Webinar

Accelerate Python Loops with the Intel AI Analytics Toolkit

Bob Chesebrough
AI Software Solutions Engineer

intel®

Learning Objectives

- At the end of the webinar, you will be able to:
  - Describe a Python loop replacement strategy using NumPy constructs which:
    - Improves readability, maintainability
    - Performs fasts on current hardware and
    - Readies code for future HW & SW accelerations that Intel builds into silicon, and which are exposed via NumPy
  - Describe NumPy clause to aid aggregations, reductions, broadcasting, and “where” and “select” to significantly accelerate your Python code
  - Describe the value of the Intel AI Analytics Toolkit
  - Describe underlying reasons for the acceleration due to NumPy powered by oneAPI

intel.

Intel® AI Analytics Toolkit

Accelerate end-to-end AI and data analytics pipelines with libraries optimized for Intel® architectures

Who Uses It?

Data scientists, AI researchers, ML and DL developers, AI application developers

Top Features/Benefits

- Deep learning performance for training and inference with Intel optimized DL frameworks and tools
- Drop-in acceleration for data analytics and machine learning workflows with compute-intensive Python packages

Intel® AI Analytics Toolkit

Deep Learning

- Intel® Optimization for TensorFlow
- Intel® Optimization for PyTorch
- Intel® Neural Compressor
- Model Zoo for Intel® Architecture

Machine Learning

- Intel® Extension for Scikit-learn
- Intel-optimized XGBoost

Data Analytics

- Intel® Distribution of Modin
- OmniSci Backend

Intel-optimized Python

- NumPy
- SciPy
- Pandas
- Numba
- Python

CPU GPU

Hardware support varies by individual tool. Architecture support will be expanded over time.

Get the Toolkit HERE or via these locations

Intel Installer | Docker | Apt, Yum | Conda | Intel® DevCloud

Learn More: software.intel.com/oneapi/ai-kit

Intel® AI Analytics Toolkit

Deep Learning

- Intel® Optimization for TensorFlow
- Intel® Optimization for PyTorch
- Intel® Neural Compressor
- Model Zoo for Intel® Architecture

Machine Learning

- Intel® Extension for Scikit-learn
- Intel-optimized XGBoost

Data Analytics

- Intel® Distribution of Modin
- OmniSci Backend

Intel-optimized Python

- NumPy
- SciPy
- Pandas
- Numba
- Python

CPU GPU

Hardware support varies by individual tool. Architecture support will be expanded over time.

Numpy – powered by oneAPI

- Stock version has oneAPI included
- Download AI Analytics toolkit here (get latest functionality here first)
- Are you getting the performance you expect using NumPy?
- Are you using NumPy effectively?
- Note: Keep NumPy library up to date
- Are you using NumPy effectively?

intel.

Python is Great & Fast …

python™

- For rapidly prototyping ideas
- Tackling just about every imaginable coding task
- Getting project rolling quickly … Examples of code are everywhere!
- Easy: Dynamically typed – making programming easy
- Easy & Fast: : Leverage huge number of libraries, easily installable
- For AI: fast/no porting: easy portability of models across architectures

Python is SLOW

- **For some things:**
    - **REPEATED** low level tasks
    - Large loops
    - Nested Loops
    - List comprehensions (if large)

- **BUT**
    - There are ways to mitigate its weaknesses
    - Take advantage of those libraries
    - **NUMPY** – this is **powered by oneAPI !**
    - And others!

NumPy Vectorization:

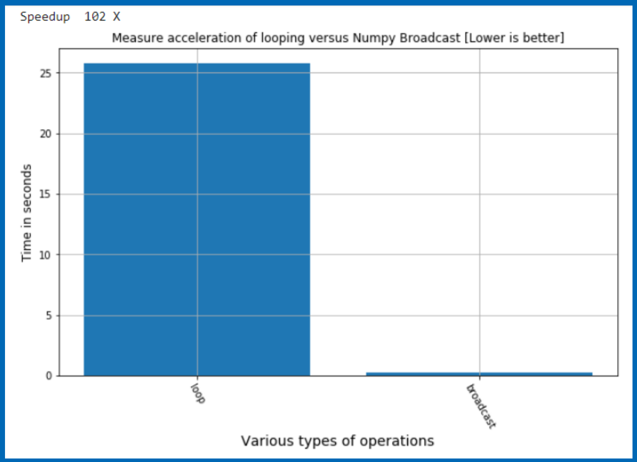
- This practice of replacing explicit loops with array expressions is commonly referred to as vectorization. In general, vectorized array operations will often be one or two (or more) orders of magnitude faster than their pure Python equivalents, with the biggest impact [seen] in any kind of numerical computations. - Wes McKinney (Pandas inventor)

Vectorization is NOT just theory!

You will **see**, **hear** about the speedups possible, then you will **experience** it in code

This is why we strongly encourage the use of libraries powered by Intel oneAPI such as NumPy, Scipy, and the rest

Get the goodness of Python but inherit vectorization speed inherent with NumPy powered by Intel oneAPI



100 X speed up using NumPy broadcasting versus loop

Why are these speedups so dramatic?

- NumPy takes advantage of vectorization: powered by Intel oneAPI
- Specifically, oneMKL, for vectorization
- Vector width allows multiple operations in single HW instruction.
- Many FP instructions
  computed in single instruction

AVX2: 256 bits: 8 floats wide

AVX512: 512 bits: 16 floats wide

Optimization Notice

We are comparing to simple loops in Python

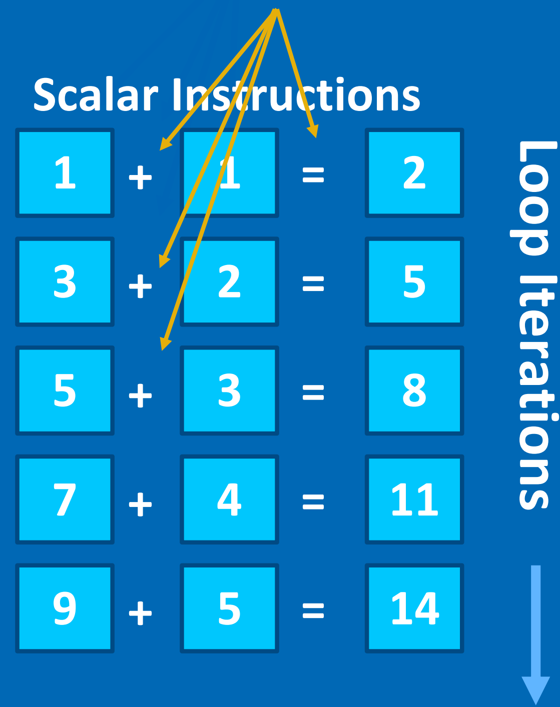
Python is dynamically typed

- Has to **check** the data type before any operation to ensure correct operations are applied

Even a simple integer is not simple

- A class or structure that contains
  - Reference counters and other values
  - These are updated every operation

In Python, these operations are COSTLY



Effect of Noncontiguous Memory Access

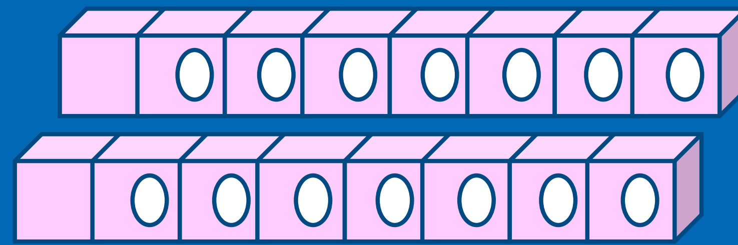
A list of integers in python are NOT generally in contiguous locations

For this list: [1, 2, 3, 4, 5, 6, 7, 8]

Accessing many of these in loops is VERY Costly (could be hundreds of clock cycles)

	5	7							
						3			
				8					
2									
							1		
		6							4

Cache is used ineffectively



Random reads from all over memory hurt performance.

Modern Intel CPU’s read in Cache line of consecutive memory so that consecutive data is already to go when needed. The cache line may contain 16 consecutive elements or more.

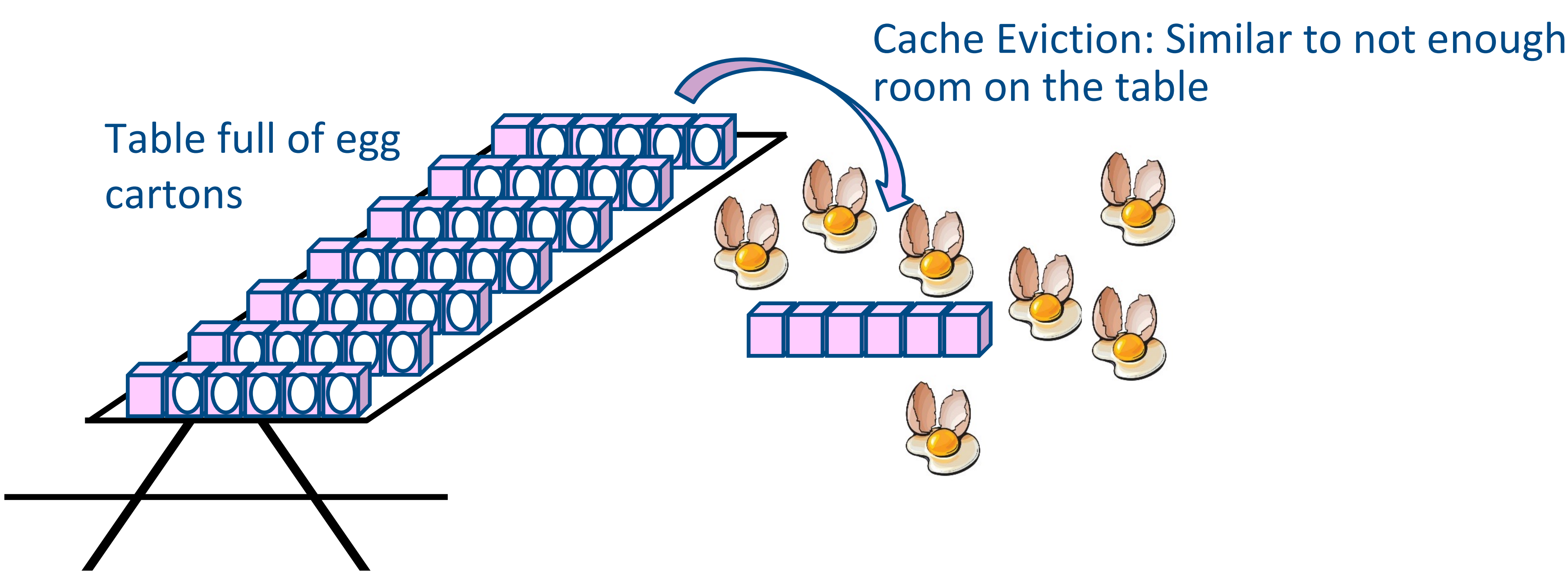
But with random reads, our next data element is read from a completely different place in memory – wasting the remaining elements that were ready to be served from the cache line.

This is analogous to a chef opening and cooking a single egg from a carton of 16 to service customer number 1.

Then opening a NEW carton of eggs from a SECOND carton for customer number 2. The other eggs get tossed out [analogous to cache line eviction].

These are SOME reasons why vectorization is better – it mitigates all the above.

Cache is used ineffectively with random memory accesses



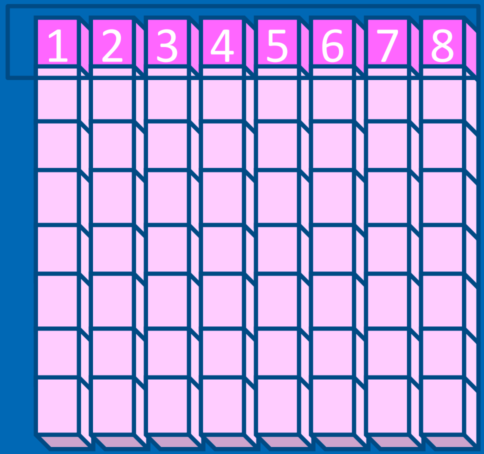
Additionally: concerning memory

An array of integers in NumPy are contiguous locations.

Modern CPU's return a cache line (like a carton of eggs) for the contiguous memory elements nearby one you chose to load initially.

The assumption is if I just used memory address 0x12345, then likely I will use 0x12346 very soon.

Contiguous memory addresses



How do I move my code patterns to NumPy?

- NumPy Vectorization encompasses…
- NumPy Universal Functions (Ufuncs)
- Aggregations
- Fancy Indexing
- Broadcasting
- NumPy Where & Select for Conditionals
- Are ALL loops vectorizable?

We will give guidance!

How to create NumPy arrays

- From existing lists:

```
np.array([1, 2, 3.0])
```

- By dimensions but empty:

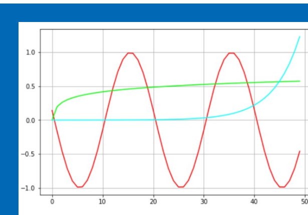
```
np.ndarray(shape=(2,2), dtype=float,
```

- By Shape – fill with zeros or ones:

```
>>> s = (2,2)
>>> np.zeros(s)
array([[ 0.,  0.],
       [ 0.,  0.]])
```

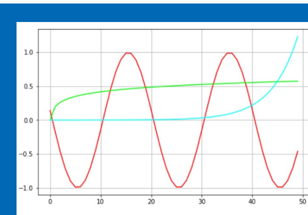
```
>>> s = (2,2)
>>> np.ones(s)
array([[1.,  1.],
       [1.,  1.]])
```

ufuncs



ufuncs are written in C (for speed) and linked into Python with NumPy's ufunc facility

Universal Functions	Description: These are vectorized
Math operations	**add, subtract, multiply, divide, reciprocal, matmul, log, exp, square, sqrt, …**
Trigonometric	sin, cos, tan, arcsin, arccos, arctan, hypot, sinh, cosh, tanh, degrees, radians
Bit-twiddling	bitwise_and, bitwise_or, bitwise_xor, invert, left_shift, right_shift
Comparison Functions	greater, greater_equal, less, less_equal, not_equal, equal, logical_and, logical_or, logical_xor, logical_not
Floating Functions	isfinite, isinf, isnan, isnat, fabs, signbit, copysign, nextafter, spacing, modf, ldexp, frexp, fmod, floor, ceil, trunc



NumPy Universal functions (ufunc): Vectorized!

- ufunc is a “vectorized” wrapper for a function
- Implements vectorization support in Intel AVX2 and AVX512
- Takes a fixed number of specific inputs
- Produces a fixed number of specific outputs
- Applies function in per element-wise fashion.
- For detailed information on universal functions, see Universal functions (ufunc) basics.

```
import numpy as np

arr = np.trunc([-3.1666, 3.6667])

print(arr)
out:
[-3.  3.]
```


intel.

Sophisticated Indexing

- Slicing and Indexing can replace many common loop concepts

```
a = np.arange(100_000_000)
t1 = time.time()
b =  np.arange(50_000_002)
N = len(a)
for i in range (N):
    if i % 2 == 0:
        b[i//2] = a[i]
t2 = time.time()
```

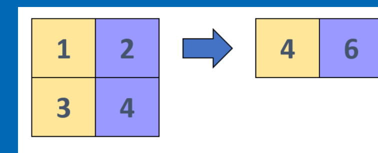
Results 4.5 X

```
b = a[::2]
```

NumPy Aggregation & Statistics Functions

Functions	Description: These are vectorized
np.mean()	Compute the arithmetic mean along the specified axis.
np.std()	Compute the standard deviation along the specified axis.
np.var()	Compute the variance along the specified axis.
np.sum()	Sum of array elements over a given axis.
np.prod()	Return the product of array elements over a given axis.
np.cumsum()	Return the cumulative sum of the elements along a given axis.
np.cumprod()	Return the cumulative product of elements along a given axis.
np.min(), np.max()	Return the minimum / maximum of an array or minimum along an axis.
np.argmin(), np.argmax()	Returns the indices of the minimum / maximum values along an axis
np.all()	Test whether all array elements along a given axis evaluate to True.
np.any()	Test whether any array element along a given axis evaluates to True.

NumPy Aggregations: Vectorized!

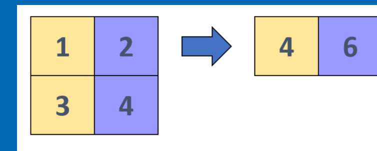


- Aggregation is an operation to reduce the dimensionality of an array or vector
- Implements vectorization support in Intel AVX2 and AVX512
- Replace loops you are using to compute averages, sums, standard deviation, min, max etc
- Use NumPy aggregation instead. Its more readable, faster, and future proof

```
A = [1 2 3]
for v in  range(len(A)):
     S += v
mean = S/len(A)
```

```
A = np.array(A)
mean = A.mean()
```

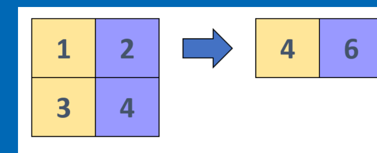
NumPy Aggregations: Vectorized!



Aggregations can be applied along different axes

```
      A
[[1 2 3
  4 5 6]]

A.sum(axis = 0)   ➡   array([5 7 9])

A.sum(axis = 1)   ➡   array([6 15])
```

Aggregation: Example




```python
a = np.arange(1_000_000).reshape(1000,1000)
t1 = time.time()
N = len(a)
sum = 0
for i in range (N):
    for j in range(N):
        sum += a[i,j]
t2 = time.time()
```

```python
sum = a.sum()
```

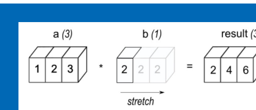
100X speed up over naïve loop

NumPy: Aggregation & Statistics

```
a = np.arange(100_000_000)
sum = []
s = 0
for i in range (N):
    s += a[i]
    sum.append(s)
```

```
sum = a.cumsum()
```

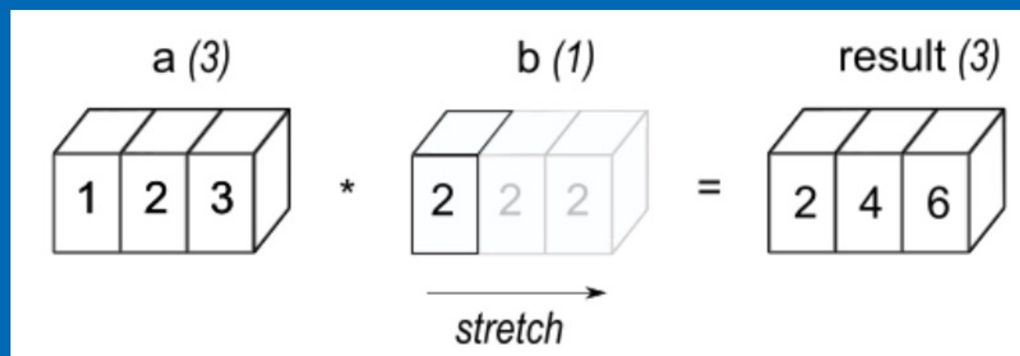
Result: 100 Million elements: roughly **13X**

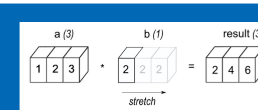


NumPy Broadcasting: Vectorized!

- Support with AVX2 and AVX512 instructions
- Apply an operator with a scalar to each element in vector
- Also, apply operator with lower dimension vector to larger dimension

```
a = np.array([1.0 2.0 3.0])
b = 2.0
a * b
    array([ 2. 4. 6.])
```





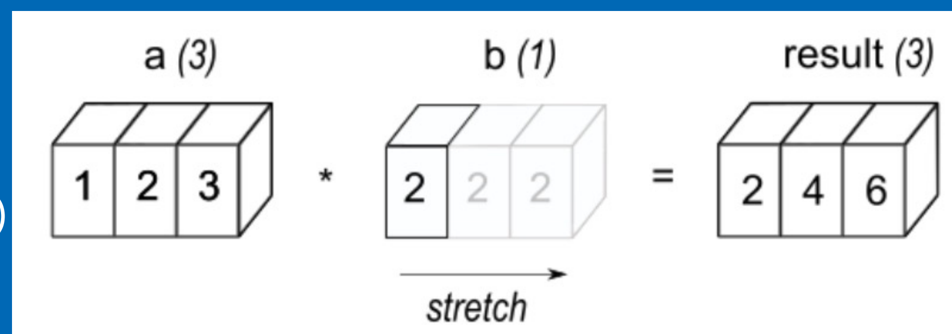
Broadcasting Graphically

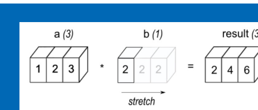
Non-matching dimensions are extended and data copied at HW level

Once dimensions match the vectors can be added, subtracted etc.

First example:

- a.shape (1, 3)
- b.shape (1,1) # extend/copy to (1, 3)
- result.shape (1, 3)





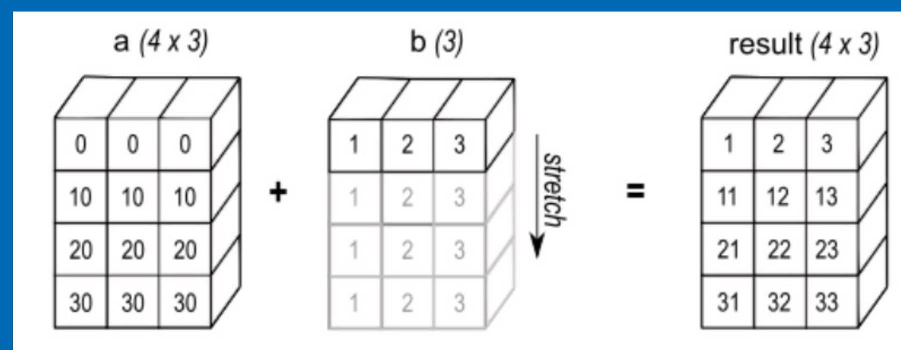
Broadcasting Graphically

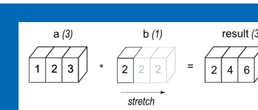
Non-matching dimensions are extended and data copied at HW level

Once dimensions match the vectors can be added, subtracted etc.

Second example:

- a.shape (4, 3)
- b.shape (1, 3) # extend/copy to (4, 3)
- result.shape (4, 3)





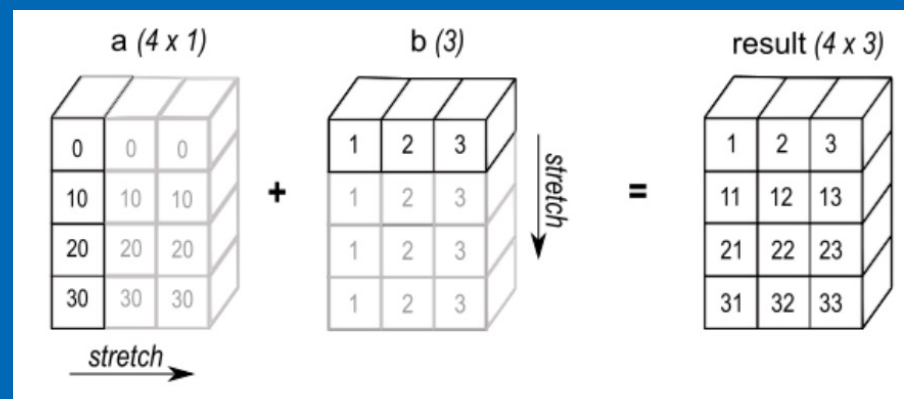
Broadcasting Graphically

Non-matching dimensions are extended and data copied at HW level

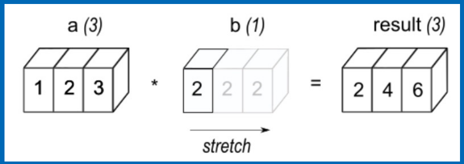
Once dimensions match the vectors can be added, subtracted etc.

Third example:

- a.shape (4, 1)
- b.shape (1, 3) # extend/copy to (4, 3)
- result.shape (4, 3)
-



Broadcasting Example

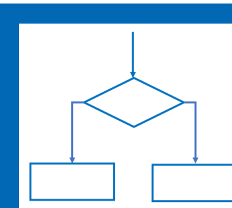


Simple Multiplication table

```
B = np.zeros((N,N))
for i in range(N):
    for j in range(N):
        B[i,j] = (i+1)*(j+1)
```

```
B = A.reshape(N,1) * A
```

numpy.where



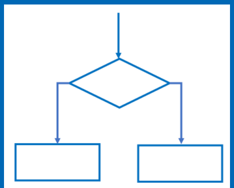
numpy.where: see : Return elements chosen from x or y depending on condition.

```
a = np.arange(10)
print("a\n",a)

np.where(a < 5, a, 10*a)
```

```
a
 [0 1 2 3 4 5 6 7 8 9]
array([ 0,  1,  2,  3,  4, 50, 60, 70, 80, 90])
```

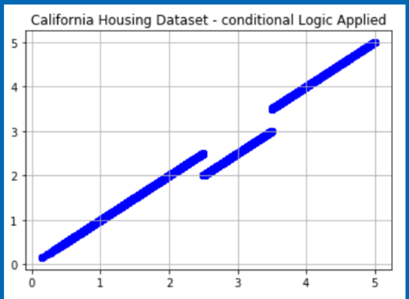

intel.

NumPy Where, NumPy Select



- If statements (conditional logic) might severely limit performance:
- Numpy: handles conditionals quickly, efficiently

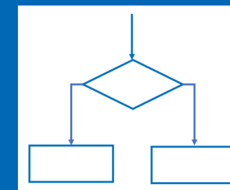
```
New = np.empty_like(T)
for i in range(len(T)):
    if ( (T[i] < buyerPriceRangeHi) & (T[i] >= buyerPriceRangeLo) ):
        New[i] = T[i] - 50_000/100_000
    else:
        New[i] = T[i]
```



Results: ~ **20 X**

```
New = np.where((T < buyerPriceRangeHi) & (T >= buyerPriceRangeLo), T - 50_000/100_000, T )
```

Numpy Where



Find row, col indices fast

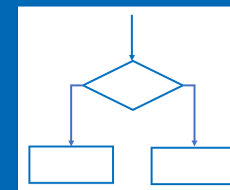
```
DBSCAN_array
 [[-1  2  1]
 [ 0  0  0]
 [ 0  1  2]
 [ 0 -1 -1]]
```

```python
DBSCAN_array = np.array([[-1,2,1],[0,0,0],[0,1,2],[0,-1,-1]])

print("\nDBSCAN_array\n",DBSCAN_array)

DBSCAN_Process = np.where(DBSCAN_array < 0,"Outlier()", "ProcessNormally()")

print("\nDBSCAN_Process\n",DBSCAN_Process)

# now we can find where all the ones are by row and column
print("row index (where outliers are): ",np.where(DBSCAN_array < 0)[0])
print("col index (where outliers are): ",np.where(DBSCAN_array < 0)[1])
```

```
DBSCAN_Process
 [['Outlier()' 'ProcessNormally()' 'ProcessNormally()']
 ['ProcessNormally()' 'ProcessNormally()' 'ProcessNormally()']
 ['ProcessNormally()' 'ProcessNormally()' 'ProcessNormally()']
 ['ProcessNormally()' 'Outlier()' 'Outlier()']]
```

```
row index (where outliers are):  [0 3 3]
col index (where outliers are):  [0 1 2]
```

Numpy.where: More Complex logic



```python
## one solution - preserves the indexing edges for easy checking
res = np.where( ( MultiplicationTable%12 == 0) | ( MultiplicationTable%9 == 0) , MultiplicationTable, 0)
res[0,:] = MultiplicationTable[0,:]
res[:,0] = MultiplicationTable[:,0]
```

```
[[  1,   2,   3,   4,   5,   6,   7,   8,   9,  10],
 [  2,   4,   6,   8,  10,  12,  14,  16,  18,  20],
 [  3,   6,   9,  12,  15,  18,  21,  24,  27,  30],
 [  4,   8,  12,  16,  20,  24,  28,  32,  36,  40],
 [  5,  10,  15,  20,  25,  30,  35,  40,  45,  50],
 [  6,  12,  18,  24,  30,  36,  42,  48,  54,  60],
 [  7,  14,  21,  28,  35,  42,  49,  56,  63,  70],
 [  8,  16,  24,  32,  40,  48,  56,  64,  72,  80],
 [  9,  18,  27,  36,  45,  54,  63,  72,  81,  90],
 [ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]]
```

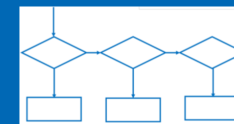
Results:

~ **20 X**

```
[[ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10],
 [ 2,  0,  0,  0,  0, 12,  0,  0, 18,  0],
 [ 3,  0,  9, 12,  0, 18,  0, 24, 27,  0],
 [ 4,  0, 12,  0,  0, 24,  0,  0, 36,  0],
 [ 5,  0,  0,  0,  0,  0,  0,  0, 45,  0],
 [ 6, 12, 18, 24,  0, 36,  0, 48, 54, 60],
 [ 7,  0,  0,  0,  0,  0,  0,  0, 63,  0],
 [ 8,  0, 24,  0,  0, 48,  0,  0, 72,  0],
 [ 9, 18, 27, 36, 45, 54, 63, 72, 81, 90],
 [10,  0,  0,  0,  0, 60,  0,  0, 90,  0]]
```


intel.

Numpy Select



Numpy.select: see

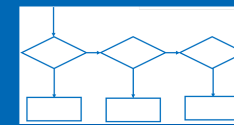
- Return an array drawn from elements in choice list, depending on conditions.
- Great for pulling together elements or functions(elements) from different arrays, DataFrames, different parts of the same array, etc

```python
x = np.arange(6)
print("X\n",x)

condlist = [x<3, x>3]

choicelist = [x, x**2]

np.select(condlist, choicelist, 42)
```

```
X
 [0 1 2 3 4 5]
array([ 0,  1,  2, 42, 16, 25])
```


intel

Numpy: Select Example



```python
for i in range(BIG):
    if A[i,4] == 10:
        A[i,5] =  A[i,2] * A[i,3]
    elif (A[i,4] < 10) and (A[i,4] >=5):
        A[i,5] =   A[i,2] + A[i,3]
    elif A[i,4] < 5:
        A[i,5] =   A[i,0] + A[i,1]
```

```python
condition = [ (A[:,4]  < 10) & (A[:,4] >= 5),
              ( A[:,4] < 5)]
choice = [ (A[:,2] + A[:,3]),
           (A[:,0] + A[:,1] ) ]
default = [(A[:,2] * A[:,3])]
A[:,5] = np.select(condition, choice, default= default )
```

Results: ~ **20 X**

The Pandas Connection

Pandas

- Pandas is built on top of Numpy
- All the methods describe before apply
- We will demonstrate alternative ways to achieve speedups when Pandas Apply is slow due to conditional logic in the custom called function

intel.

Methods

- Use “Apply” for simple functions to apply to columns
- When things get slow, convert data to NumPy arrays
-     to_numpy()
- Replace conditional logic in the Apply with numpy.where or numpy.select
- It is even possible to use numpy. select to manipulate Pandas dataframes directly

Optimization Notice

Using numpy.select as alternative to pandas.apply

```python
df['new'] = df.apply(lambda x: func(x['a'], x['b'], x['c'], x['d'], x['e']), axis=1)
```

```python
def my_function(x):
    return np.log(1+x)

def func(a,b,c,d,e):
    if e == 10:
        return c*d
    elif (e < 10) and (e>=7):
        return my_function(c+d)
    elif e < 7:
        return my_function(a+b+100)
```

Using numpy.select as alternative to pandas.apply

```
df['new'] = df.apply(lambda x: func(x['a'], x['b'], x['c'], x['d'], x['e']), axis=1)
```

Results:

200 X +

```
npArr = df.to_numpy()  # convert to numpy

condition = [ (npArr[:,idx['e']] < 10) & (npArr[:,idx['e']] >= 7),
              (npArr[:,idx['e']] < 7)]

choice = [(my_function(npArr[:,idx['c']] + npArr[:,idx['d']]      )),
          (my_function(npArr[:,idx['a']] + npArr[:,idx['b']] + 100))]

tmp = np.select(condition, choice, default= (npArr[:,idx['c']] * npArr[:,idx['d']]) )
df.loc[:,'new'] = tmp
```

Poll Audience
Live Demo/Lab

Optimization Notice

Call to Action

Loops:

- Find **large** single, double, and triple nested **loops** in your code and replace with Scipy/ Scikit-Learn*, or NumPy alternative
- Find time consuming list comprehensions and replace with a NumPy alternative .
- If statements:
- replace with numpy.where or numpy.select options if possible
- Using array masking that follows the conditional logic

Thanks for attending the session

BACKUP

Optimization Notice

Notices &

- This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.
- The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request. No product or component can be absolutely secure. Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at intel.com, or from the OEM or retailer.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.
- INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS". NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.
Notice revision #20110804

Intel(r) DevCloud testing specs

- Architecture: x86_64 CPU op-mode(s): 32-bit, 64-bit Byte Order: Little Endian Address sizes: 46 bits physical, 48 bits virtual CPU(s): 24 On-line CPU(s) list: 0-23 Thread(s) per core: 2 Core(s) per socket: 6 Socket(s): 2 NUMA node(s): 2 Vendor ID: GenuineIntel CPU family: 6 Model: 85 Model name: Intel(R) Xeon(R) Gold 6128 CPU @ 3.40GHz Stepping: 4 CPU MHz: 1200.254 CPU max MHz: 3700.0000 CPU min MHz: 1200.0000 BogoMIPS: 6800.00 Virtualization: VT-x L1d cache: 384 KiB L1i cache: 384 KiB L2 cache: 12 MiB L3 cache: 38.5 MiB NUMA node0 CPU(s): 0-5,12-17 NUMA node1 CPU(s): 6-11,18-23 Vulnerability Itlb multihit: KVM: Vulnerable Vulnerability L1tf: Mitigation; PTE Inversion Vulnerability Mds: Mitigation; Clear CPU buffers; SMT vulnerable Vulnerability Meltdown: Mitigation; PTI Vulnerability Spec store bypass: Mitigation; Speculative Store Bypass disabled v ia prctl and seccomp Vulnerability Spectre v1: Mitigation; usercopy/swapgs barriers and __user pointer sanitization Vulnerability Spectre v2: Mitigation; Full generic retpoline, IBPB condit ional, IBRS_FW, STIBP conditional, RSB filling Vulnerability Srbds: Not affected Vulnerability Tsx async abort: Mitigation; Clear CPU buffers; SMT vulnerable Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtr r pge mca cmov pat pse36 clflush dts acpi mmx f xsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rd tscp lm constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc cpuid aperf mperf pni pclmulqdq dtes64 monitor ds_cpl vmx s mx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid d ca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadli ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowprefetch cpuid_fault epb cat_l3 cdp_l3 inv pcid_single pti ssbd mba ibrs ibpb stibp tpr_sh adow vnmi flexpriority ept vpid ept_ad fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpci d rtm cqm mpx rdt_a avx512f avx512dq rdseed adx smap clflushopt clwb intel_pt avx512cd avx512b w avx512vl xsaveopt xsavec xgetbv1 xsaves cqm_l lc cqm_occup_llc cqm_mbm_total cqm_mbm_local dt herm ida arat pln pts hwp hwp_act_window hwp_ep p hwp_pkg_req pku ospke md_clear flush_l1d