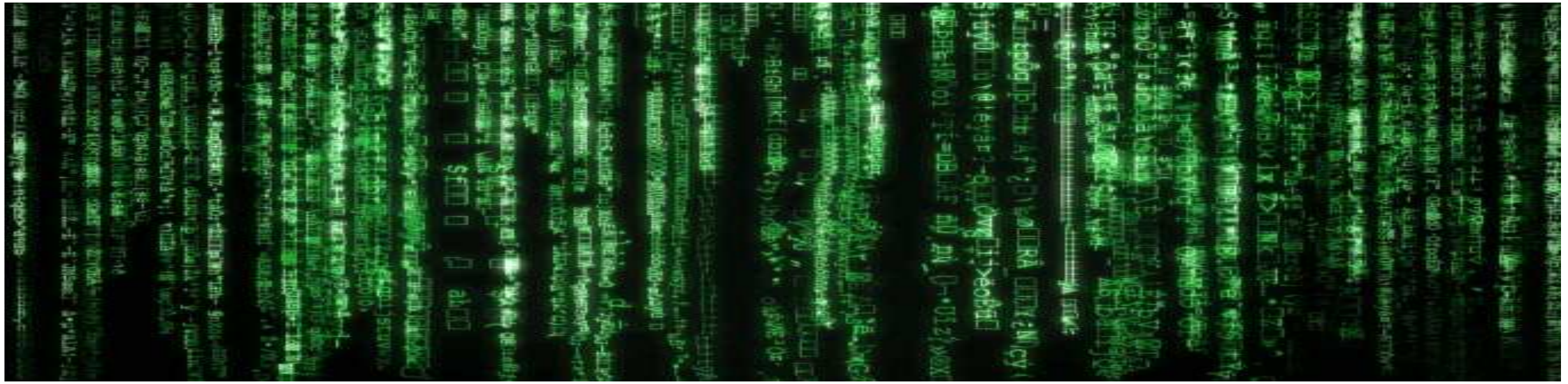

INF 110 Discovering Informatics

NumPy and Data Science

NAU
NORTHERN
ARIZONA
UNIVERSITY

Why Are We Doing This?

- Arrays and lists have a lot in common.
- So why not just use built-in lists?

- Answer: **EFFICIENCY**.

- Python has a reputation for being inefficient in terms of speed and memory usage (*in other words, it's slow and greedy*)

- In contrast, FORTRAN or C are considered efficient.
  - ***But they are hard to learn!***

Problem: Python is actually pretty slow..

- Python isn't as fast as languages like C or Fortran; Why?
  - Bytecode interpreted
  - Garbage collection
  - Dynamic typing
  - Duck typing
  - High abstraction level
  - Basically all the things that make it pleasant to use also make it slow
- Every operation is relatively expensive (time, memory)!

How slow?

Example:

Iterate over a list of 100 million numbers and perform 100 million additions

You can do this in both C and Python:

Code in C

```c
#include <stdio.h>
int main() {
  int i; double s=0;
  for (i=1; i<=100000000; i++) s+=i;
  printf("%.0f\n",s);
}
```

Code in Python

```python
s=0.
for i in xrange(1,100000001):
    s+=i
print s
```

Both of the codes compute the sum of integers from 1 to 100,000,000.

How slow?

- The C code took 0.109 seconds
- This Python code took 8.657 seconds
- That's ***80 times*** slower!
- For complicated operations it's worse!

Code in C

```c
#include <stdio.h>
int main() {
  int i; double s=0;
  for (i=1; i<=100000000; i++) s+=i;
  printf("%.0f\n",s);
}
```

Code in Python

```python
s=0.
for i in xrange(1,100000001):
    s+=i
print s
```

Both of the codes compute the sum of integers from 1 to 100,000,000.

How slow?

- The
  seco
- This
  8.65
- That's ***80 times*** slower!
- For complicated operations it's worse!

So if Python is so bad at its job, then why use it?

```
s=0.
for i in xrange(1,100000001):
    s+=i
print s
```

Both of the codes compute the sum of integers from 1 to 100,000,000.

NumPy's Trick

- Methods are written in optimized C and then accessed by Python
- They also use ***vectorization***:
  - Methods work on lots of data at once – minimizing how much Python code gets executed
- Other languages use the same trick (e.g. MatLab, Mathematica, Maple)
- Python is particularly easy to extend in this way

The Same Example – now with NumPy

```
x = np.arange(1, 100000001)
np.sum(x)
```

- Only a few Python methods get called.

- We're not telling Python to do any arithmetic or loops
  - The 100,000,000 addition operations happen in ***highly optimized*** C code.

- It's why numpy array values have to be the same data type.
- It also requires you to think about things in a ***slightly different*** way.

Calculating Things the "Traditional" CS Way

"Take each element in an array, add one and then scale by five"

- Begin with an iterator
- Work through the list
- Add one
- Then multiply by five

```
for i in len(values):
    values[i] = (values[i] + 1) * 5
```

But this is SLOW!

Performing Calculations the NumPy way

Avoid using loops and instead do everything with method calls.

- Start with the array you assigned to the variable “values”
- Use the numpy add function, supply argument
- Call the numpy multiply function

```
values = np.add(values, 1)
values = np.multiply(values, 5)
```

It does the same thing, but uses C on the backend with vectorization. This is ***FAST!***

NumPy Cheat Sheet

- There are **hundreds** of NumPy methods
- Only a few dozen are commonly used
- Check out the NumPy Cheat Sheet at dataquest.io

https://www.dataquest.io/blog/numpy-cheat-sheet/

NumPy – A Brief History

- Different array standards competed from 1995-2005
- NumPy 1.0 was released in 2006
- Quickly became the dominant numeric array class for Python
- Still evolving!

Some methods

- `np.prod` Multiply all elements together
- `np.sum` Add all elements together
- `np.all` Test if all elements are true values (non-zero numbers are true)
- `np.any` Test if any elements are true values (non-zero numbers are true)
- `np.count_nonzero` Count the number of non-zero elements

Some methods

- `np.add` Add a scalar or array to an array
- `np.subtract` Subtract a scalar or array from an array
- `np.multiply` Multiply a scalar or array by an array
- `np.divide` Divide a scalar or array by an array

(a scalar is an object in an array – so they have to be the same data type!)

Live Code It's Cold Outside

Task: Change an array of Celsius temperature values to Fahrenheit. Hint: $T_{(°F)} = T_{(°C)} \times 9/5 + 32$

Learning Outcomes

- Creating NumPy arrays
- Looking up NumPy documentation in Jupyter
- Adding and multiplying values in NumPy

Live Code It's Cold Outside Redux

Task: Do the same thing but with a table! Add a column with your temperature conversion

Learning Outcomes

- Loading tables
- Sorting columns
- Adding new columns


end