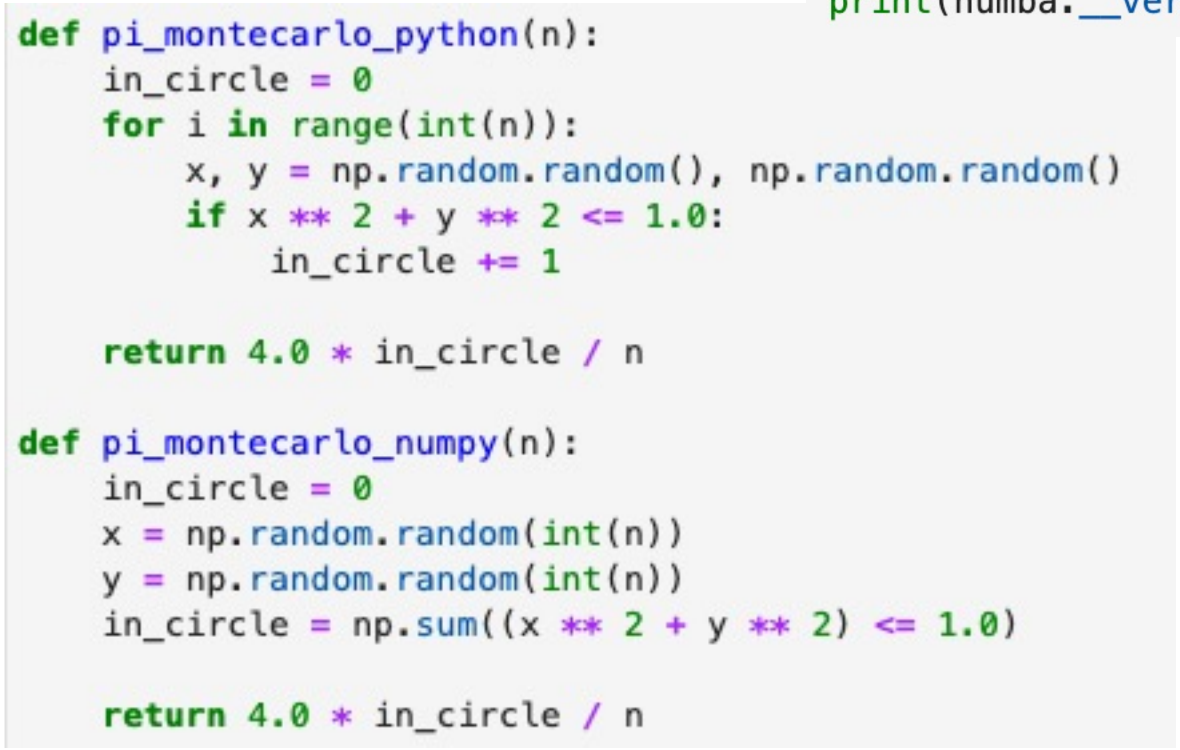
The Jit compiler options/toggles

- **parallel=True** - enables the automatic parallelization of a number of common NumPy constructs.
- **error_model** - controls the divide-by-zero behaviour. Setting it to 'python' causes divide-by-zero to raise exception. Setting it to 'numpy' causes divide-by-zero to set the result to +/-inf or nan.
- **fastmath=True** - enables the use of otherwise unsafe floating point transforms as described in the LLVM documentation.
- **locals** dictionary - used to force the types and signatures of particular local variables. Recommended to let Numba's compiler infer the types of local variables by itself.
- **boundscheck=True** - enables bounds checking for array indices. Out of bounds accesses will raise IndexError. Enabling bounds checking will slow down typical functions, so it is recommended to only use this flag for debugging etc.

Exercise 1 - Monte Carlo



```python
print(numba.__version__)
```



```python
def pi_montecarlo_python(n):
    in_circle = 0
    for i in range(int(n)):
        x, y = np.random.random(), np.random.random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n

def pi_montecarlo_numpy(n):
    in_circle = 0
    x = np.random.random(int(n))
    y = np.random.random(int(n))
    in_circle = np.sum((x ** 2 + y ** 2) <= 1.0)

    return 4.0 * in_circle / n
```

Exercise 1- Monte Carlo

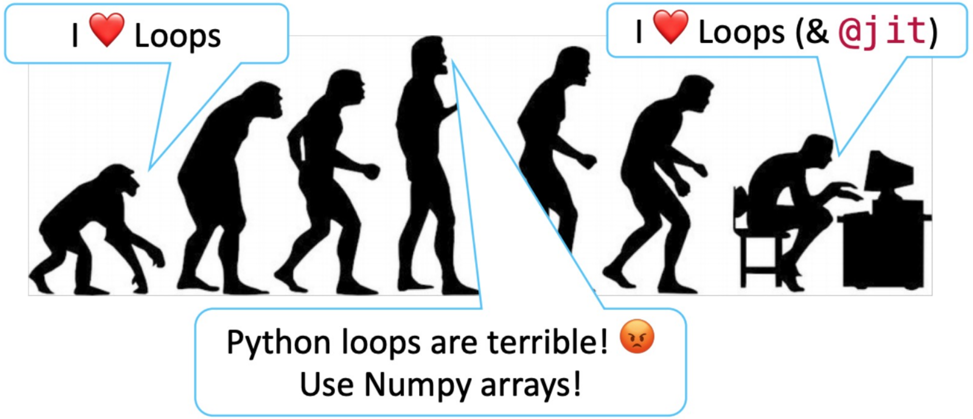
- See which is faster – Python or NumPy version of Monte Carlo method estimating Pi.
- Add a @jit() to the line above both functions – and run using the job submission file - notice something about the times?

Compilation time - explained

- Once the **compilation** has taken place Numba **caches** the machine code version of the function for the particular types of arguments presented. Eg if we changed n = 1000000.0
- To benchmark Numba-compiled functions, it is important to time them without including the compilation step - the **compilation will only happen once for each set of input types**, but the function will be called many times.
- By adding @jit decorator we see **major speed ups** for Python and a bit for NumPy.
- This is where Numba is very useful - speeding up python loops that cannot be converted to NumPy or it's too complicated. NumPy can sometimes reduce readability.
- Get huge speed ups with minimum effort.

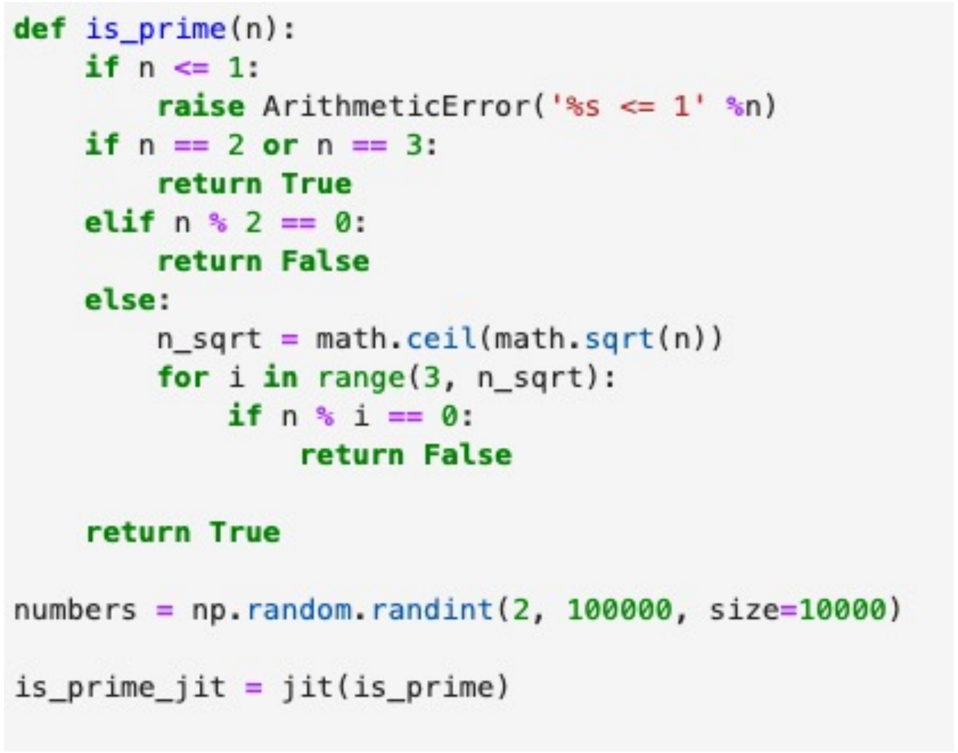
ICHEC
Irish Centre for High-end Computing

- Don’t write NumPy Haikus. If loops are simpler, write loops and use Numba!” - Stan Seibert, Numba team, Anaconda



Credit: Jason Watson (PyGamma19)

Demonstrating Modes



```python
def is_prime(n):
    if n <= 1:
        raise ArithmeticError('%s <= 1' %n)
    if n == 2 or n == 3:
        return True
    elif n % 2 == 0:
        return False
    else:
        n_sqrt = math.ceil(math.sqrt(n))
        for i in range(3, n_sqrt):
            if n % i == 0:
                return False

    return True

numbers = np.random.randint(2, 100000, size=10000)

is_prime_jit = jit(is_prime)
```

- modes_demo.py
- Run as is
- Add @jit()
- Try no python mode
- Fix problem
- Compare times again

Explaining nopython mode

- **@jit(nopython=True)** is equivalent to **@njit**
- The behaviour of the **nopython** compilation mode is to essentially compile the decorated function so that it will run entirely without the involvement of the **Python interpreter**.
- If it can't do that an exception is raised. These exceptions usually indicate places in the function that need to be modified in order to achieve **better-than-Python performance**. Strongly recommend always using **nopython mode**.
- Object mode (**forceobj=True**) can extract loops and compiles them in nopython mode - useful for functions that are bookend by uncompilable code but have a compilable core loops - this is done automatically.
- forcobj - supports nearly all of python but cannot speed up by a large factor
- nopython - supports a subset of python but runs at C/C++/fortran speeds

ICHEC
Irish Centre for High-end Computing

```python
def plot_mandel(mandel):
    fig=plt.figure(figsize=(10,10))
    ax = fig.add_subplot(111)
    ax.set_aspect('equal')
    ax.axis('off')
    ax.imshow(mandel, cmap='gnuplot')

def kernel(zr, zi, cr, ci, radius, num_iters):
    count = 0
    while ((zr*zr + zi*zi) < (radius*radius)) and count < num_iters:
        zr, zi = zr * zr - zi * zi + cr, 2 * zr * zi + ci
        count += 1
    return count

def compute_mandel_py(cr, ci, N, bound, radius=1000.):
    t0 = time.time()
    mandel = np.empty((N, N), dtype=int)
    grid_x = np.linspace(-bound, bound, N)

    for i, x in enumerate(grid_x):
        for j, y in enumerate(grid_x):
            mandel[i,j] = kernel(x, y, cr, ci, radius, N)
    return mandel, time.time() - t0

def python_run():
    kwargs = dict(cr=0.3852, ci=-0.2026,
                  N=400,
                  bound=1.2)
    print("Using pure Python")
    mandel_func = compute_mandel_py
    mandel_set = mandel_set, runtime = mandel_func(**kwargs)

    print("Mandelbrot set generated in {} seconds".format(runtime))
    plot_mandel(mandel_set)
    mandel_timings.append(runtime)
```

Exercise 2 Mandelbrot

Go through code in exercise2.py
Spot a function that we could Numba –fy ?

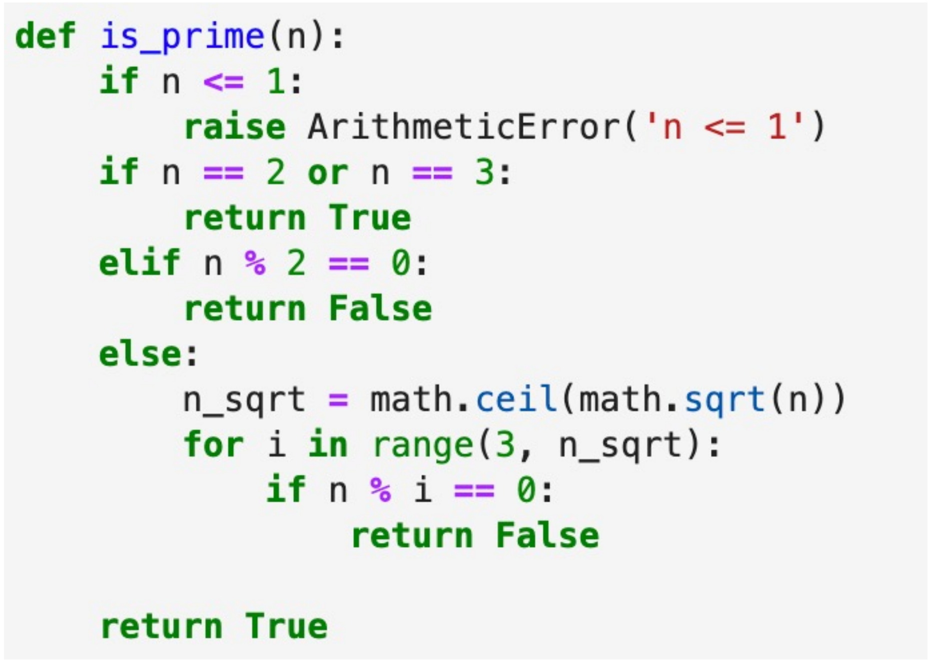
```
scp course01@kay.ichec.ie:/ichec/home/users/course01/sohpc-training-2021/D2-Python/04-Numba/small_exercises/mandel.png .
```

Mandelbrot continue -Demo

Can we speed it up more?? Try the Compute function??

- Run mandel_demo1.py locally
- Try njit and jit
- Look at error
- Go to mandel_demo2.py (same but removed time.time() within function)
- Look at error
- Fix
- Compare timings

is_prime.py – numba toggles



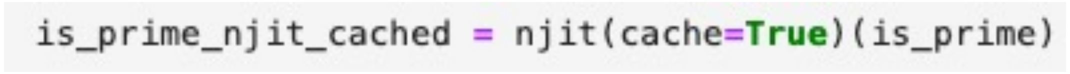
```python
def is_prime(n):
    if n <= 1:
        raise ArithmeticError('n <= 1')
    if n == 2 or n == 3:
        return True
    elif n % 2 == 0:
        return False
    else:
        n_sqrt = math.ceil(math.sqrt(n))
        for i in range(3, n_sqrt):
            if n % i == 0:
                return False

    return True
```

is_prime_toggles.py
demonstrate locally

ICHEC Irish Centre for High-end Computing

cache=True

- The point of using `cache=True` is to avoid repeating the compile time of large and complex functions at each run of a script. In this example the function is simple and the time saving is limited but for a script with a number of more complex functions, using cache can significantly reduce the run-time.



```
is_prime_njit_cached = njit(cache=True)(is_prime)
```

Eager Compilation using function signatures

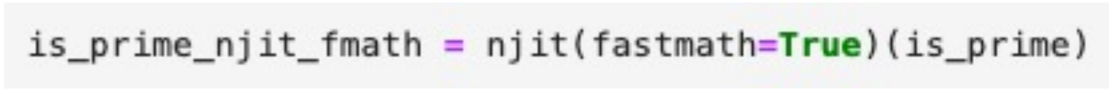
- order them from smaller precision to higher

```
is_prime_eager = njit(['boolean(int32)','boolean(int64)' ])(is_prime)
```

- Compare njit & eager
- Compare with njit version to this eager version using numbers.astype(np.int64)
- Try np.float64 in both – what do you expect?

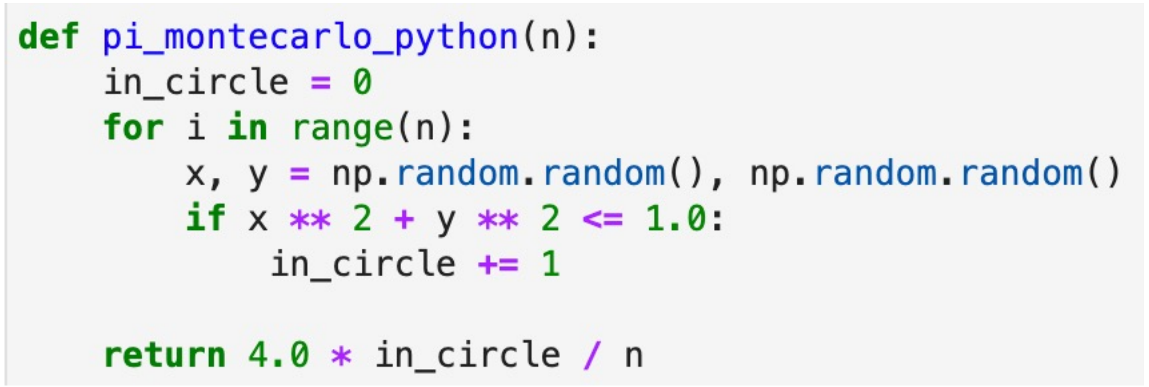
fastmath=True

- Enables the use of otherwise unsafe floating point transforms
- Meaning it's possible to relax some numerical rigour with view of gaining additional performance.
- For example assume the arguments and result are not NaN or Infs.
- https://llvm.org/docs/LangRef.html#fast-math-flags



```
is_prime_njit_fmath = njit(fastmath=True)(is_prime)
```

Exercise 3 - toggles



```
def pi_montecarlo_python(n):
    in_circle = 0
    for i in range(n):
        x, y = np.random.random(), np.random.random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n
```

pi_montecarlo_python_njit = ###
pi_montecarlo_python_eag = ###
pi_montecarlo_python_fm = ###
pi_montecarlo_python_cache = ###
pi_montecarlo_python_ALL = ###

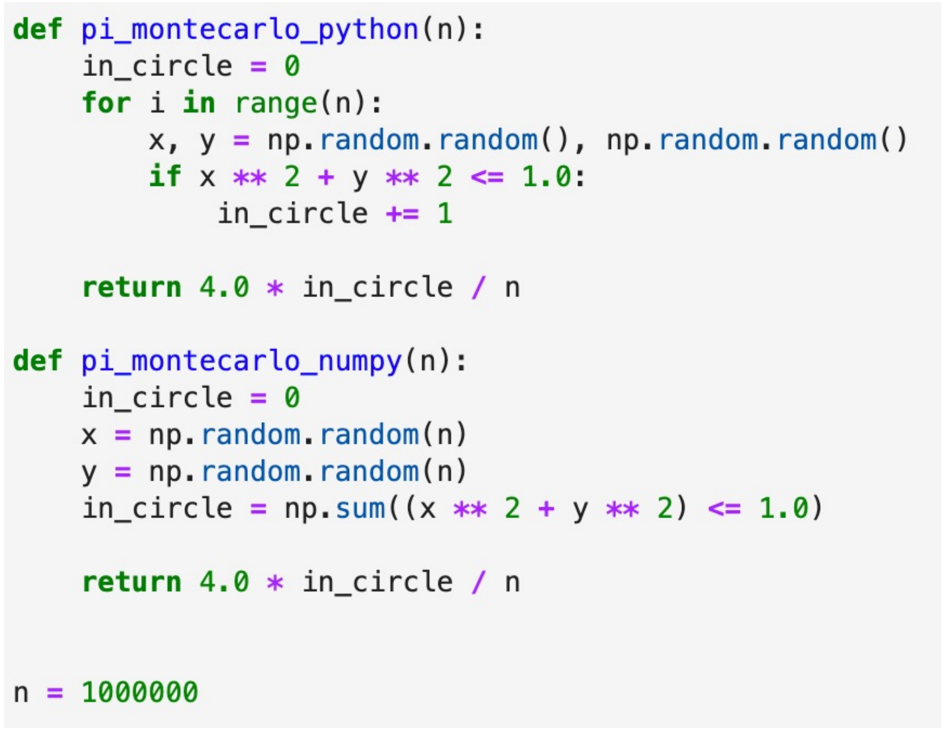
parallel=True

- Add parallel=True to use multi-core CPU via threading
- Use **numba.prange** with parallel=True if you have **for** loops
- With the default parallel=False, numba.prange is the same as range.
- Default number of threads (max) = **numba.config.NUMBA_NUM_THREADS**

```
max_threads = numba.config.NUMBA_NUM_THREADS
```

```
numba.set_num_threads(2)
```

parallel=True - example



```python
def pi_montecarlo_python(n):
    in_circle = 0
    for i in range(n):
        x, y = np.random.random(), np.random.random()
        if x ** 2 + y ** 2 <= 1.0:
            in_circle += 1

    return 4.0 * in_circle / n

def pi_montecarlo_numpy(n):
    in_circle = 0
    x = np.random.random(n)
    y = np.random.random(n)
    in_circle = np.sum((x ** 2 + y ** 2) <= 1.0)

    return 4.0 * in_circle / n


n = 1000000
```

- Change range to prange
- Show locally

Exercise 4: parallel=True

- Repeat what I just did and submit to Kay
- Note the max number of threads now

Creating ufuncs using numba.vectorize

- A universal function (or **ufunc** for short) is a function that operates on ndarrays in an **element-by-element** fashion. Is a “vectorized” wrapper for a function. For example np.add() is a ufunc.
- wo types of ufuncs:
    - Those which operate on scalars, ufuncs (see **@vectorize** below).
    - Those which operate on higher dimensional arrays and scalars, these are “generalized universal functions” or **gufuncs** - (**@guvectorize**).
- The **@vectorize** decorator allows Python functions taking scalar input arguments to be used as **NumPy ufuncs**. Creating a traditional NumPy ufunc involves writing some C code - Numba makes this easy. This decorator means Numba can compile a pure Python function into a ufunc that operates over NumPy arrays as fast as traditional ufuncs written in C.

Creating ufuncs using numba.vectorize

- The **vectorize() decorator** has two modes of operation:
- **Eager**, or decoration-time, compilation: If you pass one or more type signatures to the decorator, you will be building a Numpy universal function (ufunc).
- **Lazy**, or call-time, compilation: When not given any signatures, the decorator will give you a Numba dynamic universal function (**DUFunc**) that dynamically compiles a new kernel when called with a previously unsupported input type.
- If you pass several signatures, beware that you have to pass most specific signatures before least specific ones (e.g., single-precision floats before double-precision floats), otherwise type-based dispatching will not work as expected. EG (**int32,int64,float32,float64**)

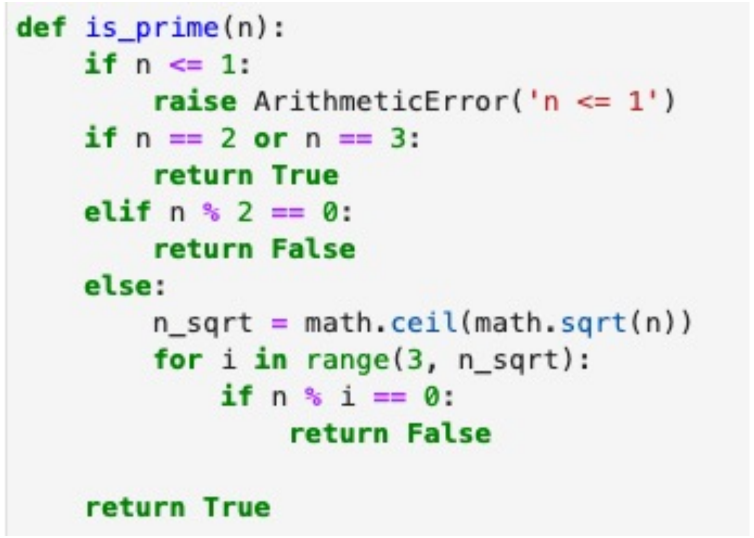
Creating ufuncs using numba.vectorize

- Here is a very simple example:

```
def numpy_sin(a, b):
    return np.sin(a) + np.sin(b)

numpy_sin_vec = numba.vectorize(['int64(int64, int64)','float64(float64, float64)'])(numpy_sin)

numpy_sin_vec_par = numba.vectorize(['int64(int64, int64)','float64(float64, float64)'],
                                    target='parallel')(numpy_sin)
```

Exercise 5 Vectorize is_prime

- Think about the input types and output types.
- Note that once its vectorised we won’t need the **for loop** when putting the numbers in



```python
def is_prime(n):
    if n <= 1:
        raise ArithmeticError('n <= 1')
    if n == 2 or n == 3:
        return True
    elif n % 2 == 0:
        return False
    else:
        n_sqrt = math.ceil(math.sqrt(n))
        for i in range(3, n_sqrt):
            if n % i == 0:
                return False

    return True
```

Numba limitations

- Numba only compiles **individual functions** not whole scripts
- Only supports a **subset** of Python and Numpy
- But that is **changing** fast!
- Keep an eye out for **new versions** to stay up to date

Summary

- Numba is very useful for **speeding up Python loops**
- The two modes - recommend **nopython=True, njit**
- cache=True for **reducing compilation time**
- eager compilations also reduces compilation and calculation time
- fastmath = True
- Achieve parallelism very easily - automatic with NumPy, and for Python for loops use **prange**