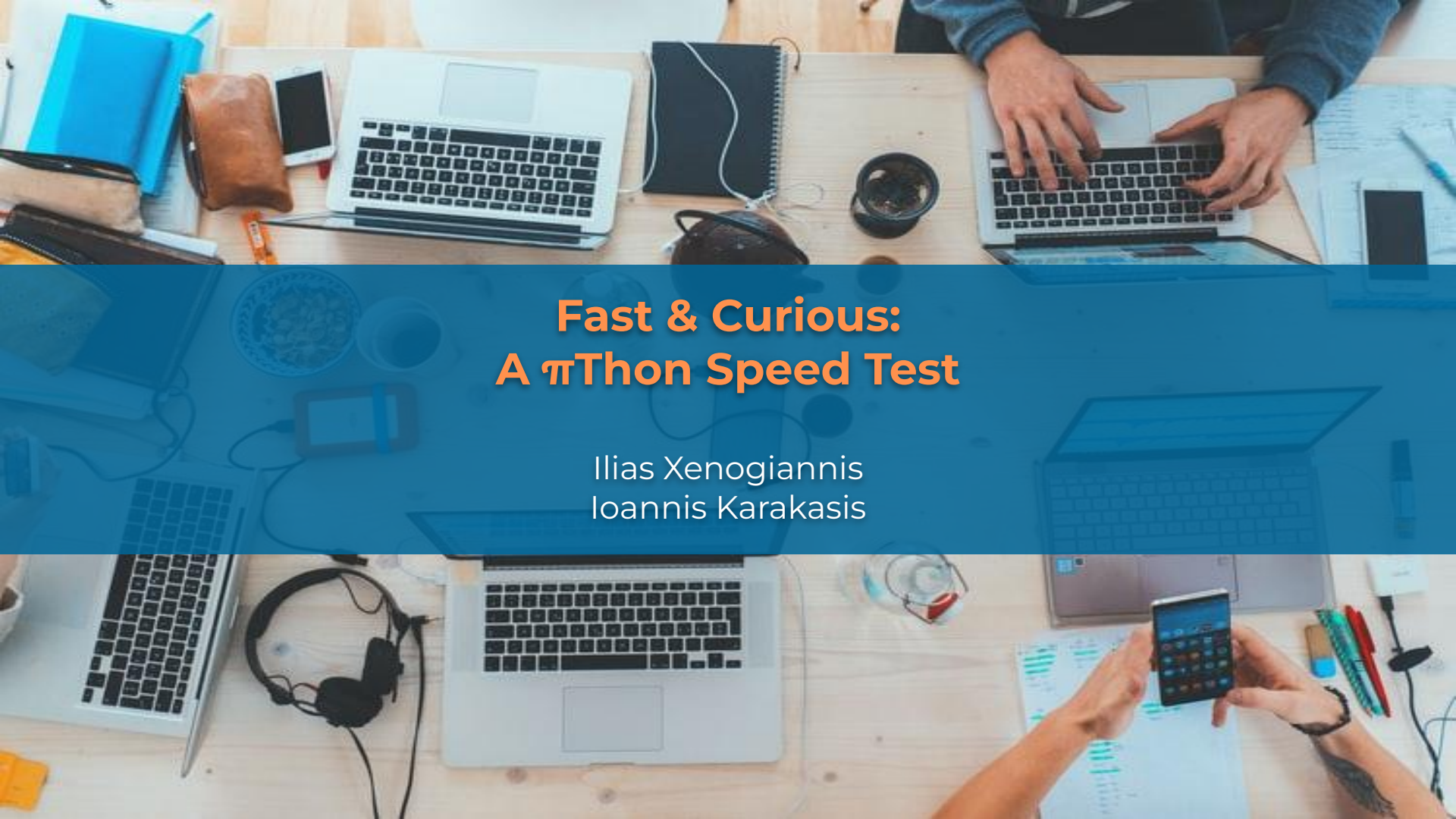
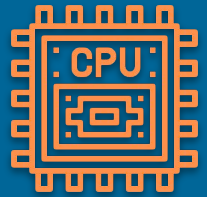


Fast & Curious:
A πThon Speed Test

Ilias Xenogiannis
Ioannis Karakasis

Agenda

Time



Space

Creating a list

Append

```
def list_creation_a(n):

    sum = []

    for i in range(0,n):
        sum.append(i**2)

    return sum
```

387 ms

Comprehension

```
def list_creation_b(n):

    return [i**2 for i in range(0,n)]
```

370 ms

x 1.04

Numpy

```
def list_creation_c(n):

    return np.arange(n) ** 2
```

5.46 ms

x 70.87

Summing a list

Index

```
def sum_list_A2(lista):
    sum = 0
    for i in range(0,len(lista)):
        sum = sum + lista[i]
    return sum
```

92.1 ms

In

```
def sum_list_A1(lista):
    sum = 0
    for i in lista:
        sum = sum + i
    return sum
```

58.1 ms

x 1.58

Python Built In

```
def sum_list_B1(lista):
    return sum(lista)
```

11.1 ms

x 8.29

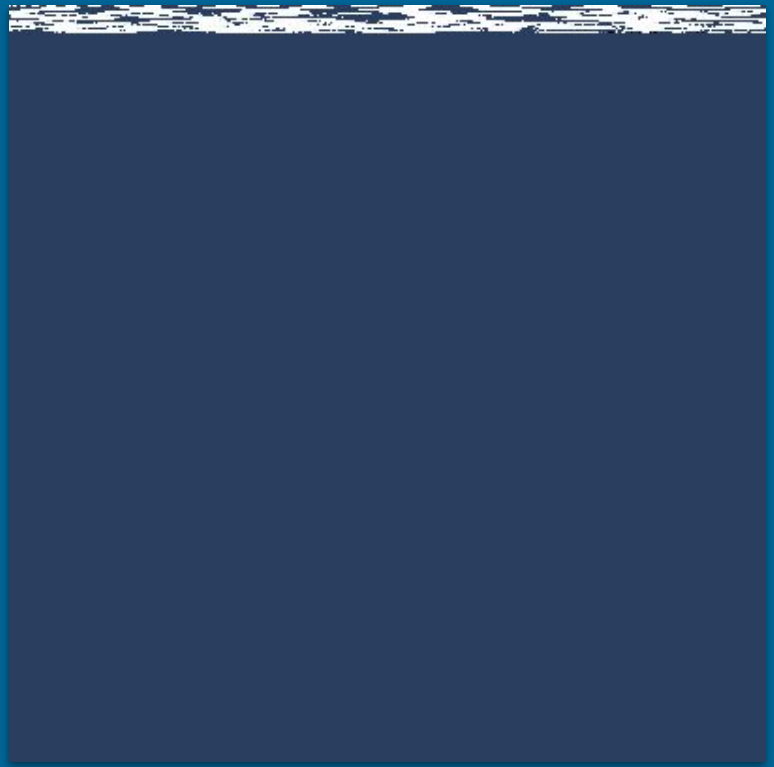
Numpy

```
def sum_list_B2(lista):
    return np.sum(lista)
```

1.3 ms

x 70.84

Monte Carlo



$$\pi = 4 * \frac{Count_{red}}{Count_{total}}$$

Monte Carlo Native Python

```
def is_inside_python_1(n):

    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1:
            count += 1

    return count
```

393 ms

Monte Carlo Pandas

```python
def is_inside_pandas_1(n):

    df = pd.DataFrame(np.random.uniform(0,1, size=(N, 2)), columns=list('XY'))

    sum = 0
    for i in range(0,len(df)):

        if (df.iloc[i]['X'] ** 2 + df.iloc[i]['Y'] ** 2 < 1 ):

            sum = sum + 1

    return sum
```

160000 ms

```python
def is_inside_pandas_2(n):

    df = pd.DataFrame(np.random.uniform(0,1, size=(N, 2)), columns=list('XY'))

    df.loc[(df['X'] ** 2 + df['Y'] ** 2 < 1),  'V'] = 1

    return df['V'].sum()
```

77.6 ms

Monte Carlo Numpy

```python
def is_inside_numpy_1(n):

    x = np.random.uniform(0,1, size=(n, 1))
    y = np.random.uniform(0,1, size=(n, 1))

    return np.where(x**2 + y**2 < 1, 1, 0 ).sum()
```

30.7 ms

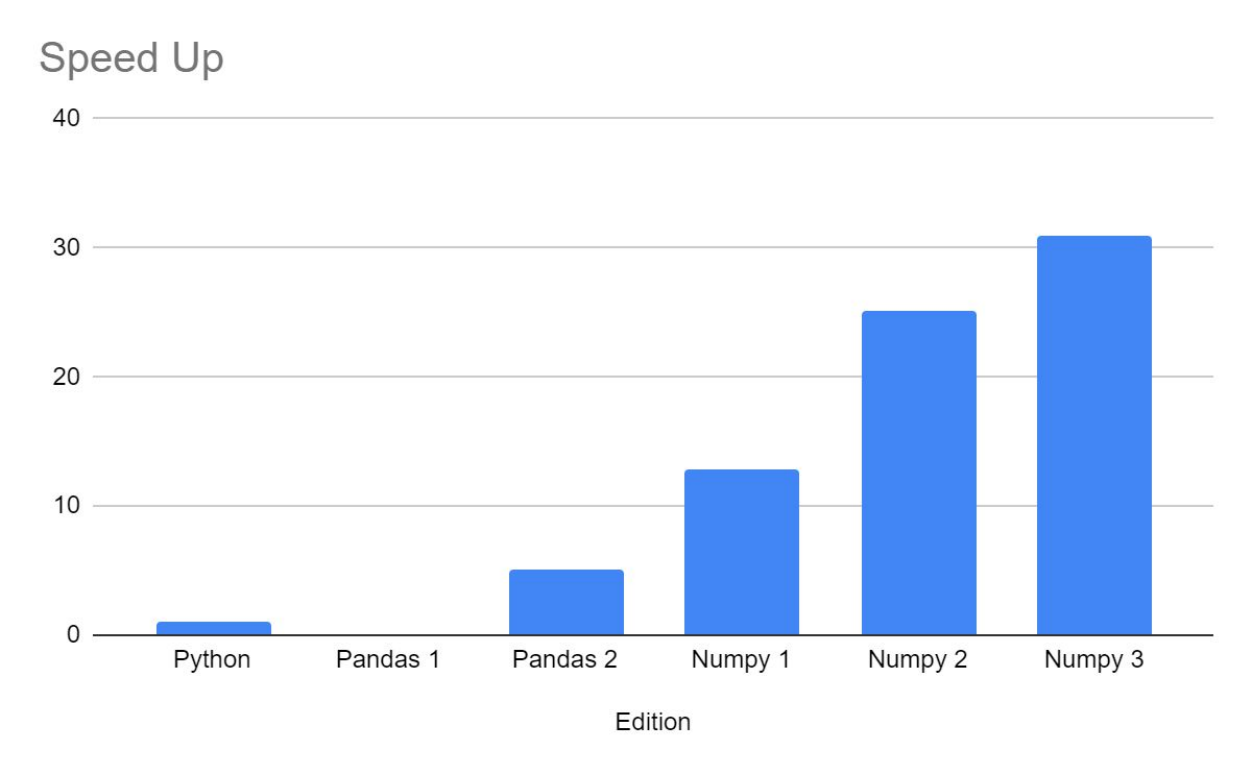
```python
def is_inside_numpy_2(n):

    rng = np.random.default_rng() #proper generator
    x = rng.random(size=(n, 1), dtype='float32')
    y = rng.random(size=(n, 1), dtype='float32')

    return np.where(x**2 + y**2 < 1, 1, 0 ).sum()
```

15.6 ms

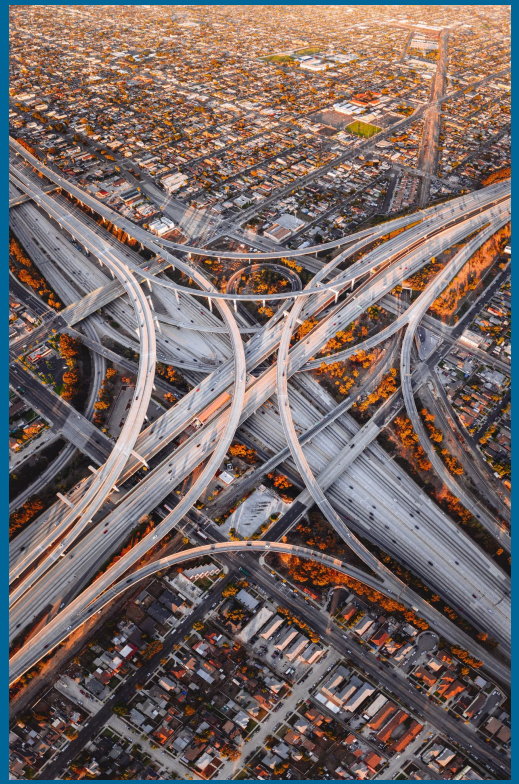
```python
def is_inside_numpy_3(n):

    rng = np.random.default_rng()
    x = rng.random(size=(n, 1), dtype='float32')
    y = rng.random(size=(n, 1), dtype='float32')
    nsuccesses = x**2 + y**2 < 1

    return  nsuccesses.sum()
```

12.7 ms

Monte Carlo Results



Is this all?



Multiprocessing

```python
def is_inside_python_1(n):

    count = 0
    for i in range(n):
        x = random.random()
        y = random.random()
        if x*x + y*y < 1:
            count += 1

    return count
```

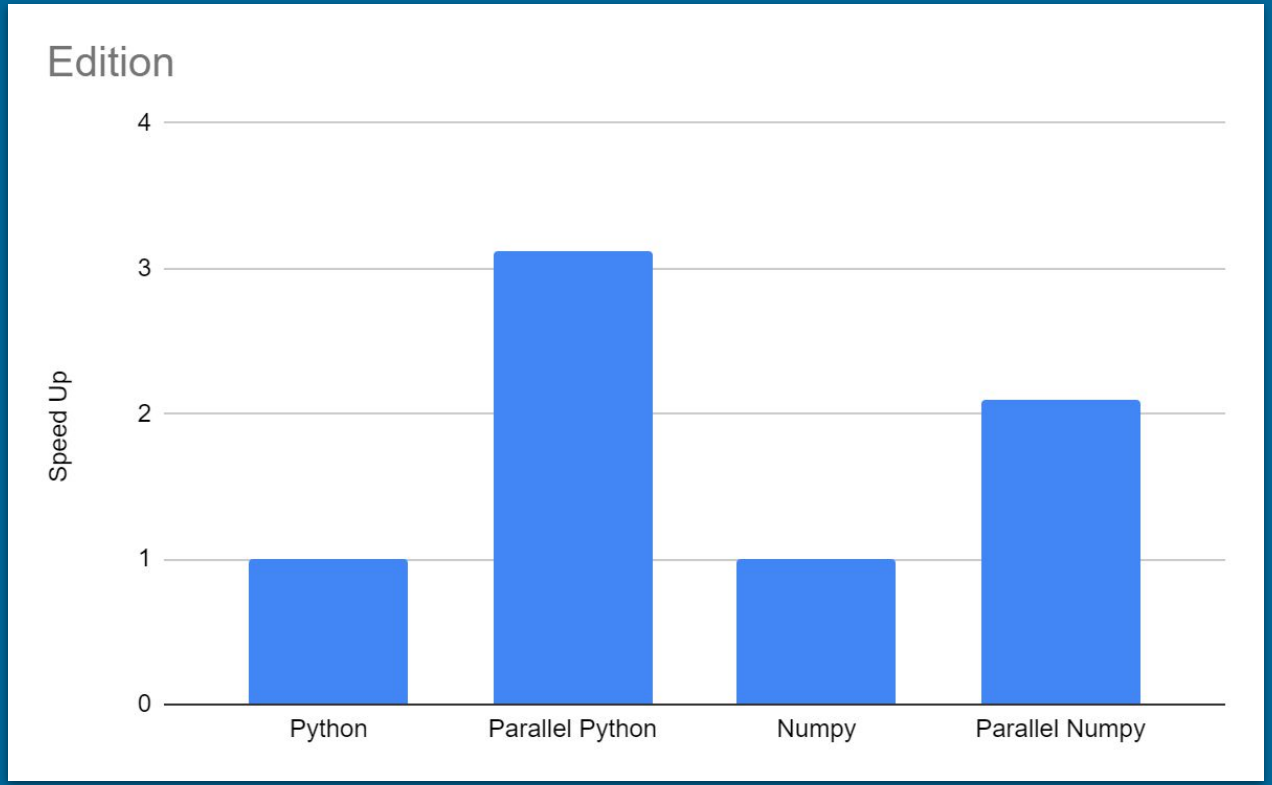
```python
def is_inside_numpy_3(n):

    rng = np.random.default_rng()
    x = rng.random(size=(n, 1), dtype='float32')
    y = rng.random(size=(n, 1), dtype='float32')
    nsuccesses = x**2 + y**2 < 1

    return  nsuccesses.sum()
```

Multiprocessing

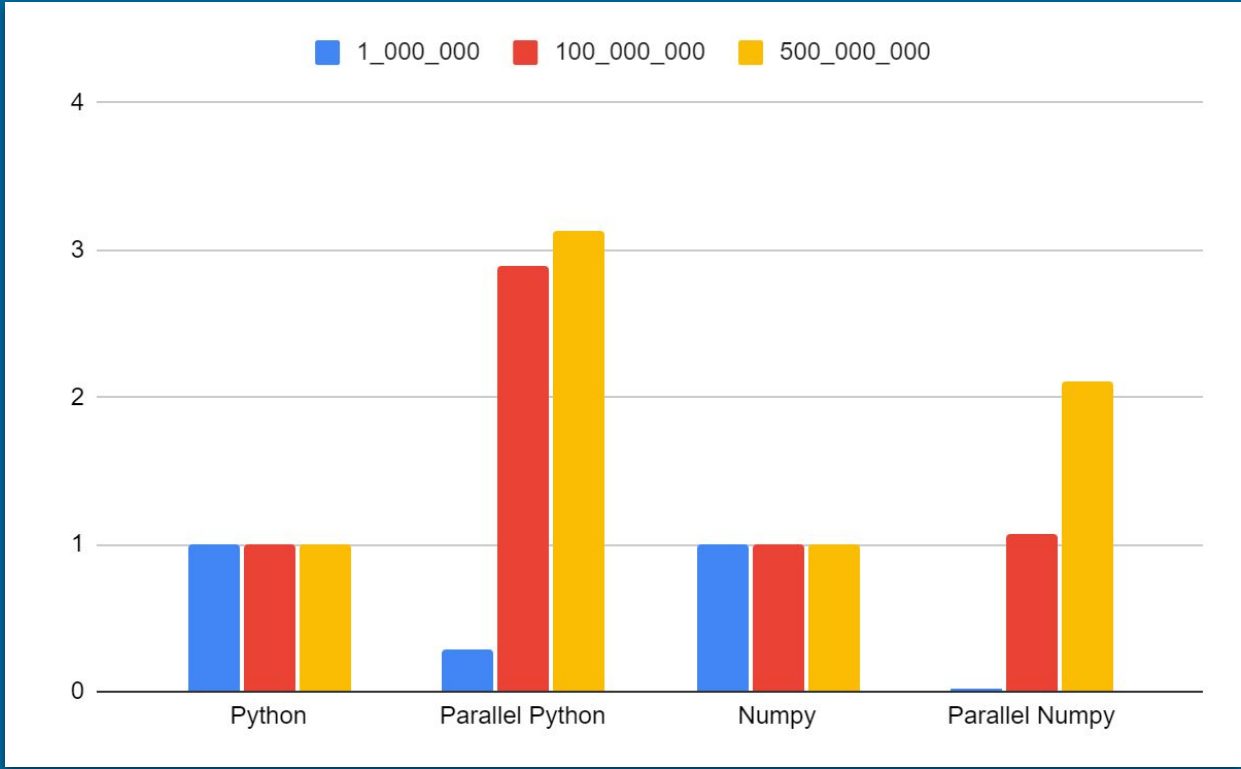
```python
def is_inside_python_parallel(n):

    np = multiprocessing.cpu_count()
    part_count=[int(n/np) for i in range(np)]

    with Pool(processes=4) as pool:
        count = pool.map(is_inside_python_1, part_count)

    return sum(count)
```

```python
def is_inside_numpy_parallel(n):

    np = multiprocessing.cpu_count()
    part_count = [int(n/np) for i in range(np)]

    with Pool(processes=4) as pool:
        count = pool.map(is_inside_numpy_3, part_count)

    return sum(count)
```

Results


Edition
Speed Up
4
3
2
1
0
Python
Parallel Python
Numpy
Parallel Numpy

Results


1_000_000
100_000_000
500_000_000
4
3
2
1
0
Python
Parallel Python
Numpy
Parallel Numpy

One more thing...

- Optimize the algorithm first
- Premature optimization is the root of all evil
- Respect your time more than the computers time
- Be aware of the overhead!

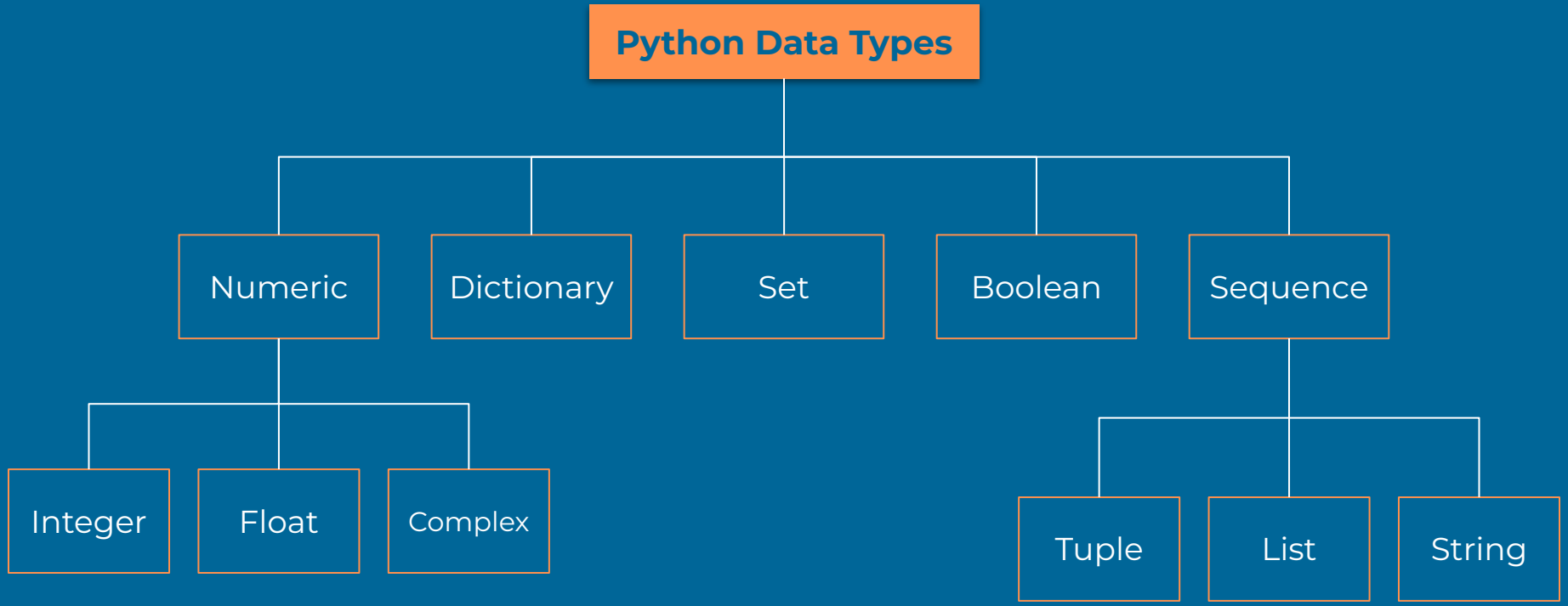
Next Steps

Use Memory space efficiently

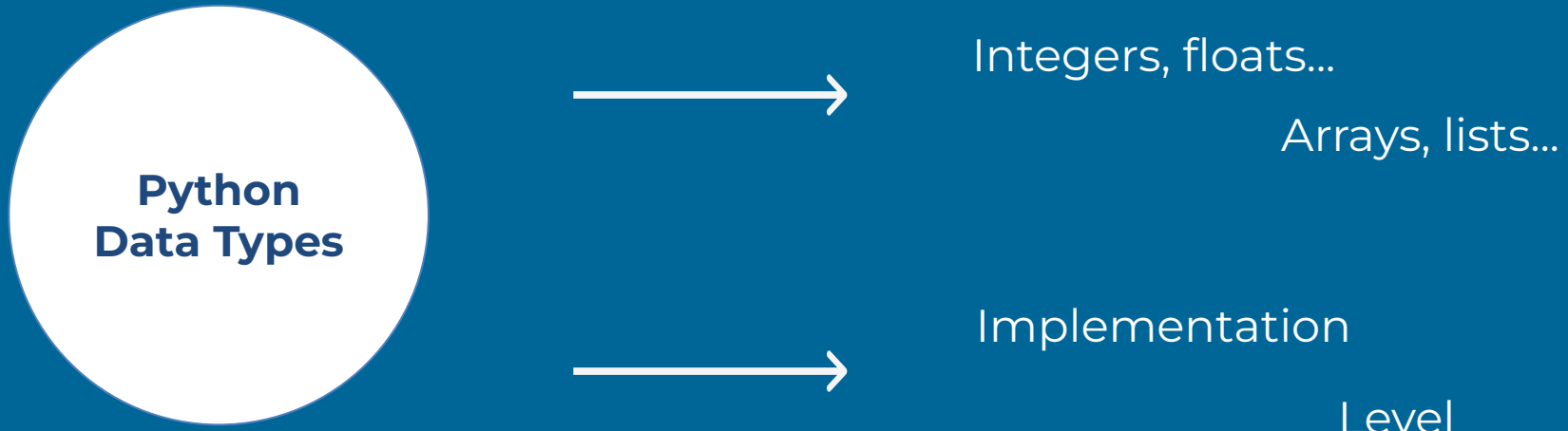
1st. A Stepback to the basics

1. Manage your data types
2. Use the most suitable file format
3. Use techniques to save memory space

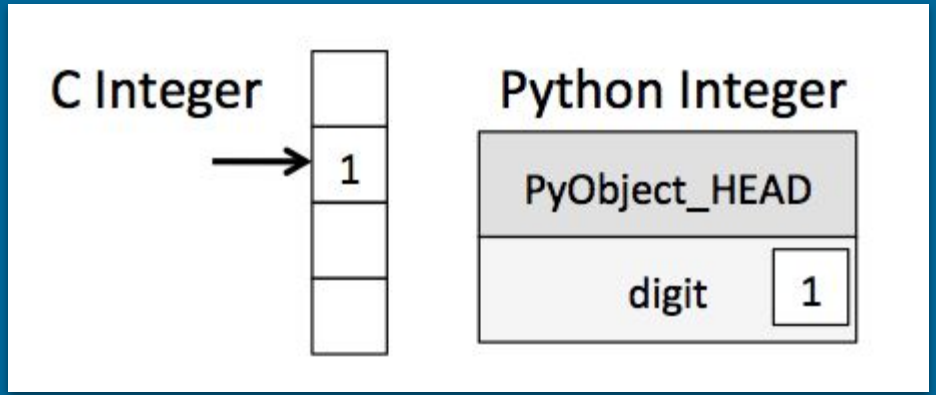
Implementation level



The truth about...

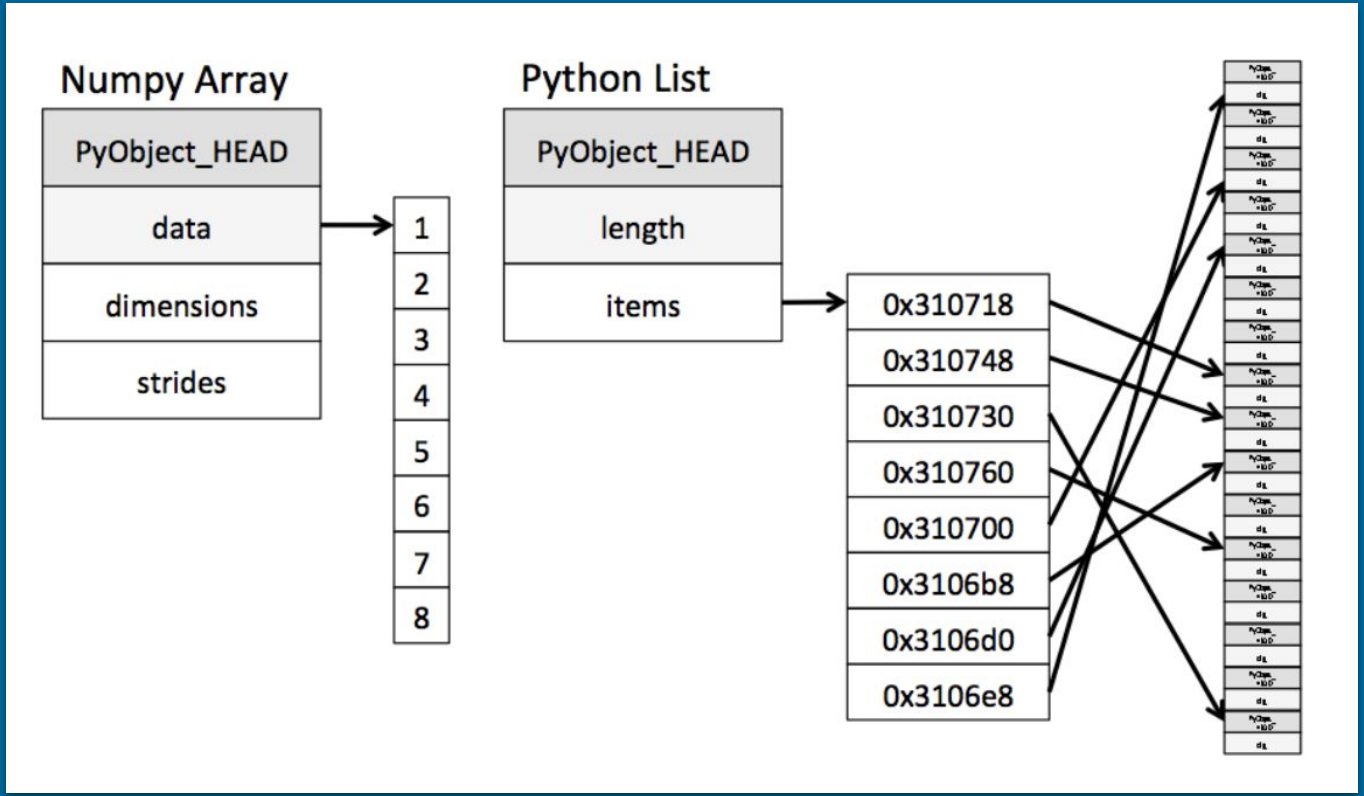


Storing an integer



- Reference count
- Type code
- Size code
- **Actual Value**

Implementation level



Numerical Subtypes

Character	Description	Example
`'b'`	Byte	`np.dtype('b')`
`'i'`	Signed integer	`np.dtype('i4') == np.int32`
`'u'`	Unsigned integer	`np.dtype('u1') == np.uint8`
`'f'`	Floating point	`np.dtype('f8') == np.int64`
`'c'`	Complex floating point	`np.dtype('c16') == np.complex128`
`'S'`, `'a'`	String	`np.dtype('S5')`
`'U'`	Unicode string	`np.dtype('U') == np.str_`
`'V'`	Raw data (void)	`np.dtype('V') == np.void`

Must check the implementation of floating points. Notice anything weird in some fractions?

Data type	Description
`bool_`	Boolean (True or False) stored as a byte
`int_`	Default integer type (same as C `long`; normally either `int64` or `int32`)
`intc`	Identical to C `int` (normally `int32` or `int64`)
`intp`	Integer used for indexing (same as C `ssize_t`; normally either `int32` or `int64`)
`int8`	Byte (–128 to 127)
`int16`	Integer (–32768 to 32767)
`int32`	Integer (–2147483648 to 2147483647)
`int64`	Integer (–9223372036854775808 to 9223372036854775807)
`uint8`	Unsigned integer (0 to 255)
`uint16`	Unsigned integer (0 to 65535)
`uint32`	Unsigned integer (0 to 4294967295)
`uint64`	Unsigned integer (0 to 18446744073709551615)
`float_`	Shorthand for `float64`
`float16`	Half-precision float: sign bit, 5 bits exponent, 10 bits mantissa
`float32`	Single-precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double-precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Shorthand for `complex128`
`complex64`	Complex number, represented by two 32-bit floats
`complex128`	Complex number, represented by two 64-bit floats

Numerical Subtypes

memory usage	float	int	uint	datetime	bool	object
1 byte		int8	uint8		bool	
2 bytes	float16	int16	uint16			
4 bytes	float32	int32	uint32			
8 bytes	float64	int64	uint64	datetime64		
variable						object

Lists, Tuples & Sets

> Important for Computationally expensive projects!

Tuples **&** **Sets**

are **FASTER** than

Lists

Internal Representation of DataFrames

index	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
0	1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
1	2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
2	3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
3	4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
4	5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

- Index Block
- Object Block
- Float Block

- Blocks don’t maintain references to column names
- Blockmanager Class
- Categorical data

A glimpse of Collections module

High Performance!

Collections → containers that are used to store collections of data such as list, dict and set.

Counter

In: Iterable
Out: Counter Dictionary

defaultdict

Optimized version of a dictionary

deque

Optimized version of queue

namedtuple

Tuple with fixed names

Some Best Practices

- Use Ravel / Reshape technique when using Numpy,
- Use slots when defining a Python Class,
- Use heapy for memory leaks
- Select type while reading, by removing the date column
- Next step...

Efficiency & File Formats

csv:
Spreadsheet file format

txt:
Plain Text
(Unstructured)

xlsx:
XML file format

json:
Designed for transmitting structured data

xml:
Human & Machine readable

HDF:
Independent of size & type of system

zip:
Archive file format

Manage to fit in Memory (RAM)

1. Compression
   - Lossless
   - Lossy
2. Chunking — Need to process all data but not at once
3. Indexing — Chunking → Filtering

Compression

Lossless

- Dropping Columns
- Lower-range numerical dtypes
- Categorical Data
- Sparse Columns

Lossy

- Changing numeric column representation
- Sampling

Chunking

- Separate elements into smaller groups
- Code that reads data / coded that processes data
- ~~Used in NLP (stemming / lemmatization)~~

Strategy for improving performance by using knowledge of a situation to aggregate related memory-allocation requests.

As long as each chunk fits in memory, on can work with datasets that are much larger than memory.

Works better when zero coordination between chunks is required.

Test Dataset

index	ID	Name	Sex	Age	Height	Weight	Team	NOC	Games	Year	Season	City	Sport	Event	Medal
0	1	A Dijiang	M	24.0	180.0	80.0	China	CHN	1992 Summer	1992	Summer	Barcelona	Basketball	Basketball Men's Basketball	NaN
1	2	A Lamusi	M	23.0	170.0	60.0	China	CHN	2012 Summer	2012	Summer	London	Judo	Judo Men's Extra-Lightweight	NaN
2	3	Gunnar Nielsen Aaby	M	24.0	NaN	NaN	Denmark	DEN	1920 Summer	1920	Summer	Antwerpen	Football	Football Men's Football	NaN
3	4	Edgar Lindenau Aabye	M	34.0	NaN	NaN	Denmark/Sweden	DEN	1900 Summer	1900	Summer	Paris	Tug-Of-War	Tug-Of-War Men's Tug-Of-War	Gold
4	5	Christine Jacoba Aaftink	F	21.0	185.0	82.0	Netherlands	NED	1988 Winter	1988	Winter	Calgary	Speed Skating	Speed Skating Women's 500 metres	NaN

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 271116 entries, 0 to 271115
Data columns (total 15 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   ID      271116 non-null  int64
 1   Name    271116 non-null  object
 2   Sex     271116 non-null  object
 3   Age     261642 non-null  float64
 4   Height  210945 non-null  float64
 5   Weight  208241 non-null  float64
 6   Team    271116 non-null  object
 7   NOC     271116 non-null  object
 8   Games   271116 non-null  object
 9   Year    271116 non-null  int64
 10  Season  271116 non-null  object
 11  City    271116 non-null  object
 12  Sport   271116 non-null  object
 13  Event   271116 non-null  object
 14  Medal   39783 non-null   object
dtypes: float64(3), int64(2), object(10)
memory usage: 31.0+ MB
```

Now we are ready to test it!

Select type while reading...

```
[6]  def memory_usage(df):
         return(round(df.memory_usage(deep=True).sum() / 1024 ** 2, 2))
```

```
[7]  init = memory_usage(df)
```

```
[8]  init

     177.77
```

```
[9]  df.info(memory_usage='deep')

     <class 'pandas.core.frame.DataFrame'>
     RangeIndex: 271116 entries, 0 to 271115
     Data columns (total 15 columns):
      #   Column  Non-Null Count   Dtype
     ---  ------  --------------   -----
      0   ID      271116 non-null  int64
      1   Name    271116 non-null  object
      2   Sex     271116 non-null  object
      3   Age     261642 non-null  float64
      4   Height  210945 non-null  float64
      5   Weight  208241 non-null  float64
      6   Team    271116 non-null  object
      7   NOC     271116 non-null  object
      8   Games   271116 non-null  object
      9   Year    271116 non-null  int64
      10  Season  271116 non-null  object
      11  City    271116 non-null  object
      12  Sport   271116 non-null  object
      13  Event   271116 non-null  object
      14  Medal   39783 non-null   object
     dtypes: float64(3), int64(2), object(10)
     memory usage: 177.8 MB
```

Select type while reading...

```
[ ] df.columns

    Index(['ID', 'Name', 'Sex', 'Age', 'Height', 'Weight', 'Team', 'NOC', 'Games',
           'Year', 'Season', 'City', 'Sport', 'Event', 'Medal'],
          dtype='object')
```

```
[11] df = df.fillna(0)

[12] df[['Name']] = df[['Name']].astype('string')

[13] df[['Sex','Team', 'NOC', 'Games','Year', 'Season', 'City',
         'Sport', 'Event', 'Medal']] =df[['Sex','Team', 'NOC', 'Games','Year', 'Season',
                                          'City', 'Sport', 'Event', 'Medal']].astype('category')
```

```
df[['Age','Height','Weight','ID']] = df[['Age','Height','Weight','ID']].astype('uint8')

[15] df[['ID']] = df[['ID']].astype('uint16')
```

Before VS After

```
[9] df.info(memory_usage='deep')

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 271116 entries, 0 to 271115
Data columns (total 15 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   ID      271116 non-null  int64
 1   Name    271116 non-null  object
 2   Sex     271116 non-null  object
 3   Age     261642 non-null  float64
 4   Height  210945 non-null  float64
 5   Weight  208241 non-null  float64
 6   Team    271116 non-null  object
 7   NOC     271116 non-null  object
 8   Games   271116 non-null  object
 9   Year    271116 non-null  int64
 10  Season  271116 non-null  object
 11  City    271116 non-null  object
 12  Sport   271116 non-null  object
 13  Event   271116 non-null  object
 14  Medal   39783 non-null   obj
dtypes: float64(3), int64(2), obj
memory usage: 177.8 MB
```

```
df.info(memory_usage='deep')

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 271116 entries, 0 to 271115
Data columns (total 15 columns):
 #   Column  Non-Null Count   Dtype
---  ------  --------------   -----
 0   ID      271116 non-null  uint8
 1   Name    271116 non-null  string
 2   Sex     271116 non-null  category
 3   Age     271116 non-null  uint8
 4   Height  271116 non-null  uint8
 5   Weight  271116 non-null  uint8
 6   Team    271116 non-null  category
 7   NOC     271116 non-null  category
 8   Games   271116 non-null  category
 9   Year    271116 non-null  category
 10  Season  271116 non-null  category
 11  City    271116 non-null  category
 12  Sport   271116 non-null  category
 13  Event   271116 non-null  category
     Medal   271116 non-null  category
s: category(10), string(1), uint8(4)
y usage: 24.4 MB
```

```
[19] after = memory_usage(df)

init/after

7.291632485643971
```

ABOUT US

John Karakasis


linkedin.com/in/john-karakasis

Ilias Xenogiannis


linkedin.com/in/ilias-xenogiannis


medium.com/@ixeno