TEXAS ADVANCED COMPUTING CENTER
WWW.TACC.UTEXAS.EDU

The University of Texas at Austin

Power Python 101

S. Charlie Dey, Director of Training and Professional Development

charlie@tacc.utexas.edu

Science in the Cloud, 2019

Agenda

- Introduction to the Jupyter Notebook
- Numpy Array vs Standard List
- Threads and Processors
- Vectorization
- Using Numpy with Threads
- Pandas
- Data Science

TACC

What are Jupyter Notebooks?

A web-based, interactive computing tool for capturing the whole computation process: developing, documenting, and executing code, as well as communicating the results.

How do Jupyter Notebooks Work?

An open notebook has exactly one interactive session connected to a kernel which will execute code sent by the user and communicate back results. This kernel remains active if the web browser window is closed, and reopening the same notebook from the dashboard will reconnect the web application to the same kernel.

What's this mean?
Notebooks are an interface to kernel, the kernel executes your code and outputs back to you through the notebook. The kernel is essentially our programming language we wish to interface with.

Jupyter Notebooks, Structure

- Code Cells

  Code cells allow you to enter and run code

  Run a code cell using Shift-Enter

- Markdown Cells

  Text can be added to Jupyter Notebooks using Markdown cells. Markdown is a popular markup language that is a superset of HTML.

Jupyter Notebooks, Structure

- Markdown Cells

You can add headings:

```
# Heading 1
# Heading 2
## Heading 2.1
## Heading 2.2
```

You can add lists

```
1. First ordered list item
2. Another item
··* Unordered sub-list.
1. Actual numbers don't matter, just that it's a number
··1. Ordered sub-list
4. And another item.
```

Jupyter Notebooks, Structure

- Markdown Cells

pure HTML

```
<dl>
  <dt>Definition list</dt>
  <dd>Is something people use sometimes.</dd>

  <dt>Markdown in HTML</dt>
  <dd>Does *not* work **very** well. Use HTML <em>tags</em>.</dd>
</dl>
```

And even, Latex!

```
$e^{i\pi} + 1 = 0$
```

Jupyter Notebooks, Workflow

Typically, you will work on a computational problem in pieces, organizing related ideas into cells and moving forward once previous parts work correctly. This is much more convenient for interactive exploration than breaking up a computation into scripts that must be executed together, as was previously necessary, especially if parts of them take a long time to run.

Jupyter Notebooks, Workflow

Let a traditional paper lab notebook be your guide:

Each notebook keeps a historical (and dated) record of the analysis as it's being explored.

The notebook is not meant to be anything other than a place for experimentation and development.

Notebooks can be split when they get too long.

Notebooks can be split by topic, if it makes sense.

Jupyter Notebooks, Shortcuts

- `Shift-Enter`: **run cell**
  - Execute the current cell, show output (if any), and jump to the next cell below. If `Shift-Enter` is invoked on the last cell, a new code cell will also be created. Note that in the notebook, typing `Enter` on its own *never* forces execution, but rather just inserts a new line in the current cell. `Shift-Enter` is equivalent to clicking the `Cell | Run` menu item.

TACC

Jupyter Notebooks, Shortcuts

- `Ctrl-Enter`: **run cell in-place**
  - Execute the current cell as if it were in "terminal mode", where any output is shown, but the cursor *remains* in the current cell. The cell's entire contents are selected after execution, so you can just start typing and only the new input will be in the cell. This is convenient for doing quick experiments in place, or for querying things like filesystem content, without needing to create additional cells that you may not want to be saved in the notebook.

Jupyter Notebooks, Shortcuts

- `Alt-Enter`: **run cell, insert below**
  - Executes the current cell, shows the output, and inserts a *new* cell between the current cell and the cell below (if one exists). (shortcut for the sequence `Shift-Enter`,`Ctrl-m a`. (`Ctrl-m a` adds a new cell above the current one.))
- `Esc` **and** `Enter`: **Command mode and edit mode**
  - In command mode, you can easily navigate around the notebook using keyboard shortcuts. In edit mode, you can edit text in cells.

TACC

Python - Variables, Refresh

in a code cell:

```
five = 5
one = 1
twodot = 2.0
print (five)
print (one + one)
message = “This is a string”
print (message)
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

Python – Data Types, refresh

```
integer_variable = 100
floating_point_variable = 100.0
string_variable = “Name”
```

Notice: We're not "typing" our variables, we're just setting them and allowing Python to type them for us.

TACC

Python – Data Types

Variables have a type

You can check the type of a variable by using the type() function:

```
print (type(integer_variable))
```

It is also possible to change the type of some basic types:

`str(int/float)`: **converts an integer/float to a string**
`int(str)`: **converts a string to an integer**
`float(str)`: **converts a string to a float**

Be careful: you can only convert data that actually makes sense to be transformed

Python – lists

A list is a sequence, where each element is assigned a position (index)
First position is 0. You can access each position using []
Elements in the list can be of different type

```
mylist1 = [“first item”, “second item”]
mylist2 = [1, 2, 3, 4]
mylist3 = [“first”, “second”, 3]
print(mylist1[0], mylist1[1])
print(mylist2[0])
print(mylist3)
print(mylist3[0], mylist3[1], mylist3[2])
print(mylist2[0] + mylist3[2])
```

Python – lists

It’s possible to use slicing:

```
print(mylist3[0:3])
print(mylist3)
```

To change the value of an element in a list, simply assign it a new value:

```
mylist3[0] = 10
print(mylist3)
```

Python – lists

There's a function that returns the number of elements in a list

```
len(mylist2)
```

Check if a value exists in a list:

```
1 in mylist2
```

Delete an element

```
len(mylist2)
del mylist2[0]
print(mylist2)
```

Iterate over the elements of a list:

```
for x in mylist2:
    print(x)
```

Python – lists

There are more functions

```
max(mylist), min(mylist)
```

It’s possible to add new elements to a list:

```
my_list.append(new_item)
```

We know how to find if an element exists, there’s a way to return the position of that element:

```
my_list.index(item)
```

Or how many times a given item appears in the list:

```
my_list.count(item)
```

Python – Anonymous Functions

type the following into a cell:

```
x = lamda a: a * 10

print (x(10))
```

TACC

Python – Anonymous Functions

try the following definition:

```
def myfunc(x):
   return lambda a: a*x

y = myfunc(10)
print (y(5))
z = myfunc(100)
print (z(5))
```

Python – NumPy

"Numerical Python"

open source extension module for Python
provides fast precompiled functions for mathematical and numerical routines
adds powerful data structures for efficient computation of multi-dimensional arrays and matrices.

NumPy, First Steps

Numpy gives us the array

```
import numpy as np
cvalues = [25.3, 24.8, 26.9, 23.9]
C = np.array(cvalues)
print(C)
```

NumPy, First Steps

And gives us an easier way to perform some simple math on them

```
print(C * 9 / 5 + 32)
```

vs.

```
fvalues = [ x*9/5 + 32 for x in cvalues]
print(fvalues)
```

NumPy, Multi-Dimension Arrays

```
A = np.array([ [3.4, 8.7, 9.9],
               [1.1, -7.8, -0.7],
               [4.1, 12.3, 4.8]])
print(A)
print(A.ndim)



B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B)
print(B.ndim)
```

NumPy, Multi-Dimension Arrays

The shape function:

```
x = np.array([ [67, 63, 87],
               [77, 69, 59],
               [85, 87, 99],
               [79, 72, 71],
               [63, 89, 93],
               [68, 92, 78]])
print(np.shape(x))
```

NumPy, Multi-Dimension Arrays

The shape function can also *change* the shape:

```
x.shape = (3, 6)
print(x)


x.shape = (2, 9)
print(x)
```

TACC

NumPy, Multi-Dimension Arrays

A couple more examples of shape:

```
x = np.array(42)
print(np.shape(x))

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B.shape)
```

NumPy, Multi-Dimension Arrays

indexing:

```
F = np.array([1, 1, 2, 3, 5, 8, 13, 21])

# print the first element of F, i.e. the element with the index 0
print(F[0])

# print the last element of F
print(F[-1])

B = np.array([ [[111, 112], [121, 122]],
               [[211, 212], [221, 222]],
               [[311, 312], [321, 322]] ])
print(B[0][1][0])
```

TACC

NumPy, Multi-Dimension Arrays

slicing:

```
A = np.array([
[11,12,13,14,15],
[21,22,23,24,25],
[31,32,33,34,35],
[41,42,43,44,45],
[51,52,53,54,55]])

print(A[:3,2:])


print(A[3:,:])
```

NumPy, Multi-Dimension Arrays

function to create an identity array

```
np.identity(4)
```

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that ∇2u = 0 with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def TimeStep(self, dt=0.0):
     """Takes a time step using straight forward Python loops."""
     g = self.grid
     nx, ny = g.u.shape
     dx2, dy2 = g.dx**2, g.dy**2
     dnr_inv = 0.5/(dx2 + dy2)
     u = g.u
     err = 0.0
     for i in range(1, nx-1):
         for j in range(1, ny-1):
             tmp = u[i,j]
             u[i,j] = ((u[i-1, j] + u[i+1, j])*dy2 +
                      (u[i, j-1] + u[i, j+1])*dx2)*dnr_inv
             diff = u[i,j] - tmp
             err += diff*diff
     return numpy.sqrt(err)
```

TACC

NumPy, By Example

The example we will consider is a very simple (read, trivial) case of solving the 2D Laplace equation using an iterative finite difference scheme (four point averaging, Gauss-Seidel or Gauss-Jordan). The formal specification of the problem is as follows. We are required to solve for some unknown function u(x,y) such that ∇2u = 0 with a boundary condition specified. For convenience the domain of interest is considered to be a rectangle and the boundary values at the sides of this rectangle are given.

```
def numericTimeStep(self, dt=0.0):
    """Takes a time step using a NumPy expression."""
    g = self.grid
    dx2, dy2 = g.dx**2, g.dy**2
    dnr_inv = 0.5/(dx2 + dy2)
    u = g.u
    g.old_u = u.copy() # needed to compute the error.

    # The actual iteration
    u[1:-1, 1:-1] = ((u[0:-2, 1:-1] + u[2:, 1:-1])*dy2 +
                     (u[1:-1,0:-2] + u[1:-1, 2:])*dx2)*dnr_inv

    return g.computeError()
```

NumPy, Exercise

Jacobi

```
Algorithm.

* Find D, the Diagonal of of A : diag(A)
* Find R, the Remainder of A - D : A - diagflat(A)

* Choose your initial guess, x[0]
    * Start iterating, k=0
        * While not converged do
           * Start your i-loop (for i = 1 to n)
               * sigma = 0
                * Start your j-loop (for j = 1 to n)
                   * If j not equal to i
                       * sigma = sigma + a[i][j] * x[j][k]
                 * End j-loop
               * x[i]k = (b[i] – sigma)/a[i][i] : x = (b - dot(R,x)) / D
           * End i-loop
        * Check for convergence
    * Iterate k, ie. k = k+1
```

TACC

Threads, Multithreading, Processes in a Nutshell

What is a thread?

A thread is a path of execution within a process.

A process can contain multiple threads.

What is a process?

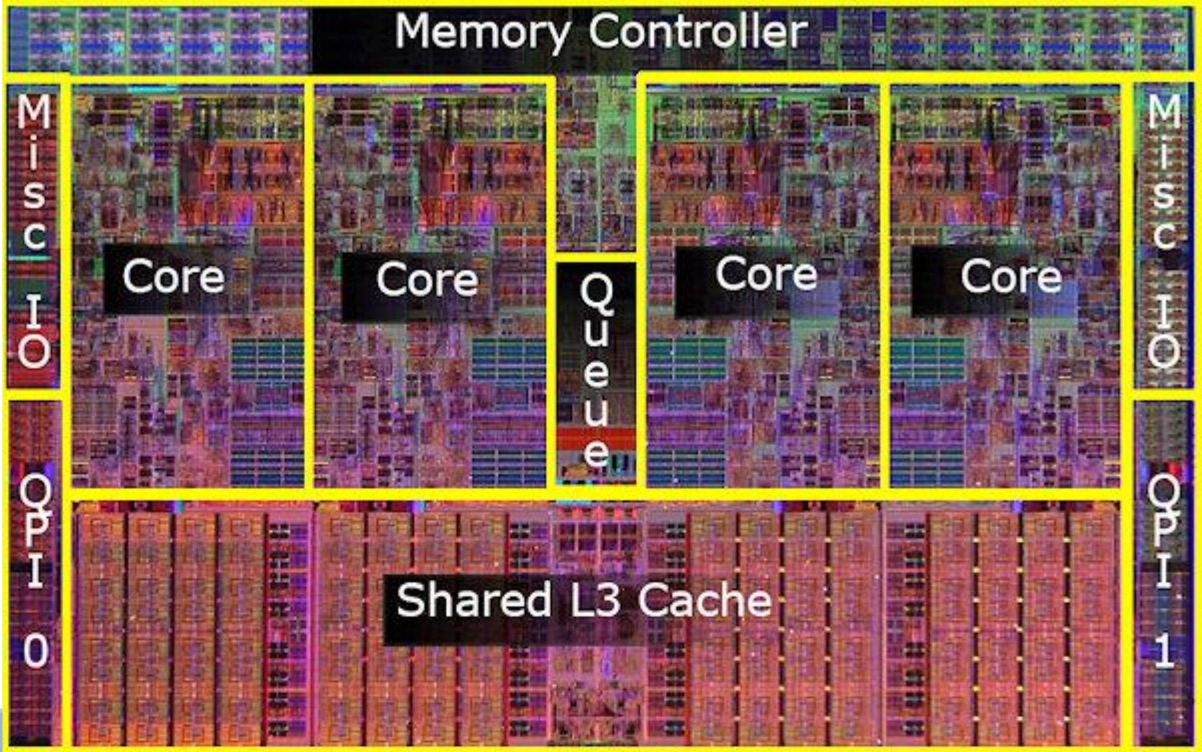
a process is the instance of a computer program that is being executed by one or many threads.

Threads, Multithreading, Processes in a Nutshell

Multithreading?

A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads.

The big picture



Advantages of Multithreading

1. ***Responsiveness****:* If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.

2. ***Faster context switch:*** Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.

3. ***Effective utilization of multiprocessor system****:* If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.

Advantages of Multithreading

4. ***Resource sharing****:* Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

5. ***Communication****:* Communication between multiple threads is easier, as the threads shares common address space. while in process we have to follow some specific communication technique for communication between two process.

6. ***Enhanced throughput of the system****:* If a process is divided into multiple threads, and each thread function is considered as one job, then the number of jobs completed per unit of time is increased, thus increasing the throughput of the system.

Parallel Programming in a Nutshell

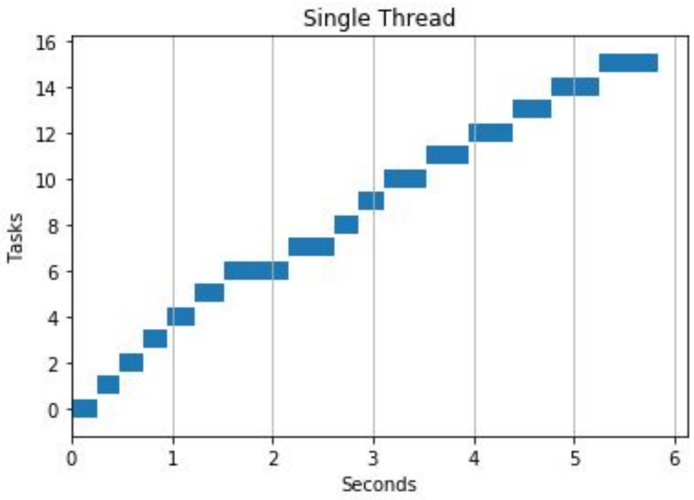
doing multiple things at the same time

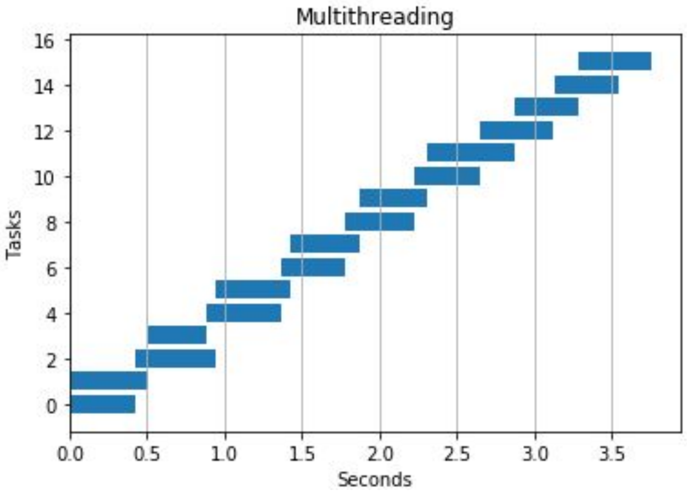
- running code simultaneously on different CPUs
- running code on the same CPU using multiple threads and achieving speedups by taking advantage of “wasted” CPU cycles

A note about parallelism in Python

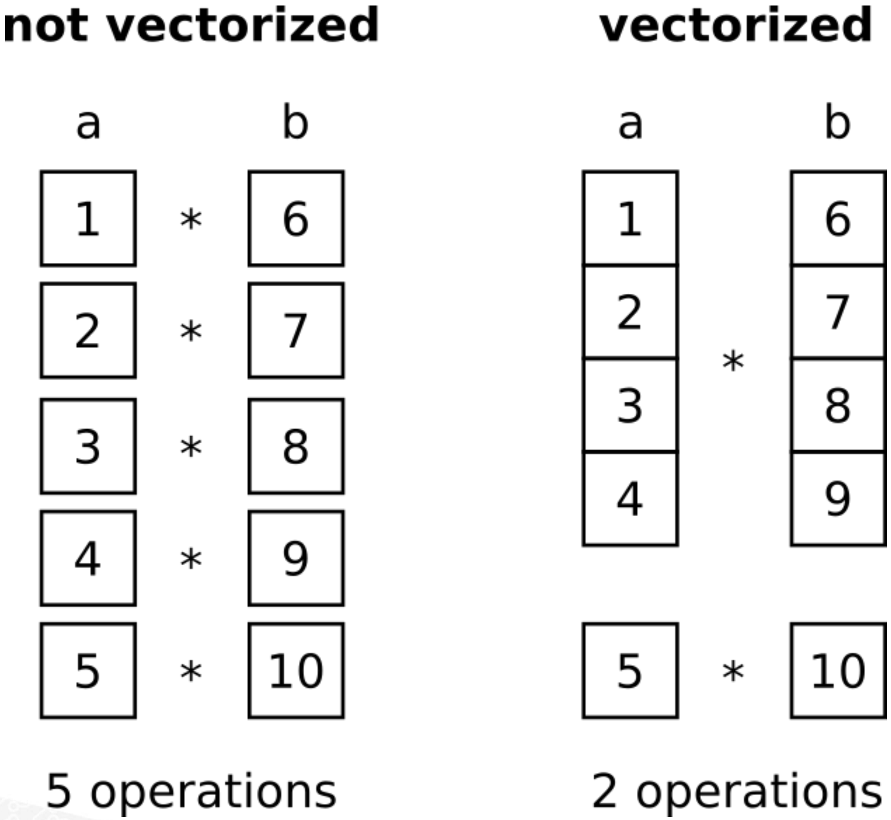
the **global interpreter lock** is a mutex - concurrency control, which is instituted for the purpose of preventing race conditions - that protects access to Python objects…

Parallel Programming in a Nutshell





Vectorization



NumPy, Vectorization

```
import time
size_of_vec = 1000
def pure_python_version():
    t1 = time.time()
    X = range(size_of_vec)
    Y = range(size_of_vec)
    Z = []
    for i in range(len(X)):
        Z.append(X[i] + Y[i])
    return time.time() - t1
```

```
def numpy_version():
    t1 = time.time()
    X = np.arange(size_of_vec)
    Y = np.arange(size_of_vec)
    Z = X + Y
    return time.time() - t1
```

NumPy, Cooler things

Let's see which is faster.

```
t1 = pure_python_version()
t2 = numpy_version()
print(t1, t2)
```

Pandas, What is it?

A software library written for the Python for data manipulation and analysis. In particular, it offers data structures and operations for manipulating numerical tables and time series

Pandas, The DataFrame

The primary pandas data structure.
Two-dimensional size-mutable, heterogeneous tabular data structure with labeled axes (rows and columns). Arithmetic operations align on both row and column labels. Can be thought of as a dict-like container for Series objects.

Pandas, *First Steps*

Let's create a simple data set, and see what Pandas can do.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

TACC

Pandas, *First Steps*

Let's create a simple data set, and see what Pandas can do.

```
s = pd.Series([1,3,5,np.nan,6,8])
s
```

***Pandas**, First Steps*

Let's create a simple data set, and see what Pandas can do.

```
dates = pd.date_range('20180101', periods=6)
dates
```

Pandas, *First Steps*

Let's create a simple data set, and see what Pandas can do.

```
df = pd.DataFrame(np.random.randn(6,4),
 index=dates, columns=list('ABCD'))
df
```

Pandas, First Steps

Let's create a simple data set, and see what Pandas can do.

```
df2 = pd.DataFrame({ 'A' : 1.,'B' :
 pd.Timestamp('20130102'),'C' :
 pd.Series(1,index=list(range(4)),dtype='float32'),'D' :
 np.array([3] * 4,dtype='int32'),'E' :
 pd.Categorical(["test","train","test","train"]),'F' :
 'foo' })

df2
```

Pandas, Viewing Data

Some common/useful functions

```
df.head()
df.tail(3)
df.index
df.columns
df.values
df.describe()
df.T
df.sort_index(axis=1, ascending=False)
df.sort_values(by='B')
```

Pandas, Selecting Data by Label

Some common/useful functions

```
df['A']
df[0:3]
df['20130102':'20130104']
df.loc[dates[0]]
df.loc[:,['A','B']]
df.loc['20130102':'20130104',['A','B']]
df.loc['20130102',['A','B']]
df.loc[dates[0],'A']
```

TACC

Pandas, Selecting Data by Position

Some common/useful functions

```
df.iloc[3]
df.iloc[3:5,0:2]
df.iloc[[1,2,4],[0,2]]
df.iloc[1:3,:]
df.iloc[:,1:3]
df.iloc[1,1]
df.iat[1,1]
```

Pandas, CSV Files

manipulating CSV files.

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range('1/1/2000',
 periods=1000))
ts = ts.cumsum()  ## cumulative sum


df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,columns=['A',
 'B', 'C', 'D'])
df = df.cumsum()

df.to_csv('foo.csv')
pd.read_csv('foo.csv')
```

Pandas, CSV Files

filtering data made easy… return of lambda

```
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index,columns=['A',
'B', 'C', 'D'])

df = df.cumsum()

df.loc[lambda df: df.B > 10]  ## What do you think this does?
```

TACC

Pandas, Joining

Dataset 1

```
raw_data = {
        'subject_id': ['1', '2', '3', '4', '5'],
        'first_name': ['Alex', 'Amy', 'Allen', 'Alice', 'Ayoung'],
        'last_name': ['Anderson', 'Ackerman', 'Ali', 'Aoni', 'Atiches']}

df_a = pd.DataFrame(raw_data, columns = ['subject_id', 'first_name',
'last_name'])

df_a
```

Pandas, Joining

Dataset 2

```
raw_data = {
        'subject_id': ['4', '5', '6', '7', '8'],
        'first_name': ['Billy', 'Brian', 'Bran', 'Bryce', 'Betty'],
        'last_name': ['Bonder', 'Black', 'Balwner', 'Brice', 'Btisan']}

df_b = pd.DataFrame(raw_data, columns = ['subject_id', 'first_name',
'last_name'])

df_b
```

Pandas, Joining

Dataset 3

```
raw_data = {
        'subject_id': ['1', '2', '3', '4', '5', '7', '8', '9', '10', '11'],
        'test_id': [51, 15, 15, 61, 16, 14, 15, 1, 61, 16]}

df_n = pd.DataFrame(raw_data, columns = ['subject_id','test_id'])


df_n
```

Pandas, Joining

Joining along rows

```
df_new = pd.concat([df_a, df_b])
df_new
```

TACC

Pandas, Joining

Joining along columns

```
d.concat([df_a, df_b], axis=1)
```

TACC

Pandas, Joining

Merging

```
pd.merge(df_new, df_n, on='subject_id')
```

Pandas, Joining

Merging, Outer Join

```
pd.merge(df_a, df_b, on='subject_id', how='outer')
```

Pandas, Joining

Merging, Inner Join

```
pd.merge(df_a, df_b, on='subject_id', how='inner')
```

Pandas, Joining

Merging, Right Join

```
pd.merge(df_a, df_b, on='subject_id', how='right')
```

Pandas, Joining

Merging, Left Join

```
pd.merge(df_a, df_b, on='subject_id', how='left')
```

Pandas, Summary of Features

Pandas allow for:

Boolean Indexing
Statistical Operations
Histogramming
Merging Data
SQL Style Joins
SQL Style Appends
SQL Style Grouping
Reshaping
Pivoting
and more!

TEXAS ADVANCED COMPUTING CENTER

WWW.TACC.UTEXAS.EDU

The University of Texas at Austin

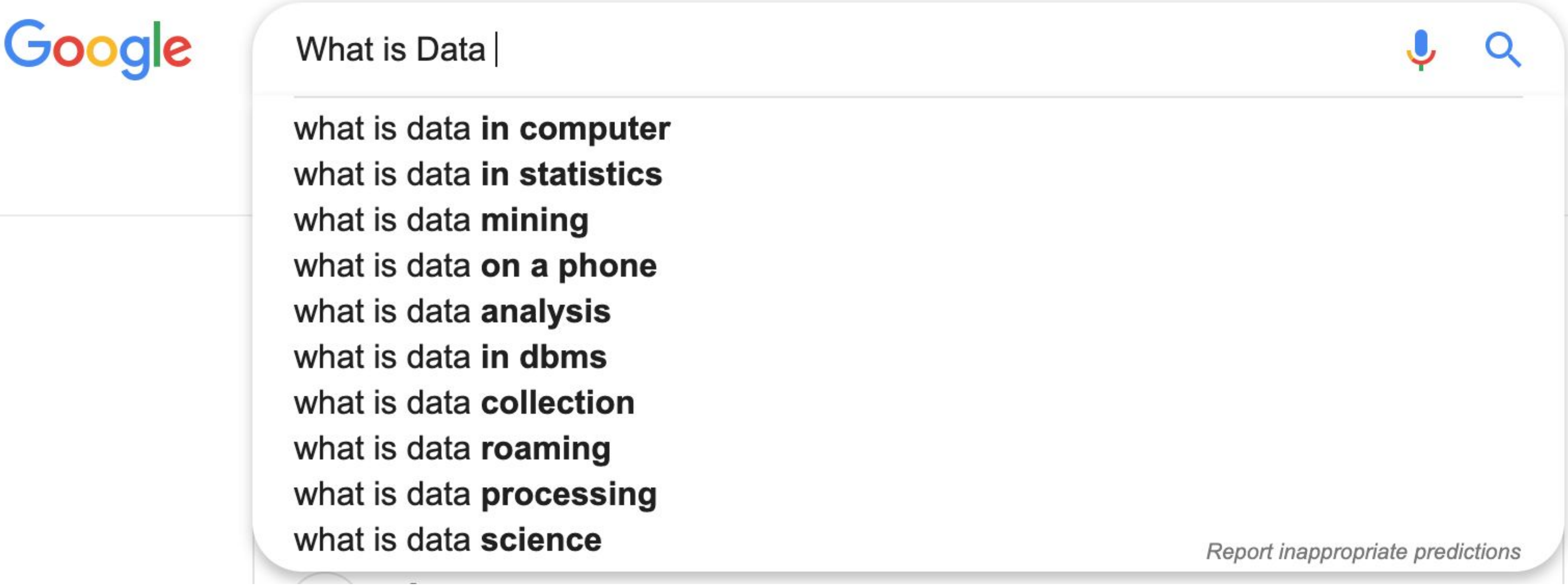
What is Data Science

PRESENTED BY:

Data Science 101

What is Data?

Data Science 101



Data Science 101

Data is a set of values of subjects with respect to qualitative or quantitative variables. Data and information or knowledge are often used interchangeably; however data becomes information when it is viewed in context or in post-analysis. Wikipedia

Data Science 101

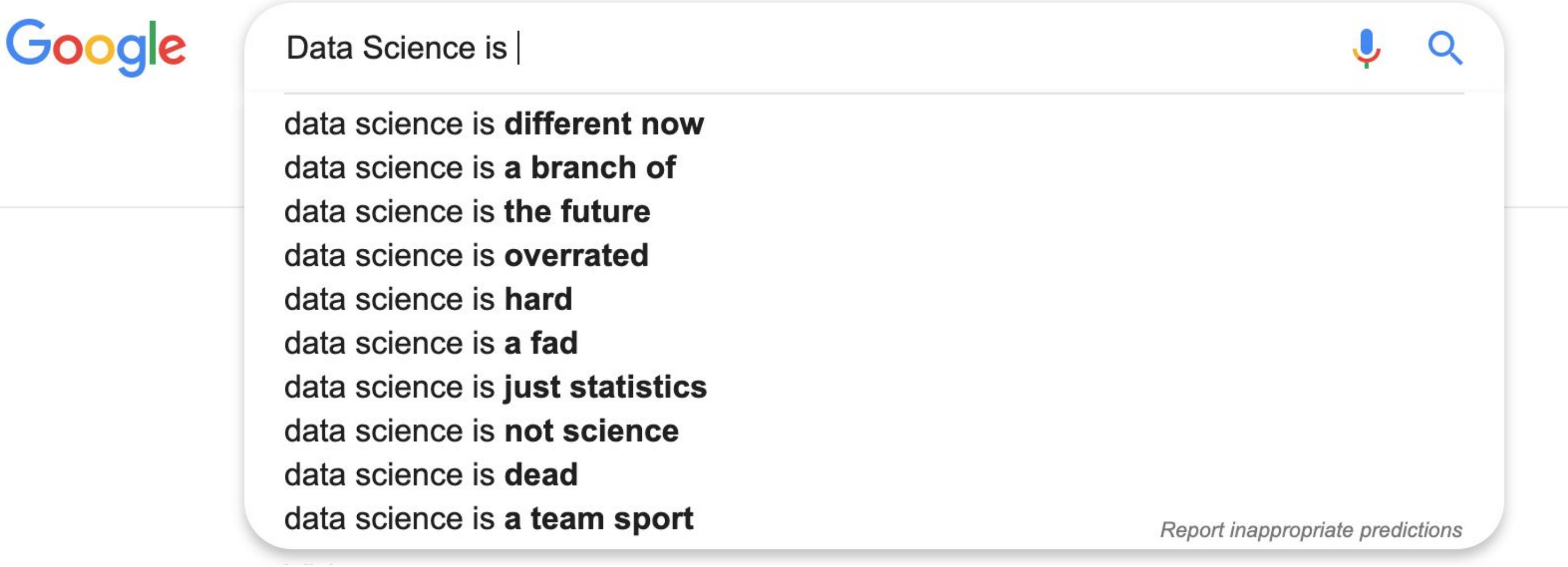
Data is everywhere!

Where is your data?

Data Science 101

What makes data important?

Data Science 101



Data Science 101

Data science is a multi-disciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data. Wikipedia

Data Science 101

Data Science is the ability to understand that there is a story hidden in the data.

Data Science 101

100 Million Dollars - Southwest Airlines saved by reducing the time their airplanes sat idle on the tarmac

39 Million Gallons - the amount of fuel UPS saved by optimizing its fleet

32,000 Dollars the amount of money it costs TACC to have our machines sitting idle

Data Science 101

Data is worth money.

Data Science 101

BIG DATA

There isn't a readily available definition of Big Data because you can't "see it"

Examples of Big Data?

Data Science 101

We are in the era of Big Data

There was a road to get to this moment with a few important stops along the way, and it's a road on which we're probably still nowhere near the end. To get to the data driven world we have today, we needed **scale**, **speed**, and **ubiquity**.

Data Science 101

Scale

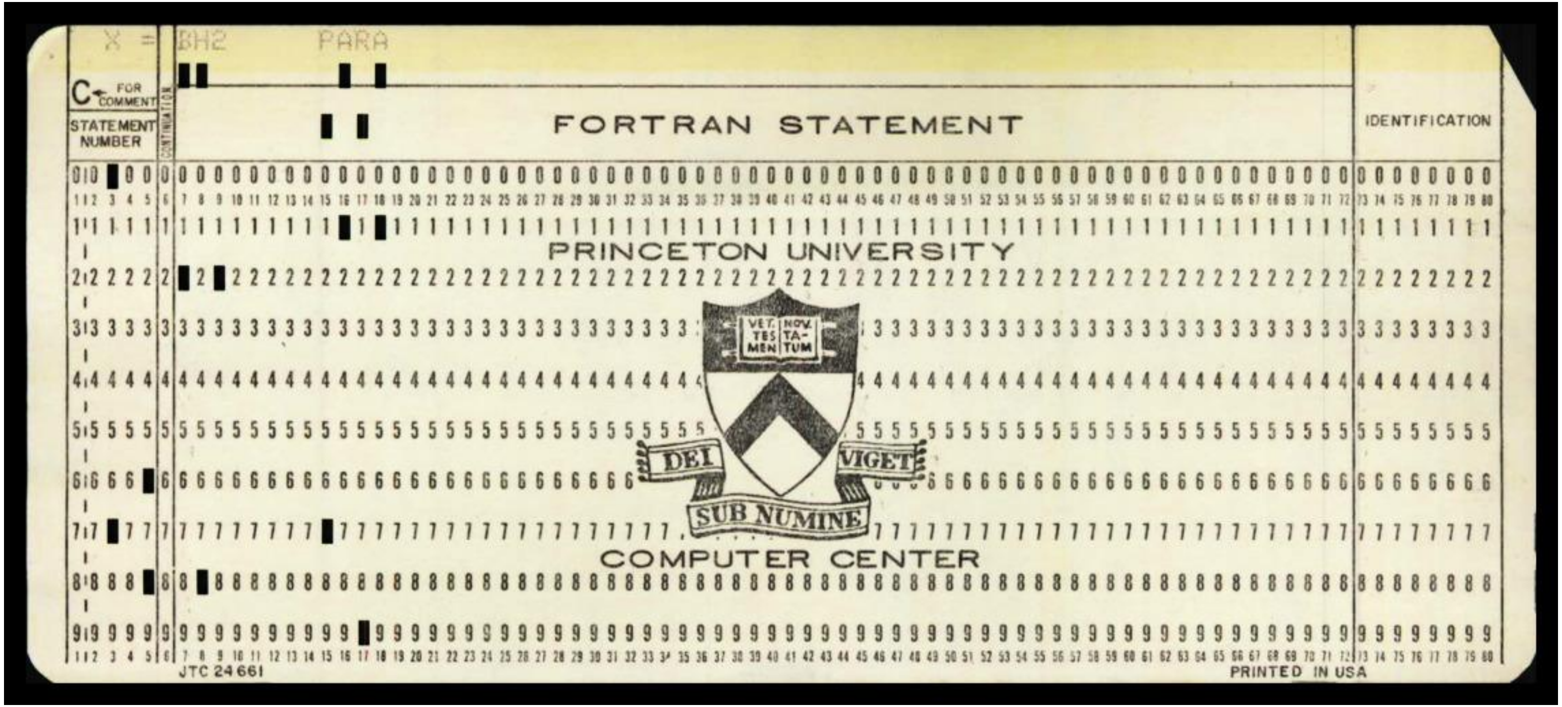
Data started with the punch card

introduced by Herman Hollereith in 1890

7.34 inches wide by 3.25 inches high and approximately .07 inches thick, a punch card was a piece of paper or cardstock containing holes in specific locations that corresponded to specific meanings.

TACC

Data Science 101



Data Science 101

Scale

Coding up data and programs through a series of holes in a piece of paper can only ***scale*** so far

it was revolutionary for its day because the existence of semi autotic data tallying allowed for faster and more accurate computation.

Data Science 101

Speed

the second prong of the big data revolution involves how fast we can move around and compute with data.

Data Science 101

Ubiquity

Definition: the fact of appearing everywhere or of being very common.

Data Science 101

Putting the science in data science

it's short answer to what you can do with the billions upon billions upon of data points being collected

Data Science 101

The data is there.

It exists

There's something valuable in it.

But what does it mean? What's going on? What can you learn? How can you use it to make better science?

Data analysis is all about asking these types of questions.

Data Science 101

Here's the catch

You have to understand how the data came to be and what the goals of the process are in order to do good analytic work.

Data Science 101

Experimentation has been around for a long time.

People have been testing out new ideas for far longer than data science has been a thing.

Experimentation is at the heart of a lot of modern data work.

Data Science 101

Machine Learning

Data scientists define machine learning as the process of using machines to better understand a process or system, and recreate, replicate or augment that system.

Data Science 101

Machine Learning, Supervised Learning

Supervised learning is probably the most well known of the branches of data science.

All about predicting something you've seen before.

You try to analyze what the outcome of the process was in the past and build a system that tries to draw out what matters and build predictions for the next time it happens.

Data Science 101

Machine Learning, Unsupervised Learning

You can do a lot of machine learning work without an observed outcome or target.

Unsupervised learning is less concerned about making predictions than understanding and identifying relationships or associations that might exist within the data.

Data Science 101

Machine Learning, Unsupervised Learning

The K Means algorithm.

This technique, calculates the distance between different points of data and groups similar data together.

This The “suggested new friends” feature on Facebook

TACC

Data Science 101

Machine Learning, Reinforcement Learning

Reinforcement learning requires an active feedback loop.

Reinforcement learning requires a dynamic dataset that interacts with the real world.

Data Science 101

Artificial Intelligence

Artificial Intelligence wants some kind of human interaction and is intended to be somewhat human or “intelligent” in the way it carries out those interactions. Therefore, that interaction becomes a fundamental part of the product a person seeks to build. Data science is more about insight and building systems. It places less emphasis on human interaction and more on providing intelligence, recommendations, or insights.

TACC

Data Science 101

In a nutshell

Data is important.

We need to understand what the data is

What the data means

To find the underlying story

Questions? Comments?