Data Science for Business

Business Problems and Data Science Solutions

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 2.1

Overview

- **Data Science Tasks**
- Data Science Models
- Data Science Process

Data Science Tasks

1) Classification: For each individual in a population, identify a (small) set of classes to which that individual belongs.

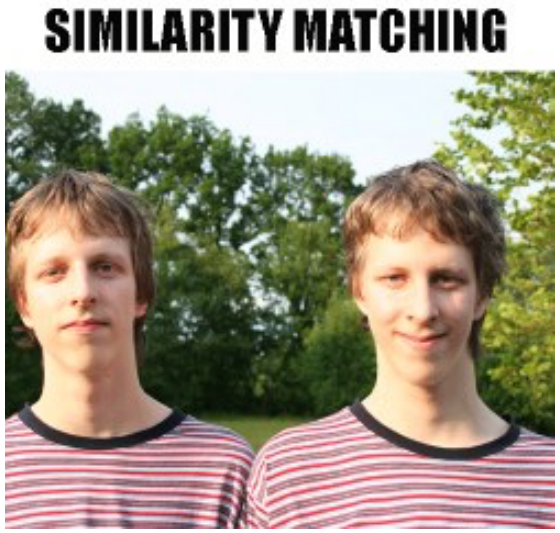
- Class probability estimation (or scoring) – What is the probability/score that the individual belongs to each class?

Data Science Tasks

2) Regression ("value estimation") is used to estimate or predict, for each individual, the numerical value of some variable.

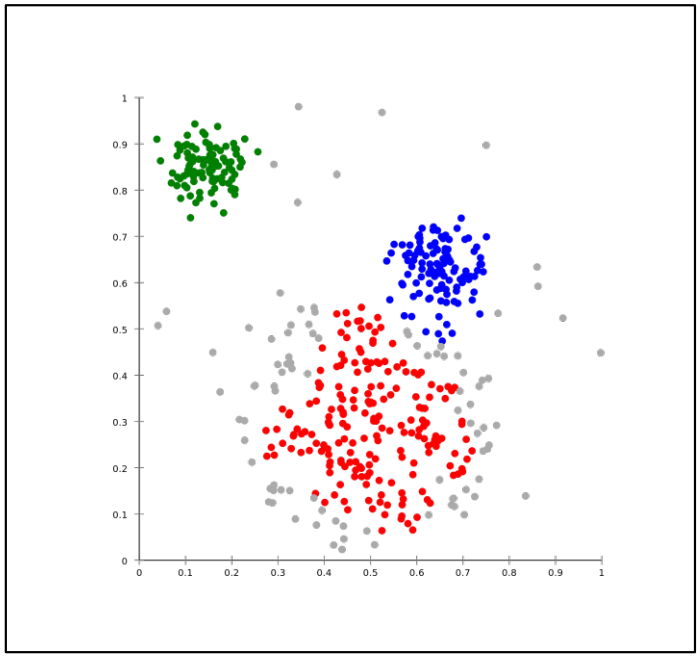
Data Science Tasks

3) Similarity matching is used to identify similar individuals based on data known about them. Similarity matching can be used directly to find similar entities.



Data Science Tasks

4) Clustering is used to group individuals in a population together by their similarity, but not driven by any specific purpose (example or variable).



Data Science Tasks

5) Co-occurrence Grouping (also known as frequent itemset mining, association rule discovery, or market-basket analysis) is used to find associations between entities based on the transactions they are involved in.

Co-occurrences and Associations

- Complexity control:
  - Support of association
    - Let's say that we require rules to apply to at least 0.01% of all transactions
  - Confidence or strength of the rule
    - Let's say that we require that 5% or more of the time, a buyer of A also buys B

- Measuring surprise: Lift and Leverage

$$\text{Lift}(A, B) = \frac{p(A, B)}{p(A)p(B)}$$

$$\text{Leverage}(A, B) = p(A, B) - p(A)p(B)$$

Lift and Leverage

- **Lift**:
  - The **Lift** of the co-occurrence of *A* and *B* is the probability that we actually see the two together, compared to the probability that we would see the two together if they were unrelated to (independent of) each other.
  - A **Lift** greater than one is the factor by which seeing *A* "boosts" the likelihood of seeing B as well.

- **Leverage:**
  - The **Leverage** tells how much more likely than chance a discovered association is. An alternative is to look at the difference of these quantities rather than their ratio.

Example: Beer and Lottery Tickets

- We operate a small convenience store where people buy groceries, liquor, lottery tickets, etc. We estimate that:
- 30% of all transactions involve beer,
- 40% of all transactions involve lottery tickets,
- and 20% of the transactions include both beer and lottery tickets.

Example: Beer and Lottery Tickets

- If the two products are unrelated:

$$p(beer) \times p(lottery\ tickets) = 0.3\ x\ 0.4\ = 0.12$$

- Otherwise:

$$Lift(beer, lottery\ tickets) = \frac{0.2}{0.12} \approx 1.67$$

$$Leverage(beer, lottery\ tickets) = 0.2 - 0.12 = 0.08$$

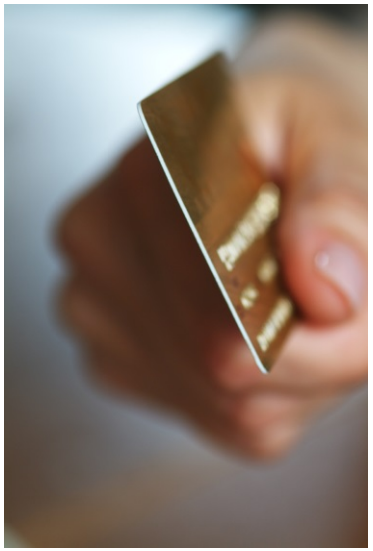
$$Support(beer, lottery\ tickets) = 20\%$$

$$Strength(beer, lottery\ tickets) = p(lottery\ tickets|beer) = \frac{0.2}{0.3} = 67\%$$

Data Science Tasks

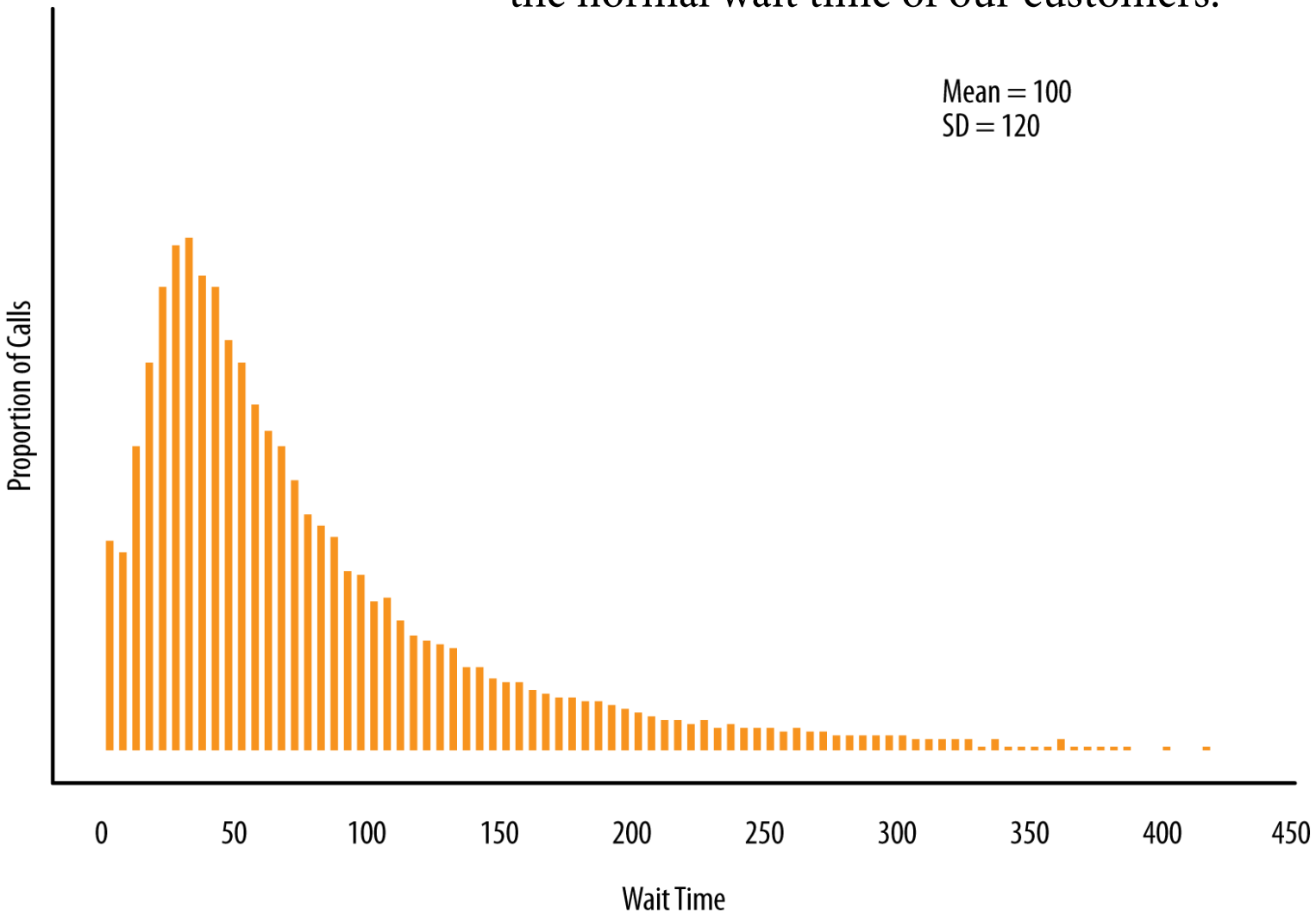
6) Profiling (also known as behavior description) is used to characterize the typical behavior of an individual, group, or population.

- A sample profiling question is: *What is the typical cellphone usage of this customer segment?*



Profiling

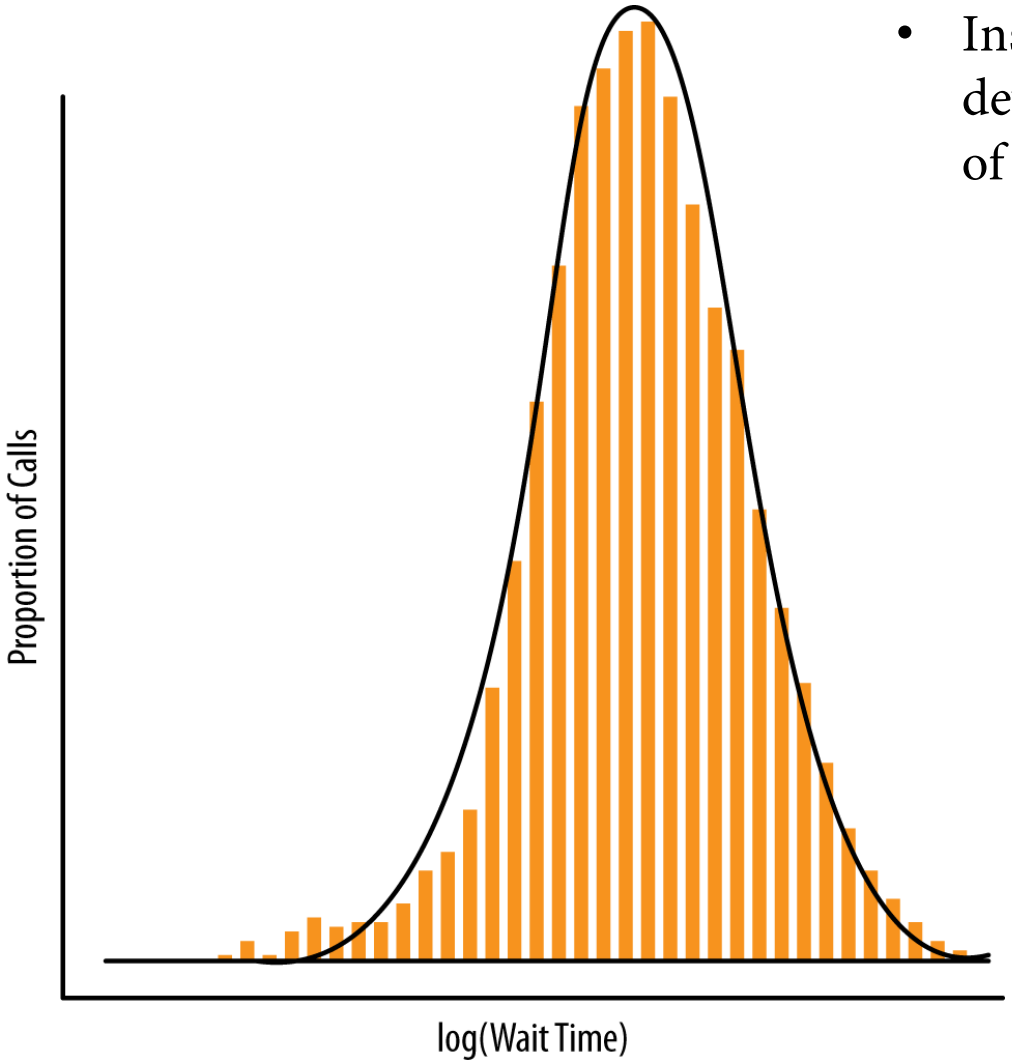
- The mean (100) does not seem to satisfy our desire to profile how long our customers normally wait; it seems too large
- Technically, the long "tail" of the distribution skews the mean upward, so it does not represent faithfully where most of the data really lie. It does not represent faithfully the normal wait time of our customers.



A distribution of wait times for callers into a bank's call center.

Profiling

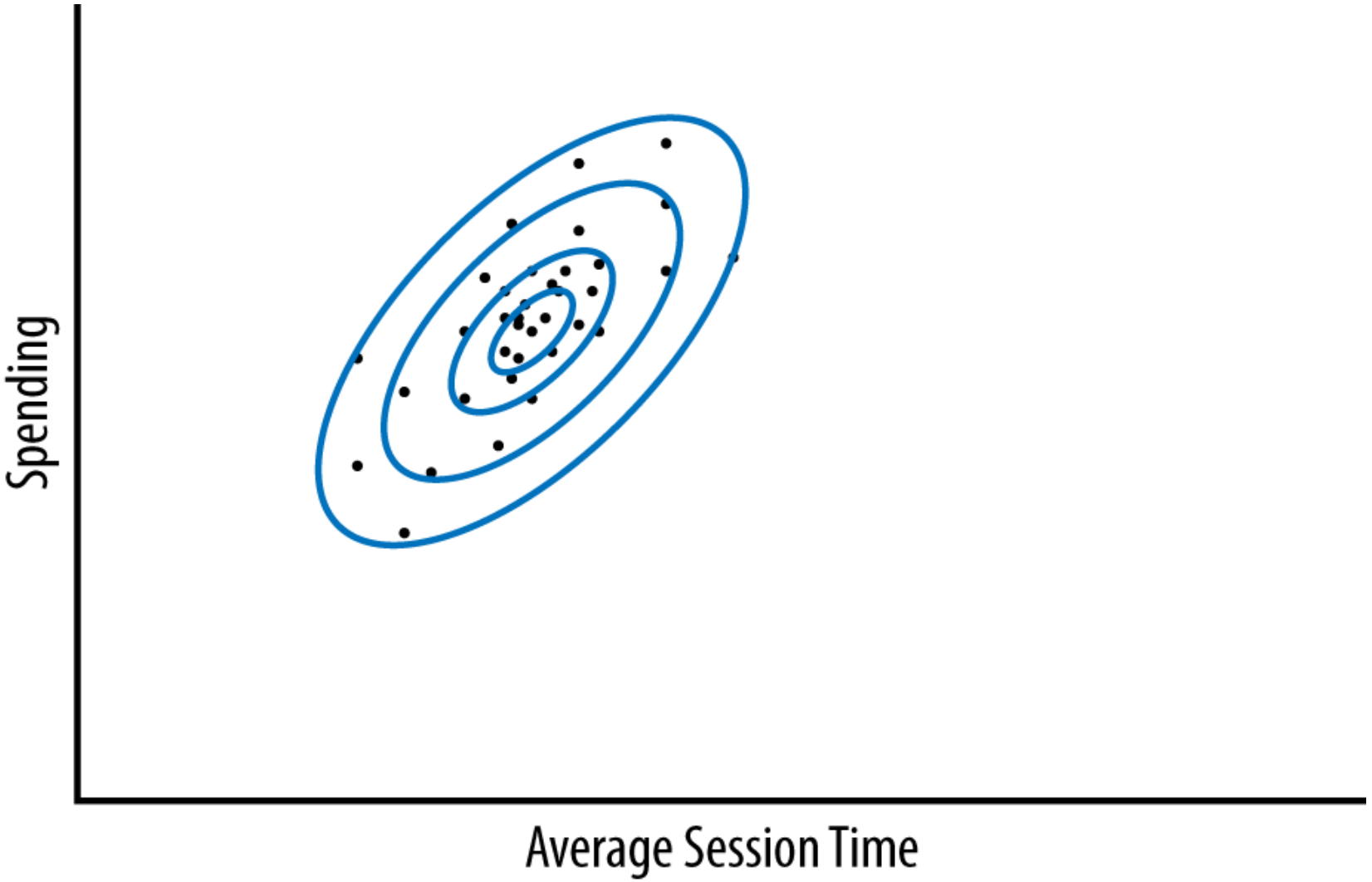
- A common trick for dealing with data that are skewed in this way is to take the logarithm (log) of the wait times
- Just after the simple transformation, the wait times look very much like the classic bell curve (an actual Gaussian distribution).
- Instead, report the mean and standard deviation as summary statistics of the profile of (log) wait times



The distribution of wait times for callers into a bank's call center after a quick redefinition of the data.

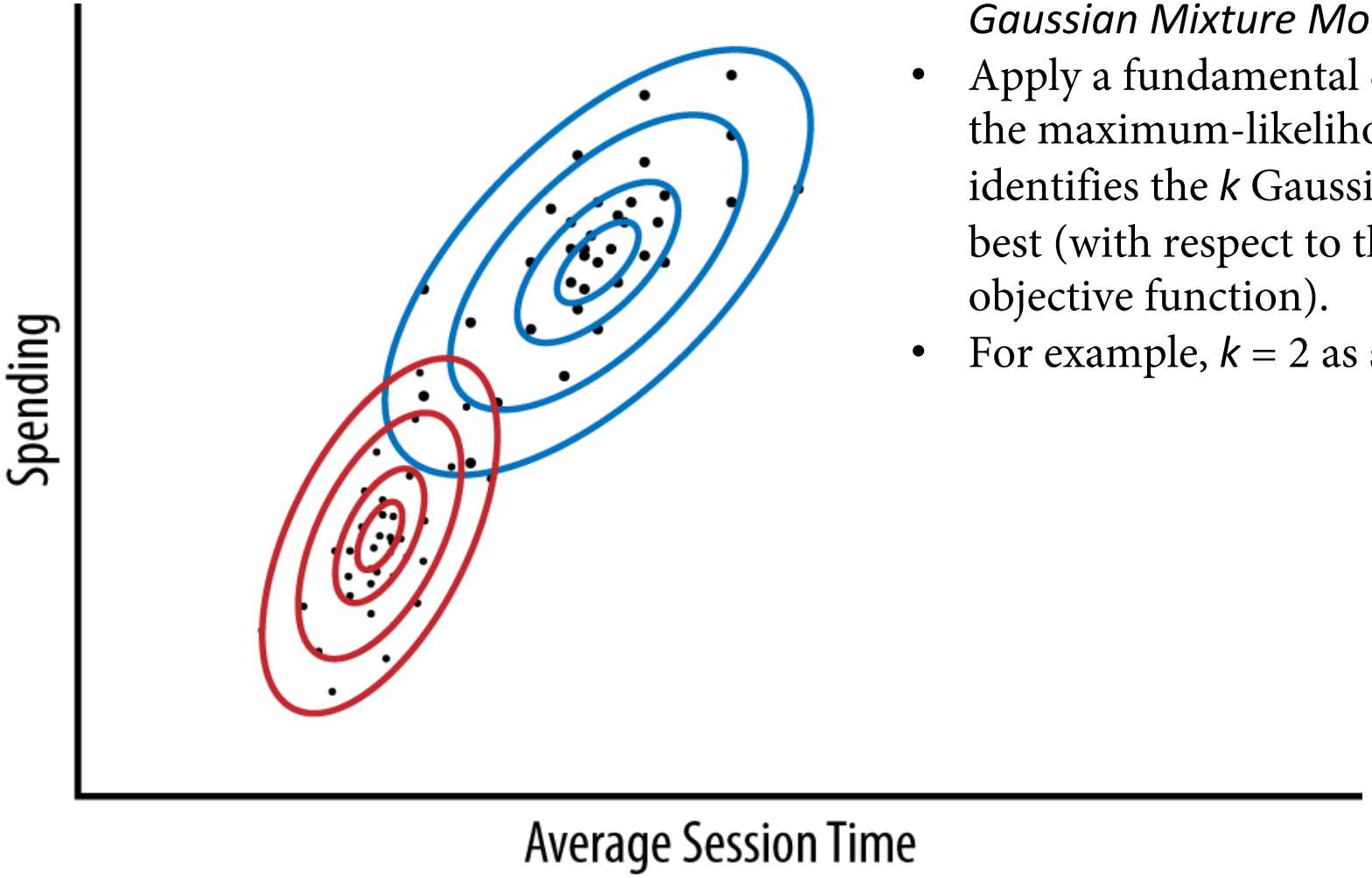
Profiling

Profiling can essentially involve clustering, if there are subgroups of the population with different behaviors



A profile of our customers with respect to their spending and the time they spend on our web site, represented as a two-dimensional Gaussian fit to the data.

Profiling



- Assume that there are *k* groups of customers, each of whose behavior is normally distributed. We can fit a model with multiple Gaussians, called a *Gaussian Mixture Model* (GMM).
- Apply a fundamental concept, finding the maximum-likelihood parameters identifies the *k* Gaussians that fit the data best (with respect to this particular objective function).
- For example, $k = 2$ as seen in this figure.

A profile of our customers with respect to their spending and the time they spend on our web site, represented as a Gaussian Mixture Model (GMM), with 2 two- dimensional Gaussians fit to the data. The GMM provides a "soft" clustering of our customers along two these two dimensions.

Data Science Tasks

7) Link Prediction is used to predict connections between data items, usually by suggesting that a link should exist, and possibly also estimating the strength of the link.

- *Since you and Karen have 10 friends in common, maybe you'd like to be Karen's friend?*

Data Science Tasks

8) Data Reduction is used to replace a large set of data with a smaller set of data that contains much of the important information in the larger set.

- Data Cube Aggregation

Aggregation operations are applied to the data in the construction of a data cube.

- Dimensionality Reduction

In dimensionality reduction redundant attributes are detected and removed which reduce the data set size.

- Numerosity Reduction

In numerosity reduction where the data are replaced or estimated by alternative.

- Discretization and concept hierarchy generation

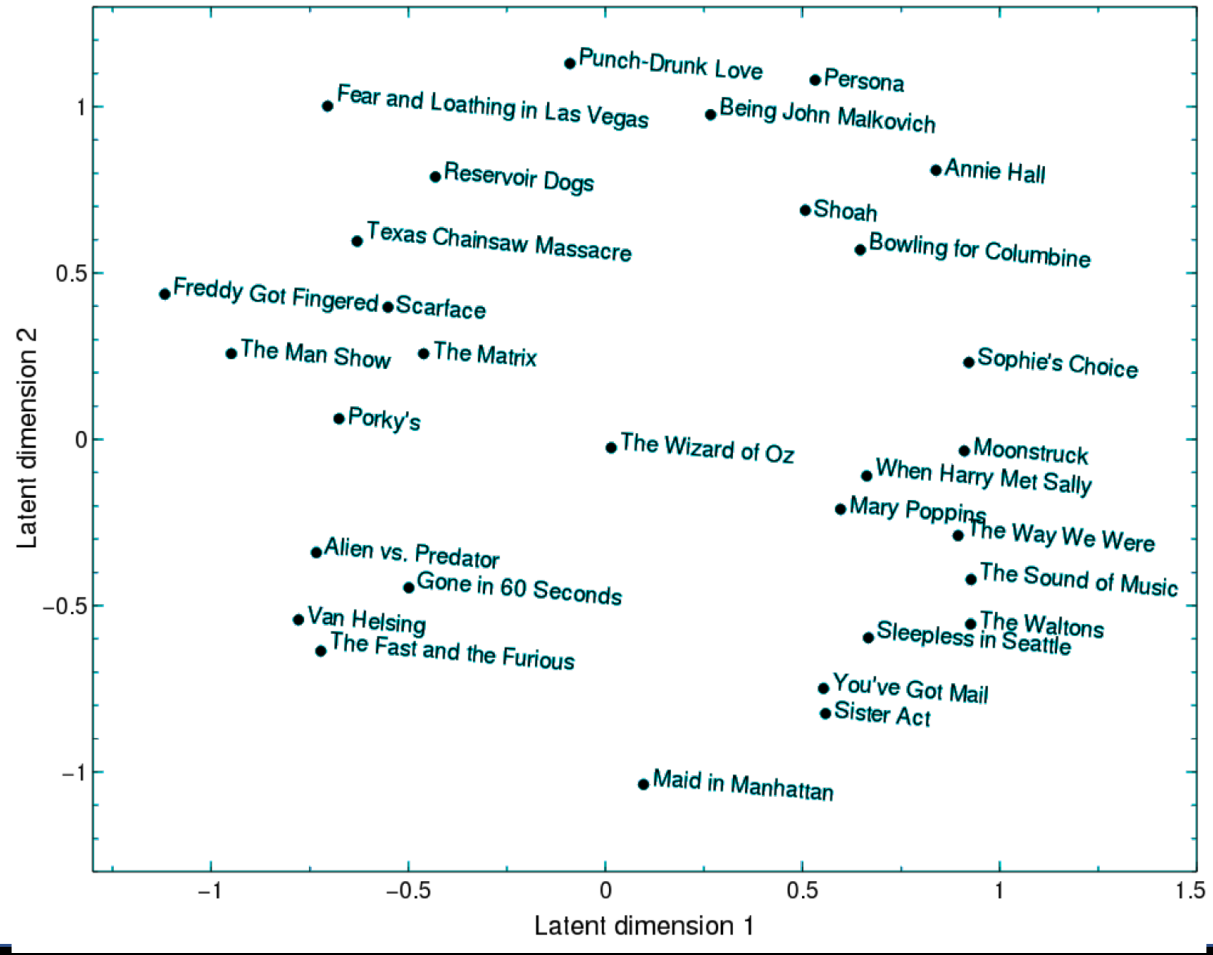
Where raw data values for attributes are replaced by ranges or higher conceptual levels.

- Data Compression

Encoding mechanisms are used to reduce the data set size.

Data Science Tasks

9) Latent Information is used to discover or refer to any (hidden) information that is not plainly observable from raw data.



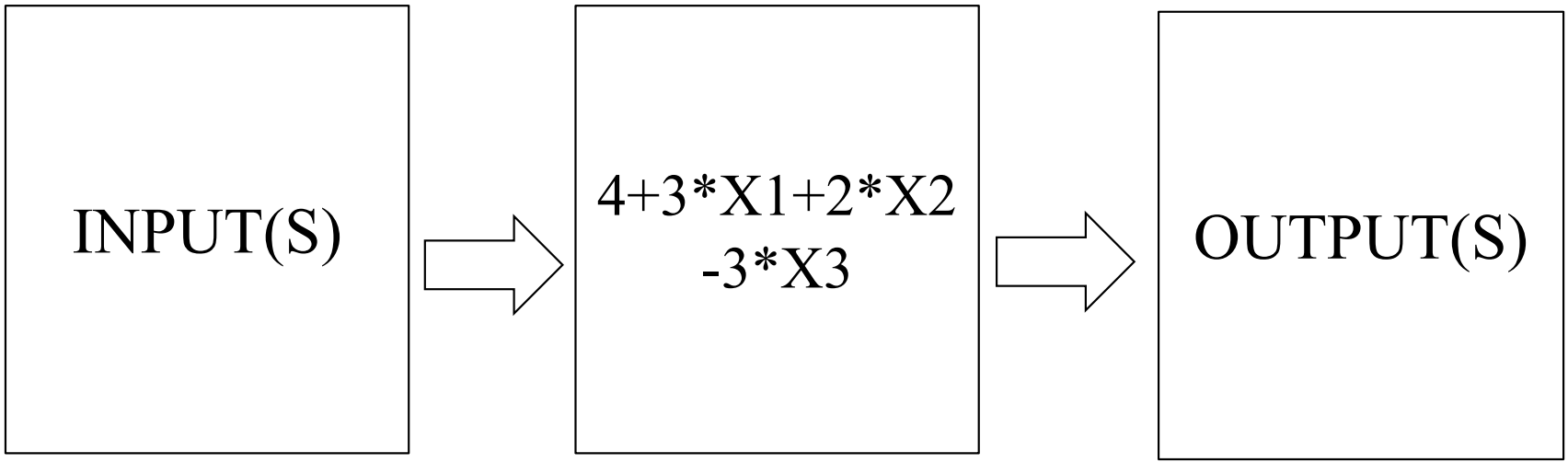
Latent Information and Movie Recommendation

Overview

- Data Science Tasks
- **Data Science Models**
- Data Science Process

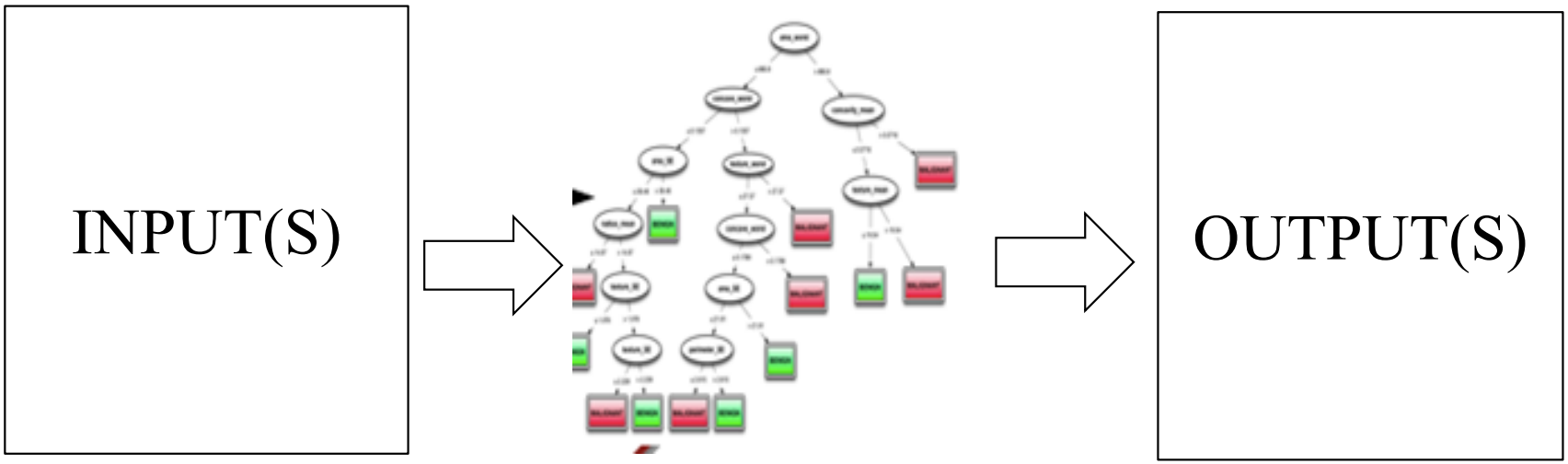
Data Science Model

Example of a simple model.



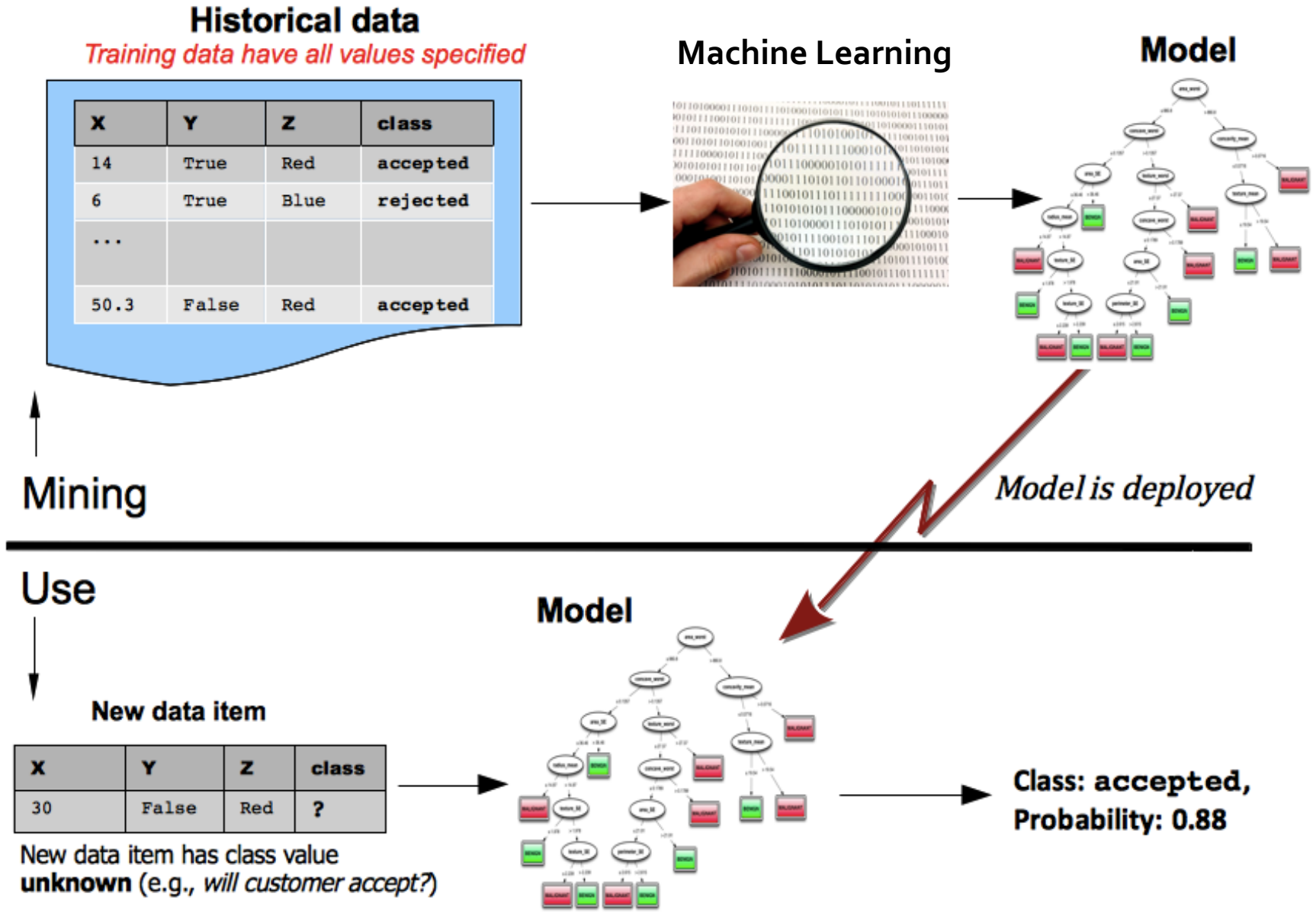
Data Science Model

Example of a simple model.



Model image from "Data Science for Business," Provost and Fawcett, 2013

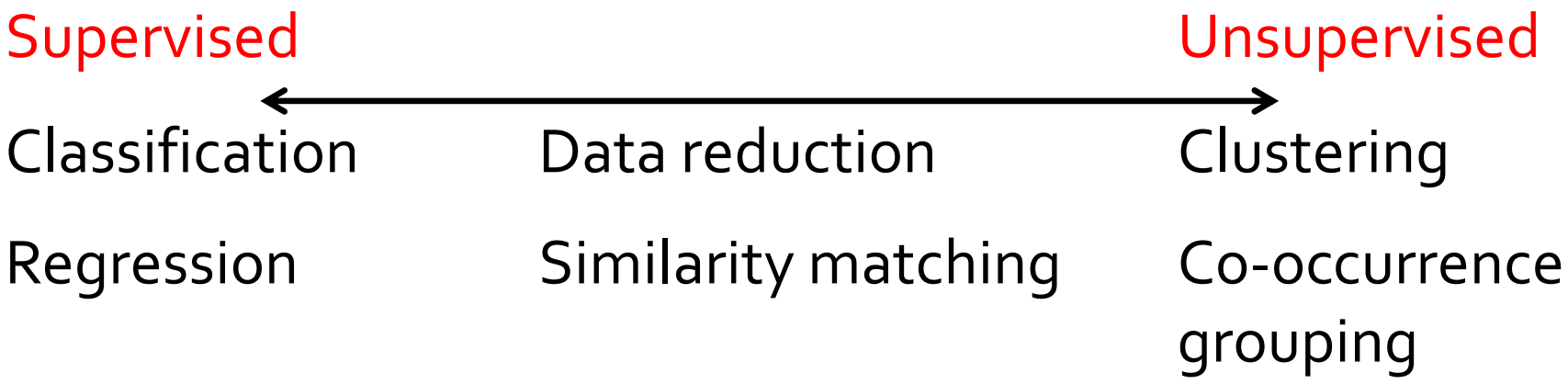
Data Science Model



Model image from "Data Science for Business," Provost and Fawcett, 2013

Supervised vs. Unsupervised Learning

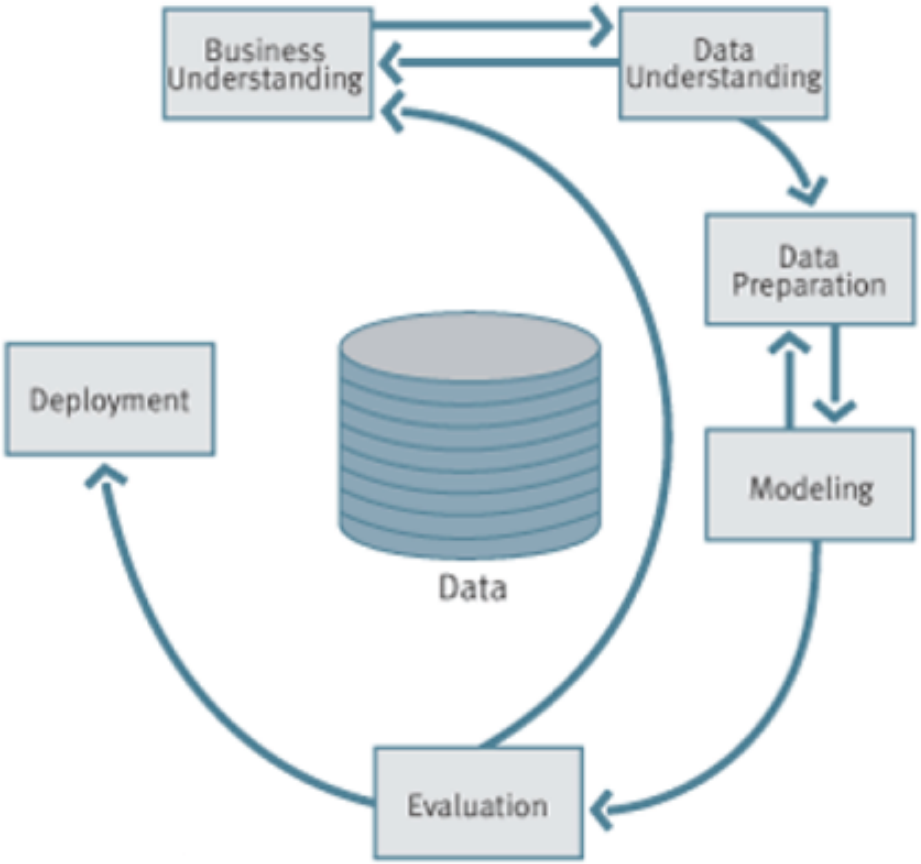
- Key Questions:
  - Is there a specific target variable?
  - (Are data on this target variable available?)



Overview

- Data Science Tasks
- Data Science Models
- **Data Science Process**

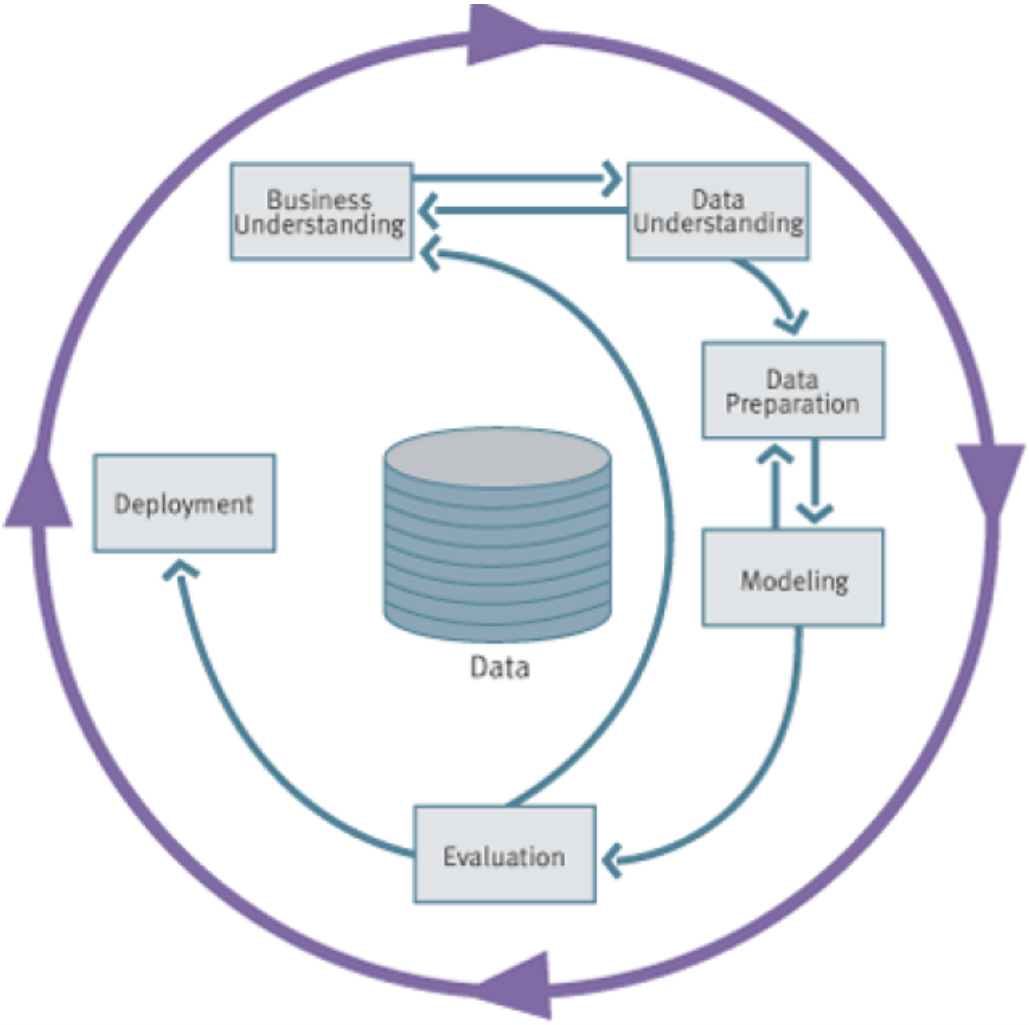
Data Science Process



* Cross-Industry Standard Process for Data Mining (Shearer, 2000)

Image from "Data Science for Business", Provost and Fawcett, 2013

Data Science Process – is iterative



Iterative Process

Very few DS projects became successful using Waterfall model

* Cross-Industry Standard Process for Data Mining (Shearer, 2000)

Image from "Data Science for Business", Provost and Fawcett, 2013

Managerial Implications

- Not a software development project
- Outcomes are uncertain
- Iterative process – prepare sufficient resources
- Intensive data preparation stage
- Different skill set than of programmers

Related Analytic Techniques and Technologies

- Statistics
- Machine learning
- Database querying
- Data Warehousing
- On-line Analytical Processing (OLAP)

Data Science and Machine Learning
Data Science Pipeline

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 2.2

Overview

- **How to begin DS projects ?**
- Basic Data Science Pipeline
  - Input and Ingestion
  - Calculation and Analysis
    - Supervised v Unsupervised Learning
  - Evaluation & Performance
  - Visualization/Output

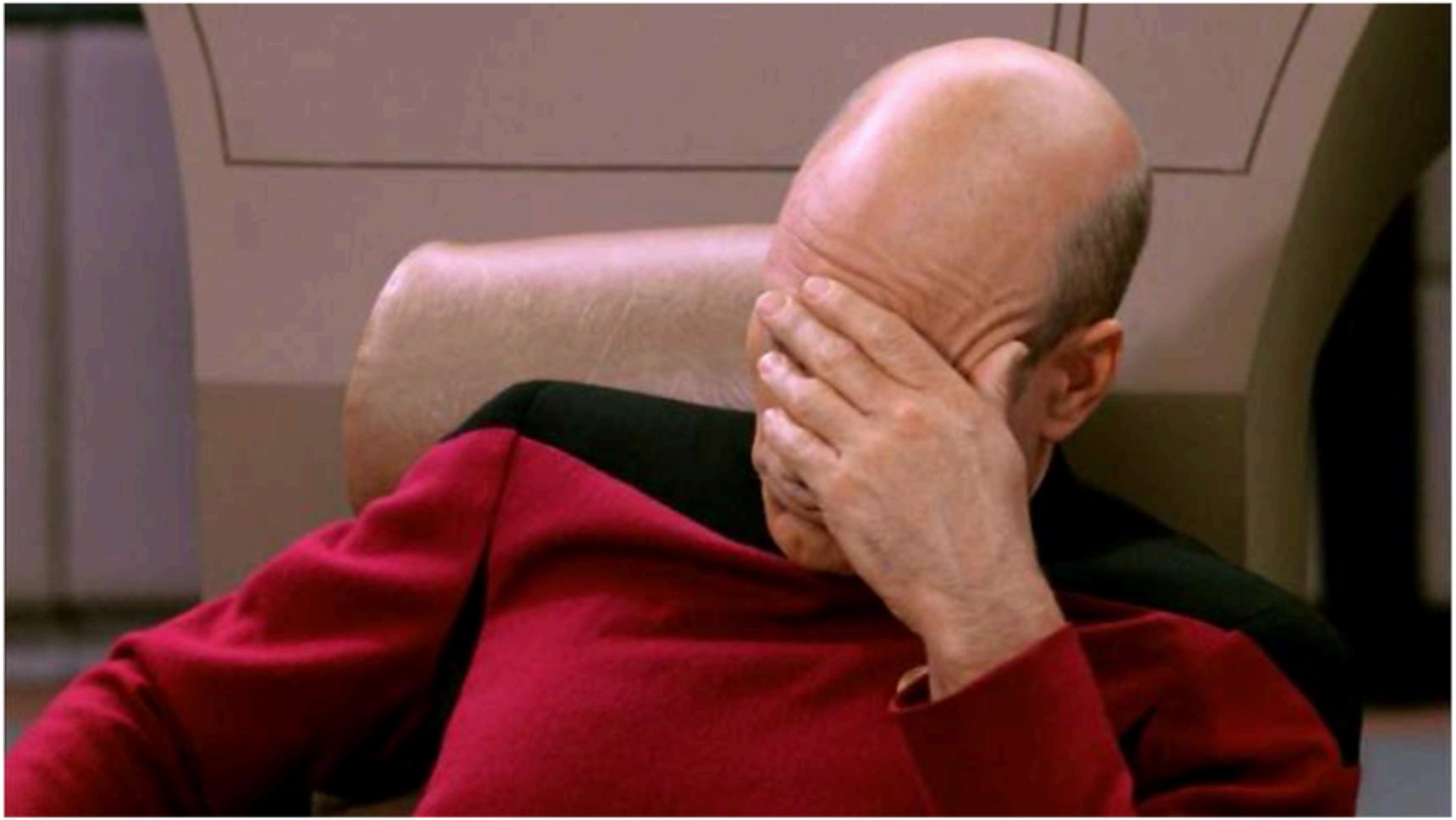
How to begin DS projects?

EXPECTATION

How to begin DS projects?

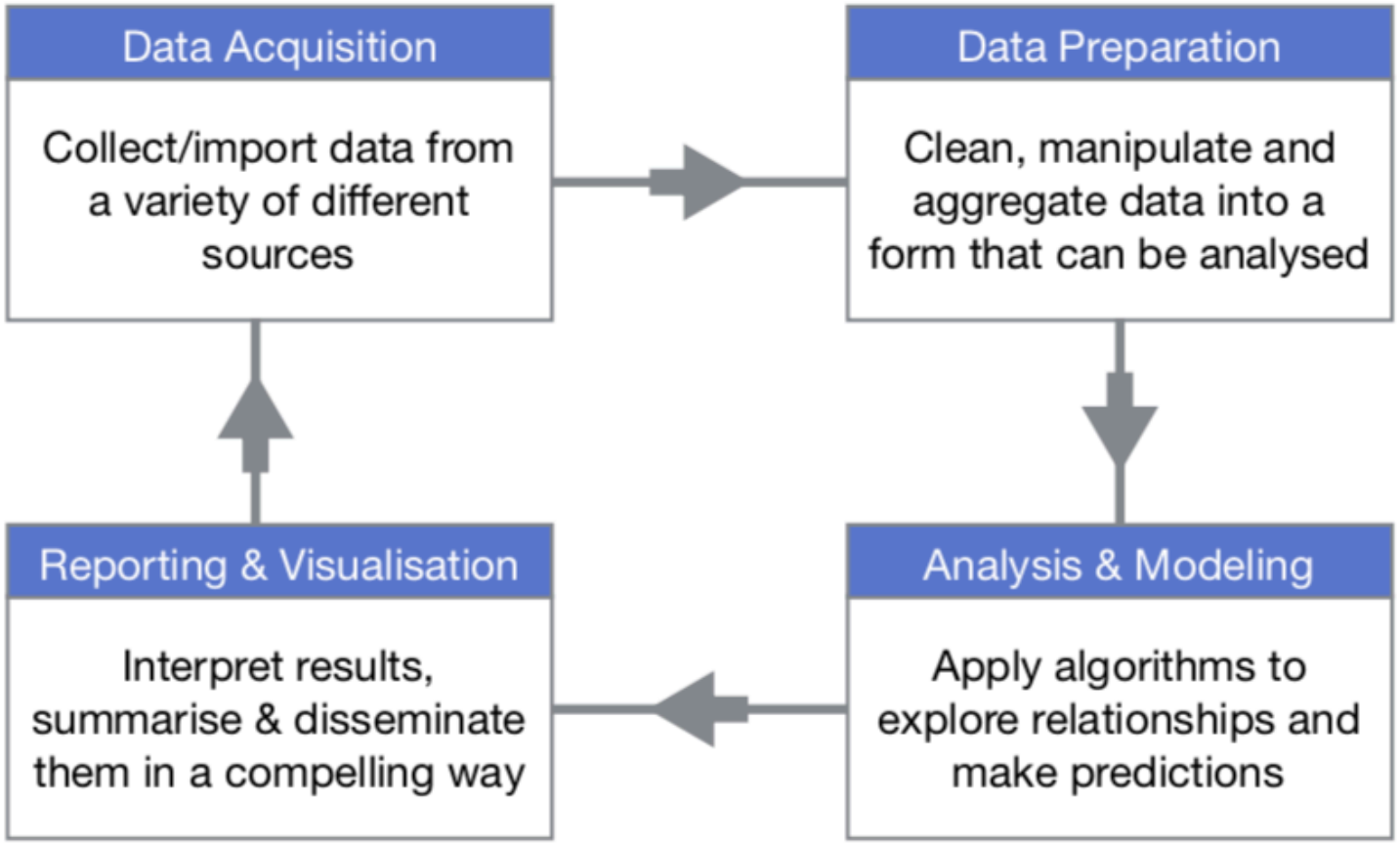
ACTUAL

How to begin DS projects?

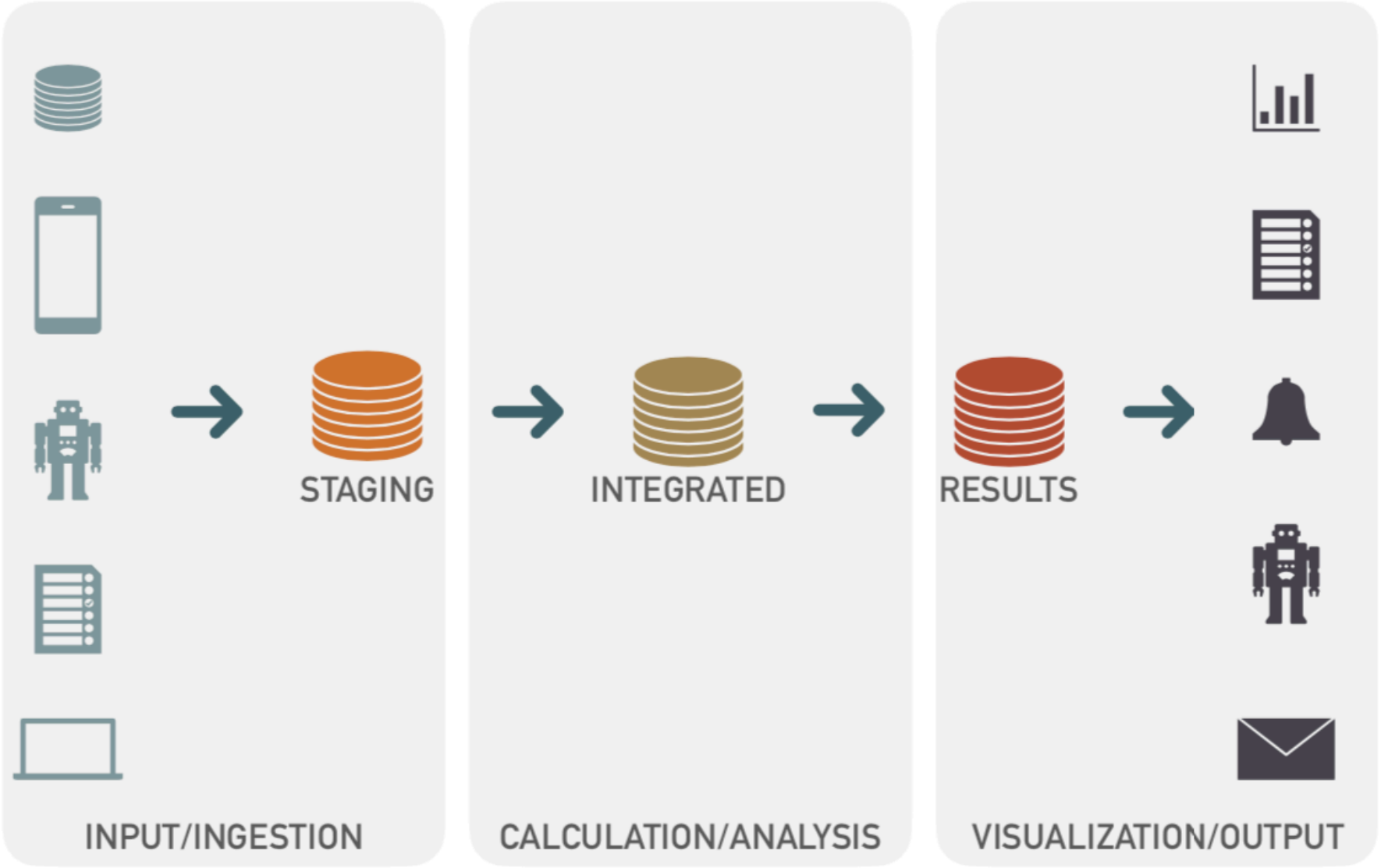


Basic Data Science Pipeline

- Data analysis projects typically involve iterating through a pipeline of steps. A simple data science pipeline consists of...



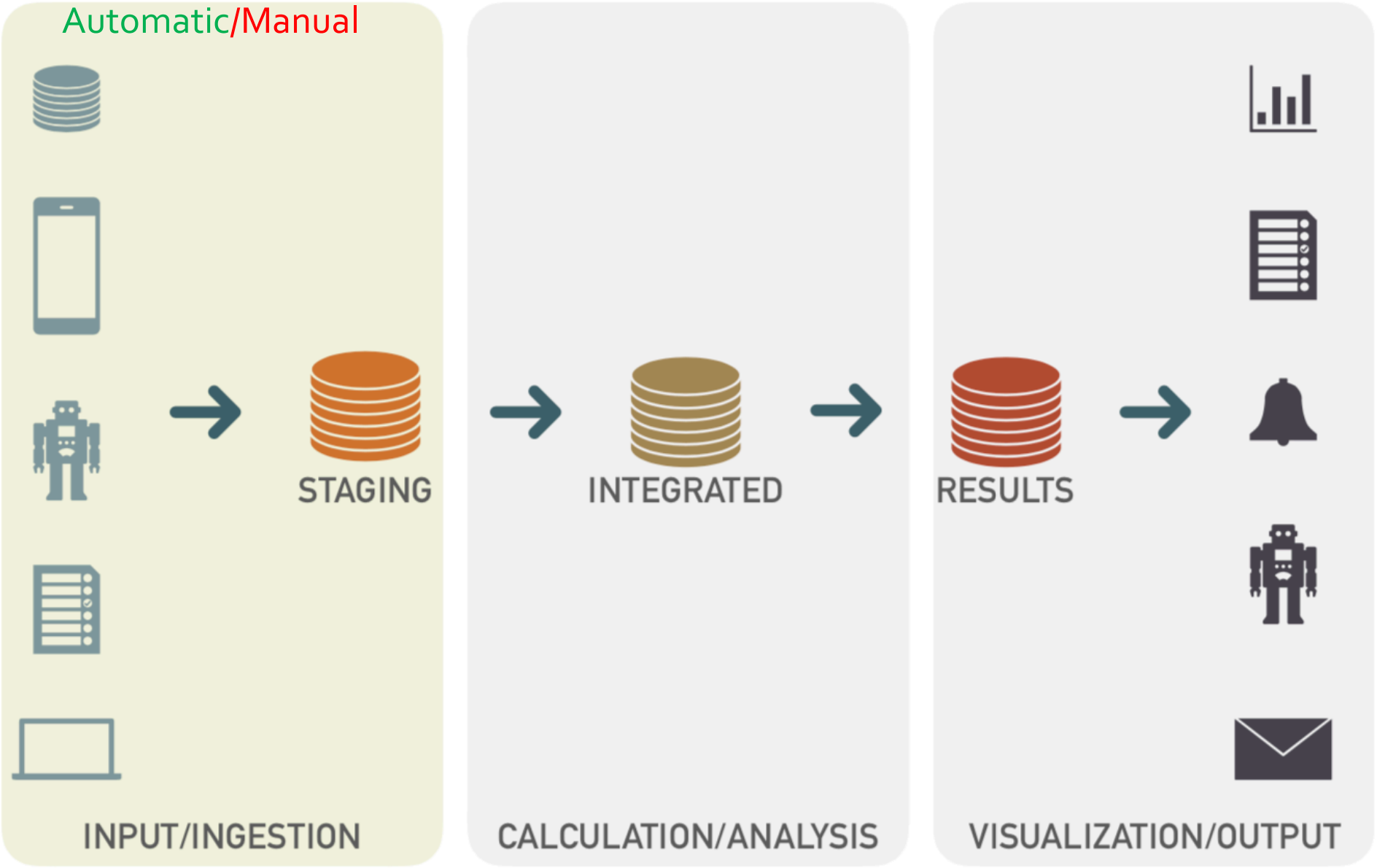
A Data Science Project



Overview

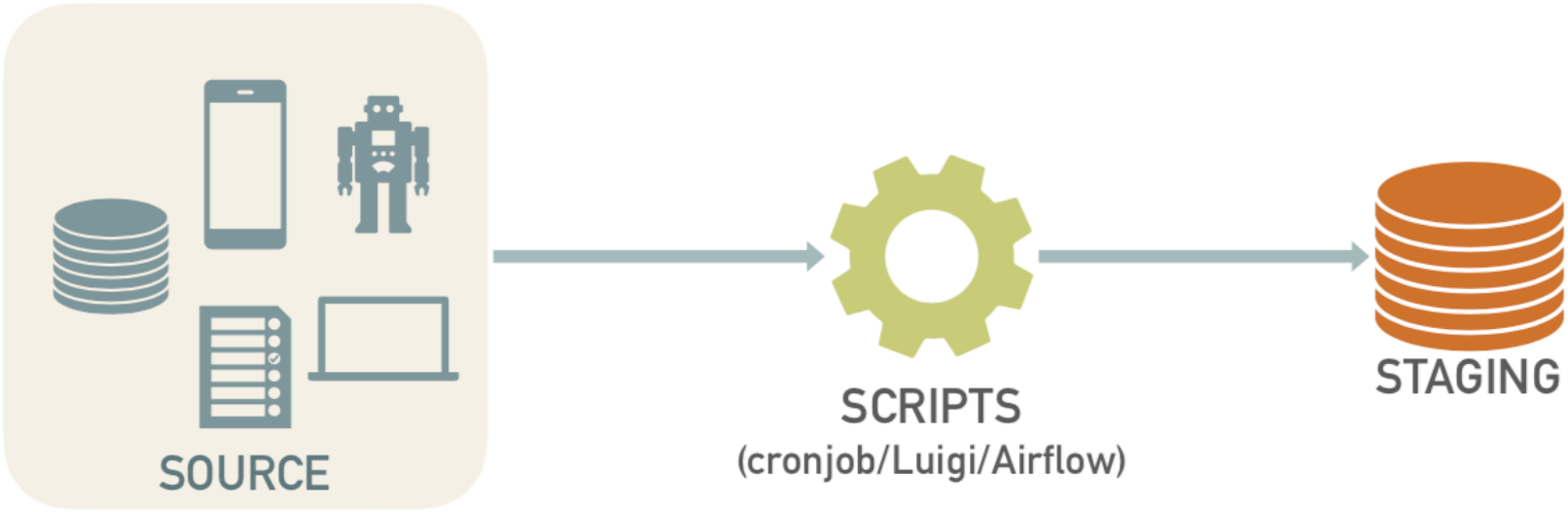
- How to begin DS projects ?
- **Basic Data Science Pipeline**
  - **Input and Ingestion**
  - Calculation and Analysis
    - Supervised v Unsupervised Learning
  - Evaluation & Performance
  - Visualization/Output

Input/Ingestion



Automatic Ingestion

- Connect to database directly
- API
- Web scraping
- Chatbot



Automatic Ingestion: **Database**

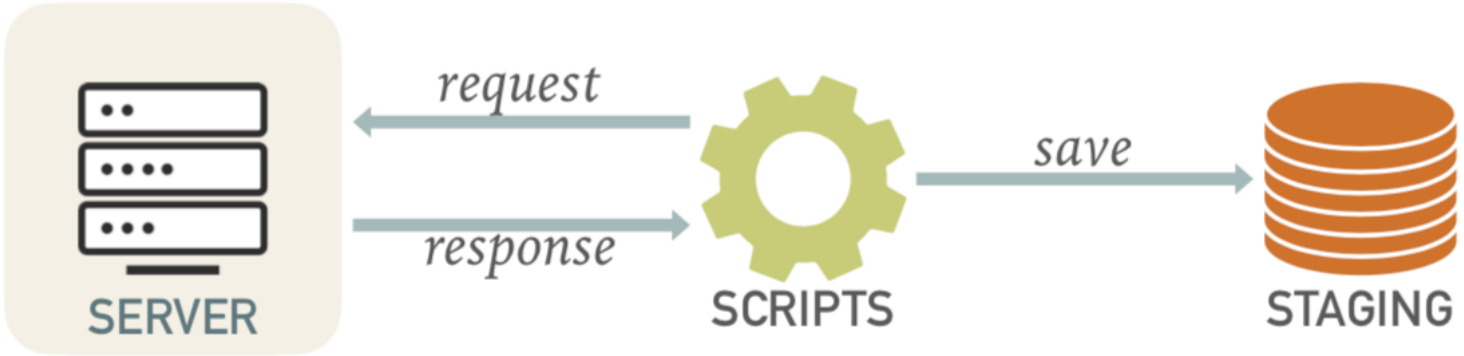

```
# Initial
from coralinedb import MySQLDB
db = MySQLDB(HOST, USERNAME, PASSWORD)

# Load data
df = db.load_table(DB_NAME, TABLE_NAME)
```

df (pandas)

C1	C2	C3
3	A	B
22	C	D
2	A	A
45	B	E

coralinedb in Github: https://github.com/jiranun/coralinedb

Automatic Ingestion: **API**



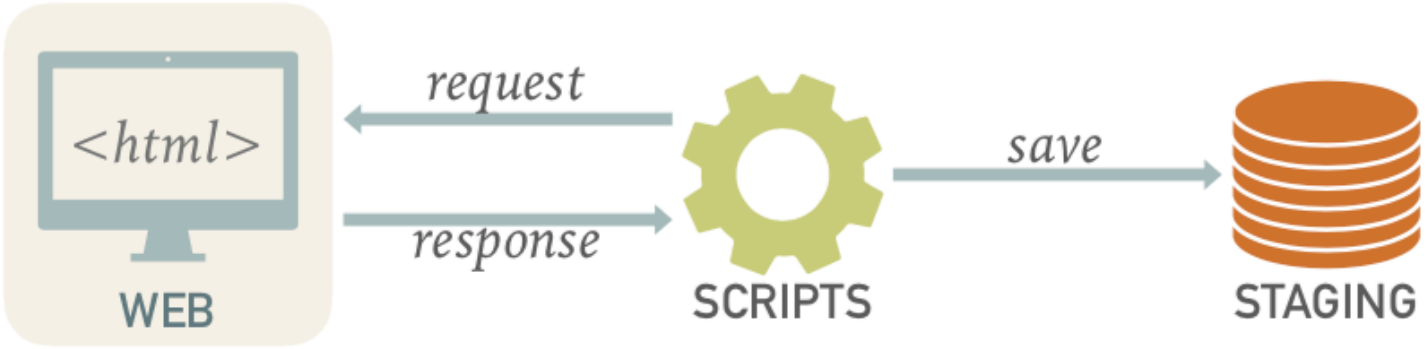

```
import requests
url = "http://.."
params = {
    'param1': 'aaa',
    'param2': 'bbb'
}
response = requests.get(url, params)
```

Responses

- XML
- JSON
- …

Some sites provide packages for API e.g. Twitter, Facebook, Google, etc.

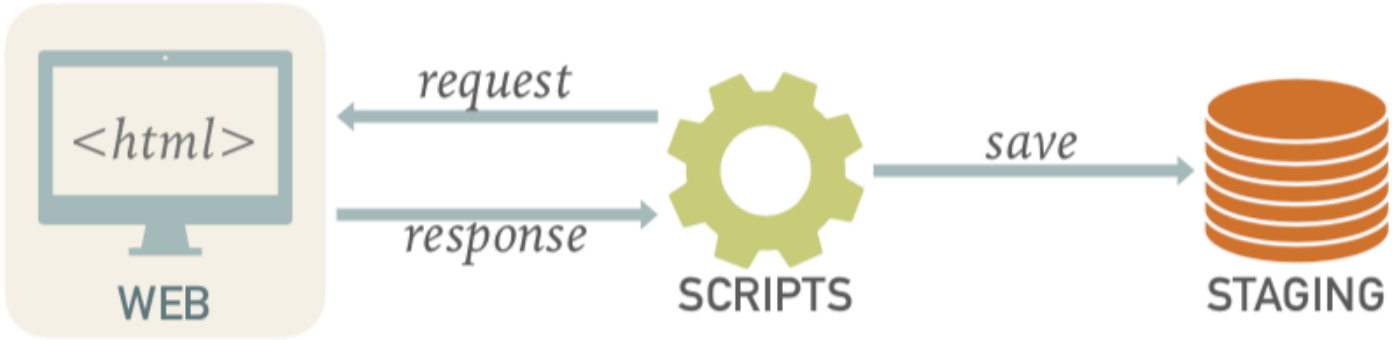
Automatic Ingestion: **Web Scraping (1)**




```
from bs4 import BeautifulSoup
import requests

resp  = requests.get("https://..")
soup = BeautifulSoup(resp.text)

table_list = soup.find_all("table")
first_img = soup.find("img").get("src")
```

Basic HTML knowledge is required

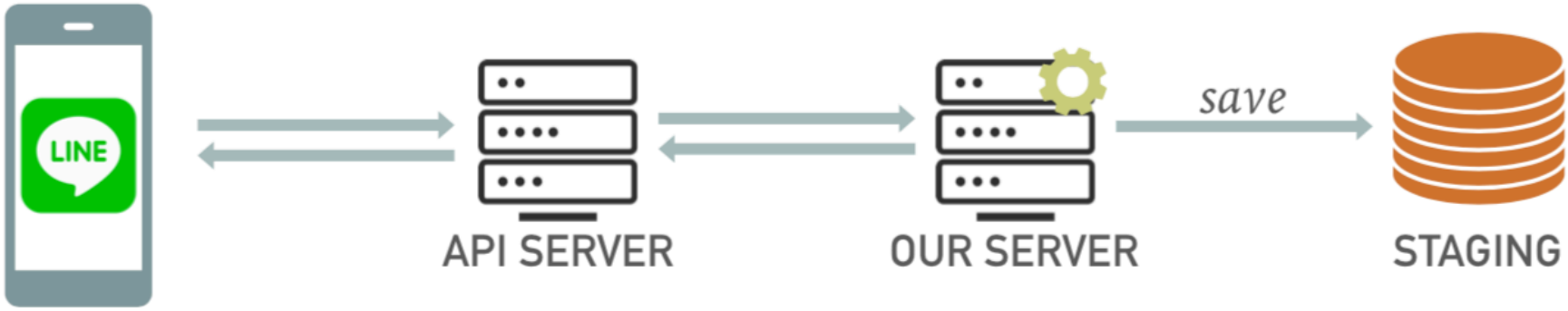
Automatic Ingestion: **Web Scraping (2)**




```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys

driver = webdriver.Firefox()
driver.get("http://www.python.org")
html = driver.page_source  # Get HTML

elem = driver.find_element_by_name("q")
elem.send_keys("pycon thailand")
elem.send_keys(Keys.RETURN)
```

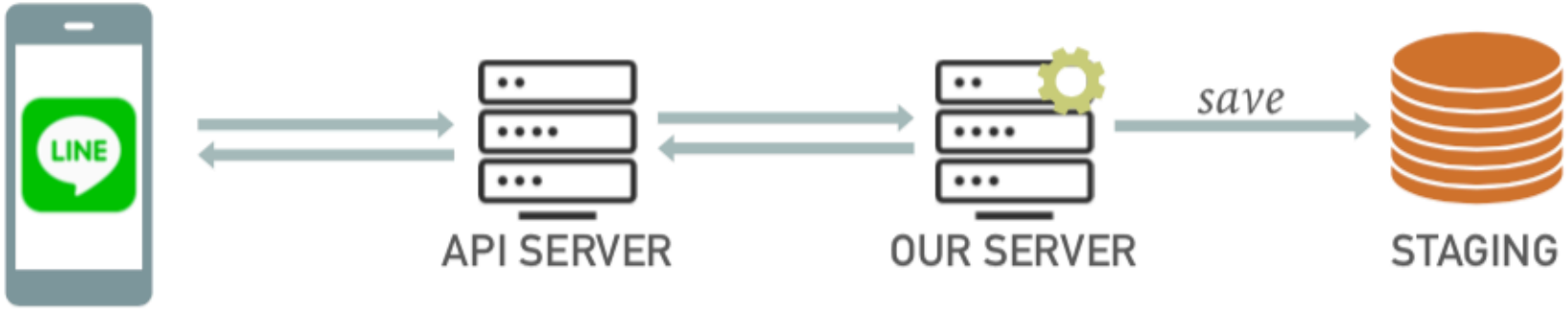
Basic HTML knowledge is required

Automatic Ingestion: **Chatbot (1)**



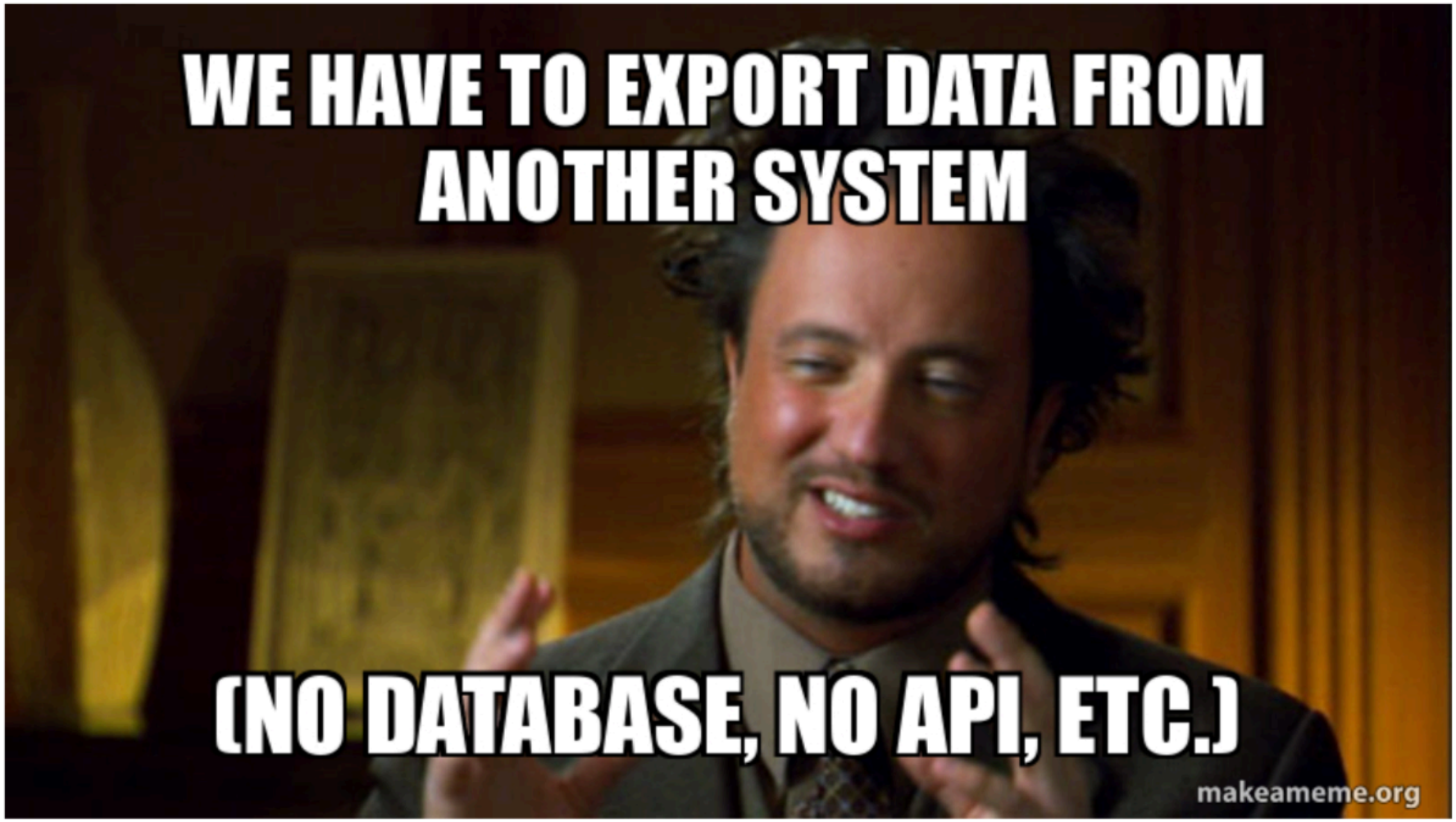

```
# Initial
from flask import Flask, request
from linebot import (LineBotApi, WebhookParser)

app = Flask(__name__)
line_bot_api = LineBotApi(LINE_CHANNEL_ACCESS_TOKEN)
parser = WebhookParser(LINE_CHANNEL_SECRET)
```

Automatic Ingestion: **Chatbot (2)**




```
@app.route("/", methods=['POST'])
def callback():
    signature = request.headers['X-Line-Signature']
    events = parser.parse(request.get_data(as_text=True),
                          signature)
    for event in events:
        line_bot_api.reply_message(
            event.reply_token,
            TextSendMessage(text=event.message.text)
        )
    return 'OK'
```

Manual Ingestion: **Why?**



Manual Ingestion: **AWS solution**

Manual Ingestion: **Create Web for Staging**

Example:

Select date: 2019/06/06

Transaction: Choose File No file chosen

Stock: Choose File No file chosen

Comment:

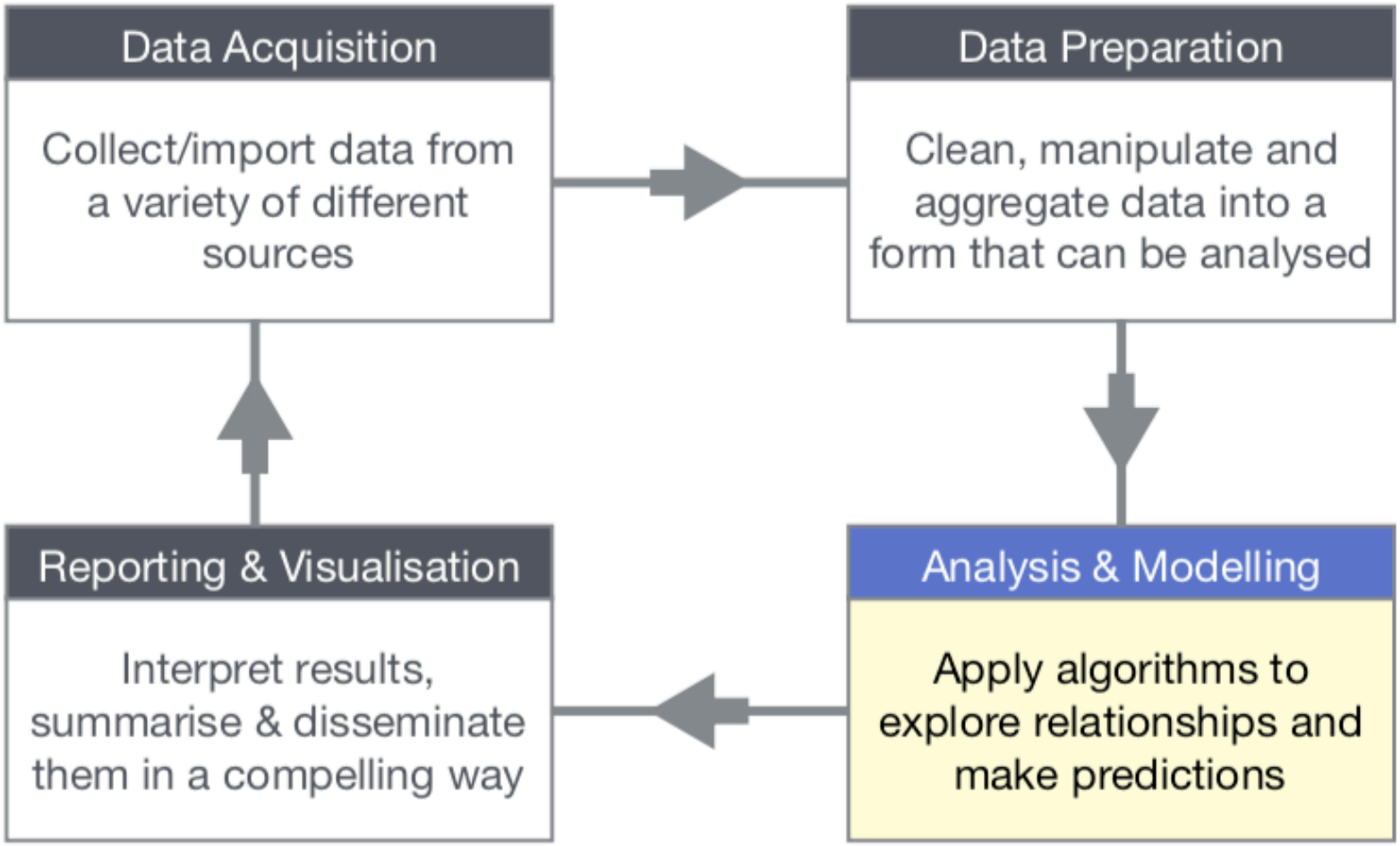
Submit

Overview

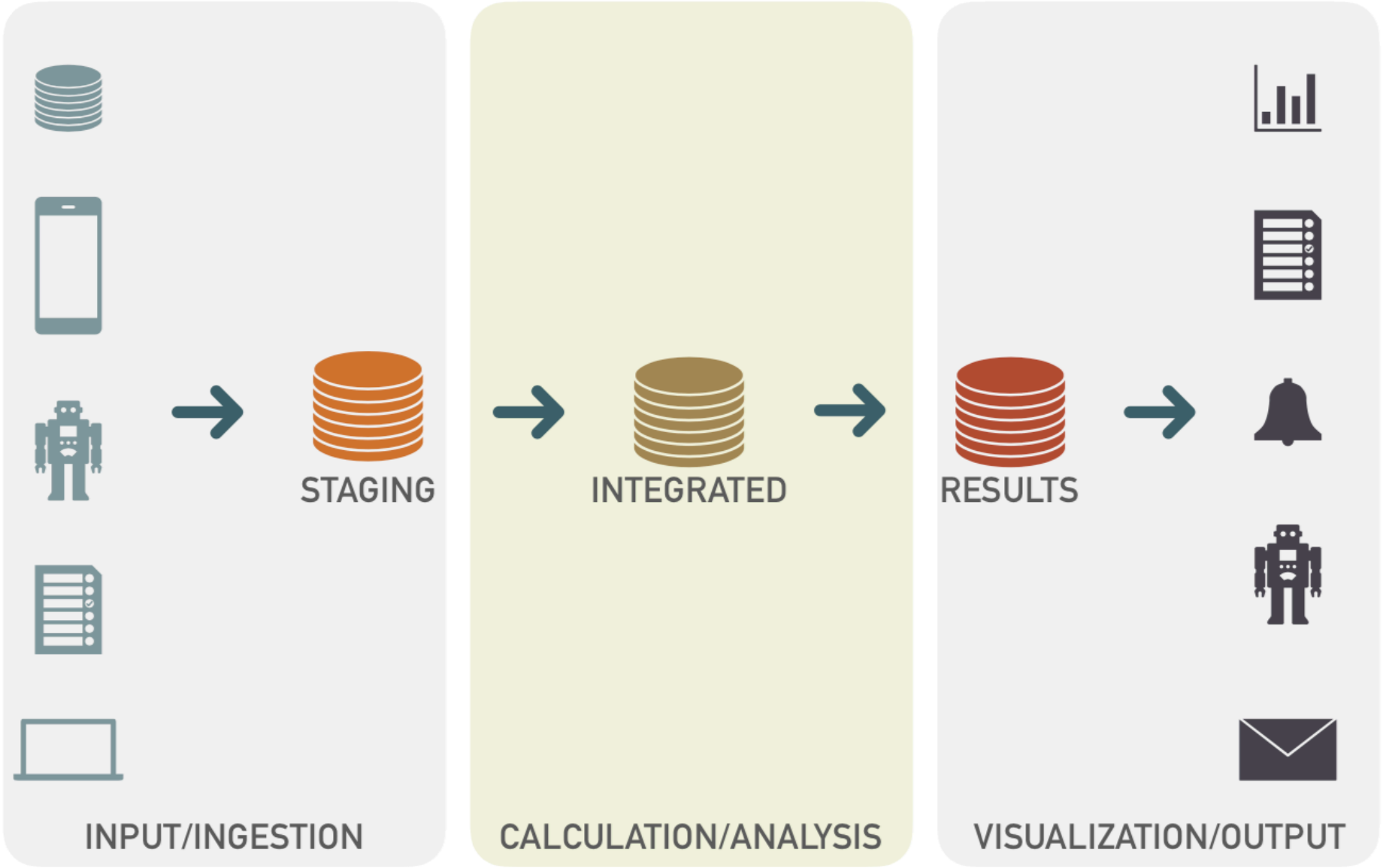
- How to begin DS projects ?
- **Basic Data Science Pipeline**
  - Input and Ingestion
  - **Calculation and Analysis**
    - **Supervised v Unsupervised Learning**
  - Evaluation & Performance
  - Visualization/Output

Data Science Pipeline: **Analysis & Modeling**

- *Most* complex component relates to data mining and modelling.



Calculation/Analysis



Calculation/Analysis

- Simple to Advance ETL
- Machine Learning
- Evaluation & Performance

ETL

- ETL: Extract Transform Load

C1	C2
3	A
22	C
2	A
45	B

C1	C2
5	A
45	B
22	C

```
import pandas as pd
df = pd.DataFrame([[3, 'A'], [22, 'C'], [2, 'A'], [45, 'B']],
                  columns=['C1', 'C2'])

result = df.groupby('C2').sum()
```

Machine Learning

Machine Learning is the training of a model from data that generalizes a decision against a performance measure.

Training a model suggests training examples.

A *model* suggests state acquired through experience.

Generalizes a decision suggests the capability to make a decision based on inputs and anticipating unseen inputs in the future for which a decision will be required.

against a performance measure suggests a targeted need and directed quality to the model being prepared.

Machine Learning

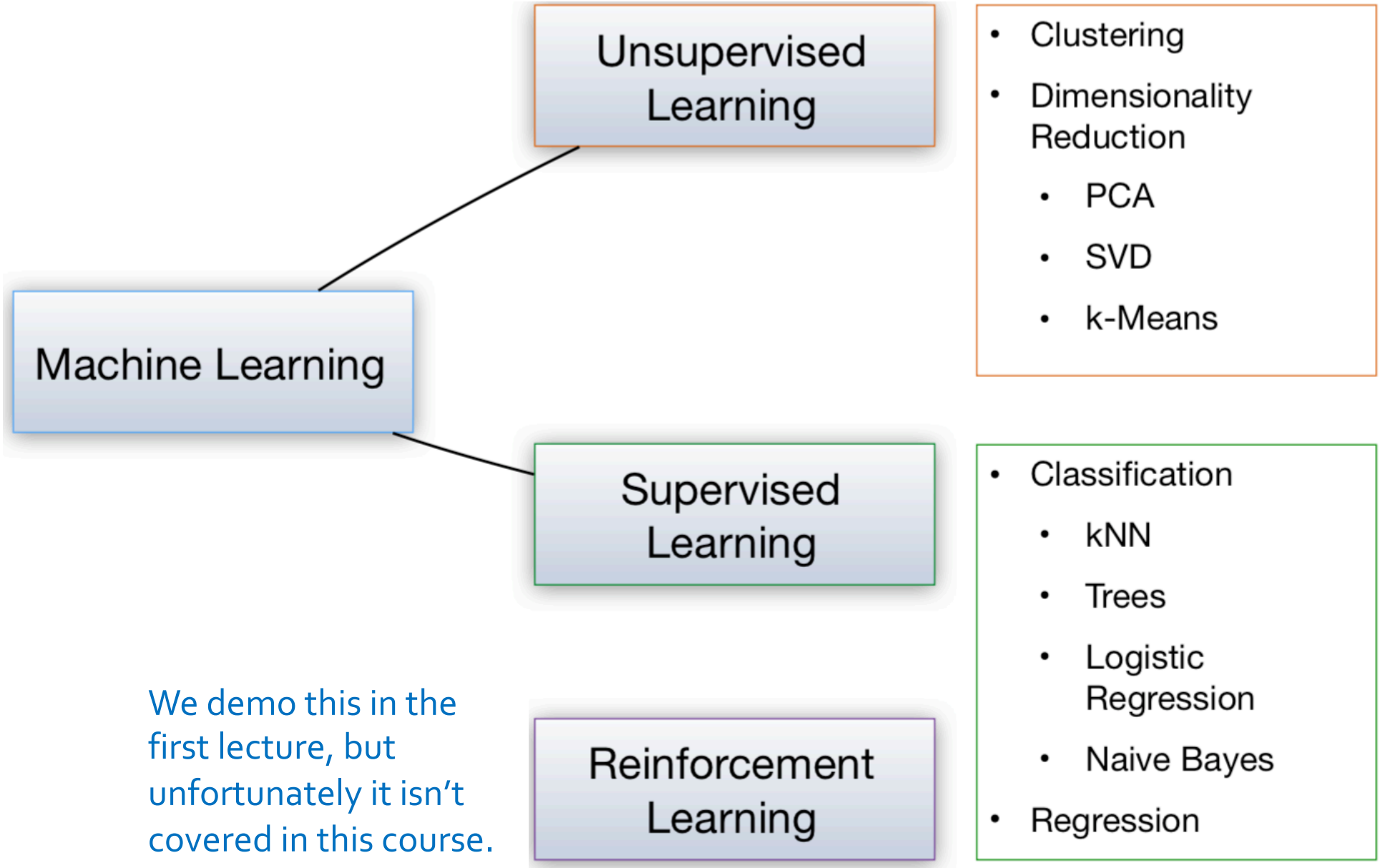
SUPERVISED & UNSUPERVISED

Supervised v Unsupervised Learning

- Supervised Learning:
  An algorithm that learns a function from examples of its inputs and outputs. It requires manually-labelled example data to learn the correct answer for a given query input.
  - e.g. Classification, Regression algorithms

- Unsupervised Learning:
  An algorithm that finds patterns in data when no manually labelled examples are available as inputs. More focused on data exploration and knowledge discovery
  - e.g. Clustering, Graph partitioning algorithms

Types of Machine Learning



We demo this in the first lecture, but unfortunately it isn't covered in this course.

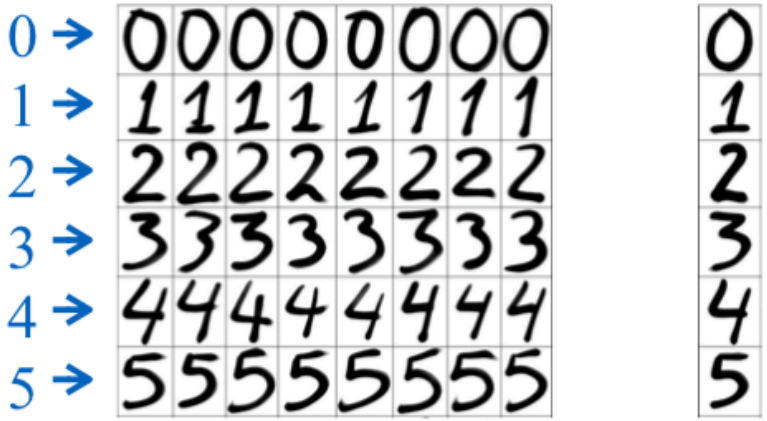
Unsupervised Learning

Split them into 2 groups (clusters)

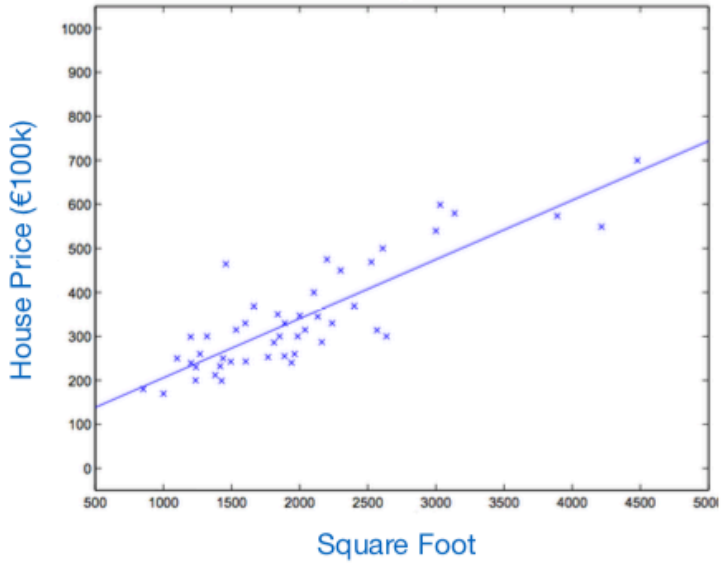
```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=2)
kmeans.fit(X)


print(kmeans.labels_)  # See members in each group
print(kmeans.cluster_centers_)  # See center of each group
```

Supervised Learning

- **Classification:**
  Examples represented by a set of features, which help decide the *class* to which a new query input belongs (i.e. output is a label)



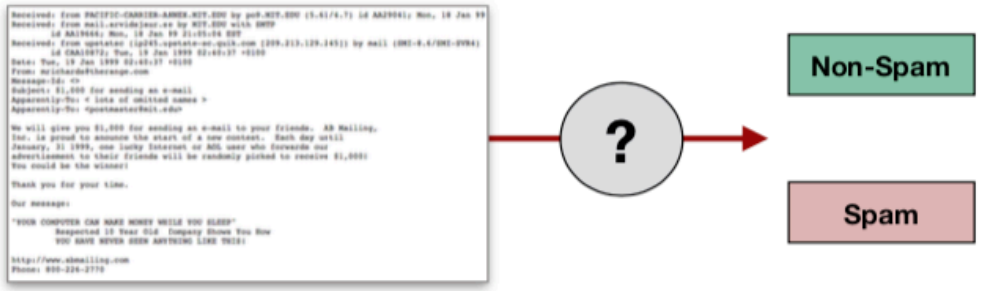
- **Regression:**
  Examples characterised by a set of features, which help decide the value of a continuous output variable (i.e. output is a number)



http://yann.lecun.com/exdb/mnist/
https://towardsdatascience.com/image-classification-in-10-minutes-with-mnist-dataset-54c35b77a38d

Classification Tasks

- **Binary Classification:**
  Assign an input to one of two possible target class labels.



Input — 2 classes

- **Multiclass Classification:**
  Assign an input to one of *M* different target class labels.

Input — 4 classes

Multiclass classification (1)

features (X)

color	weight	texture	pattern
orange	500g	rough	lines
black	400g	smooth	pentagons
.	.	.	.
.	.	.	.
.	.	.	.

label (y)

label
B
S
.
.
.

***Note: This is classification problem. For regression problem, the labeled data are numbers**

Multiclass classification (2)

features (X)

color	weight	texture	pattern
orange	500g	rough	lines
black	400g	smooth	pentagons
.	.	.	.
.	.	.	.
.	.	.	.

label (y)

label
B
S
.
.
.

```
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier()
clf.fit(X, y)

clf.predict(X_test)
```

*Note: This is classification problem. For regression problem, the labeled data are numbers

Overview

- How to begin DS projects ?
- **Basic Data Science Pipeline**
  - Input and Ingestion
  - Calculation and Analysis
    - Supervised v Unsupervised Learning
  - **Evaluation & Performance**
  - Visualization/Output

Evaluation & Performance

```
from sklearn.metrics import accuracy_score, r2_score

accuracy_score(y_true, y_predict)  # Classification
mean_squared_error(y_true, y_predict) # Regression
r2_score(y_true, y_predict)  # Regression
```

Evaluation

```
from numba import jit

@jit(nopython=True)
def foo(n):
    f = 1
    for i in range(1, n+1):
        f = f*i
    return f
```

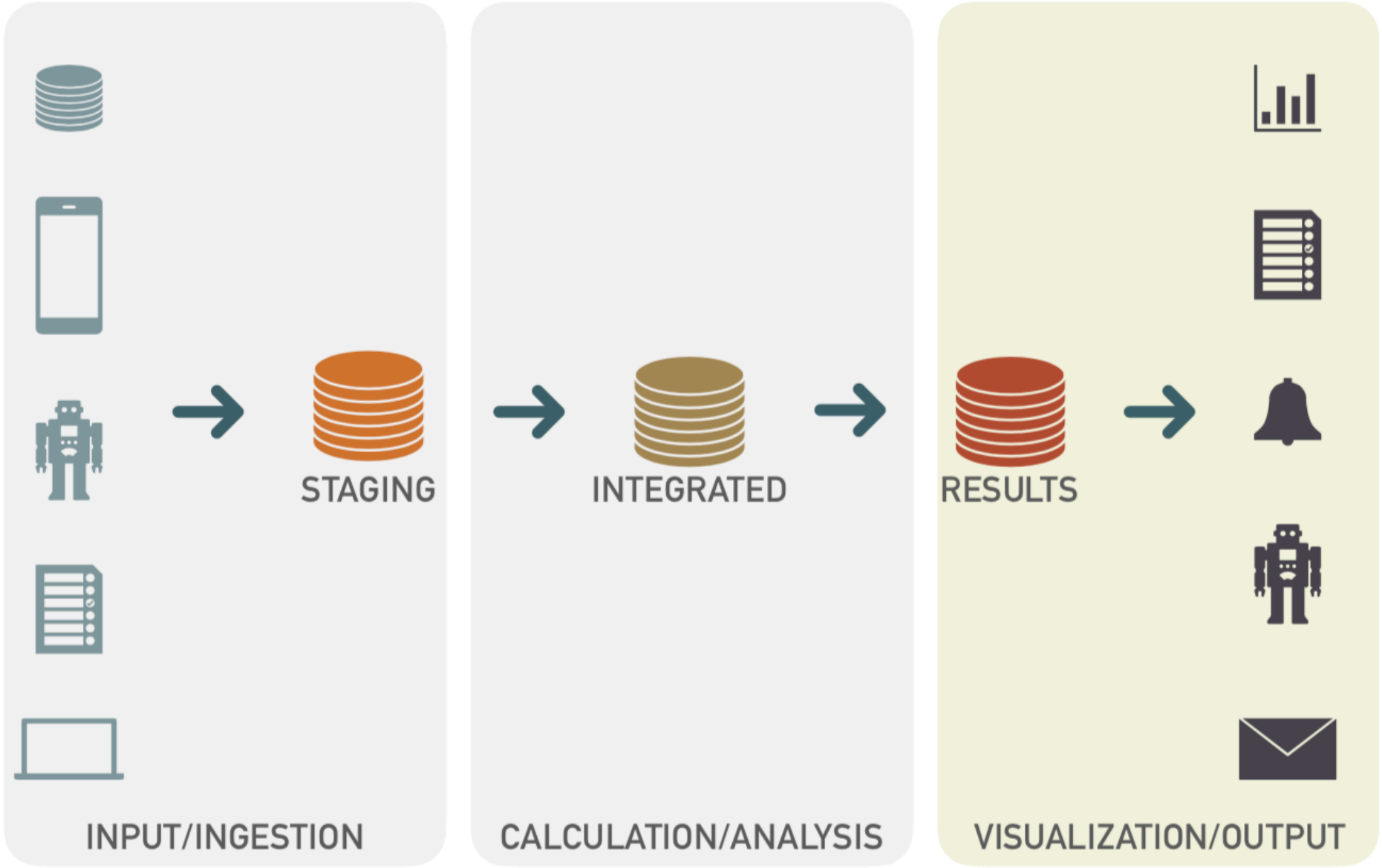
Improve Performance

Other packages
- Cython
- Dask

Overview

- How to begin DS projects ?
- **Basic Data Science Pipeline**
  - Input and Ingestion
  - Calculation and Analysis
    - Supervised v Unsupervised Learning
  - Evaluation & Performance
  - **Visualization/Output**

Visualization/Output



Reports: **Tables** (Excel, CSV, DB Tables)


```
import pandas as pd
from coralinedb import MySQLDB

. . .

df.to_csv("results.csv")  # Save to a csv file
df.to_excel("results.xlxs") # Save to an excel file
db.save_table(df, DB_NAME, TABLE_NAME)
```

Reports: **PDF**


```
from reportlab.pdfgen import canvas

c = canvas.Canvas("hello.pdf")
c.drawString(100,100,"Hello World")
c.showPage()
c.save()
```

hello.pdf

Notifications/Alerts: **Email**


```
import smtplib

server = smtplib.SMTP('smtp.gmail.com', 587)
server.ehlo()
server.starttls()
server.login(SENDER_EMAIL, PASSWORD)

server.sendmail(SENDER_EMAIL, TO_LIST, EMAIL_BODY)
```

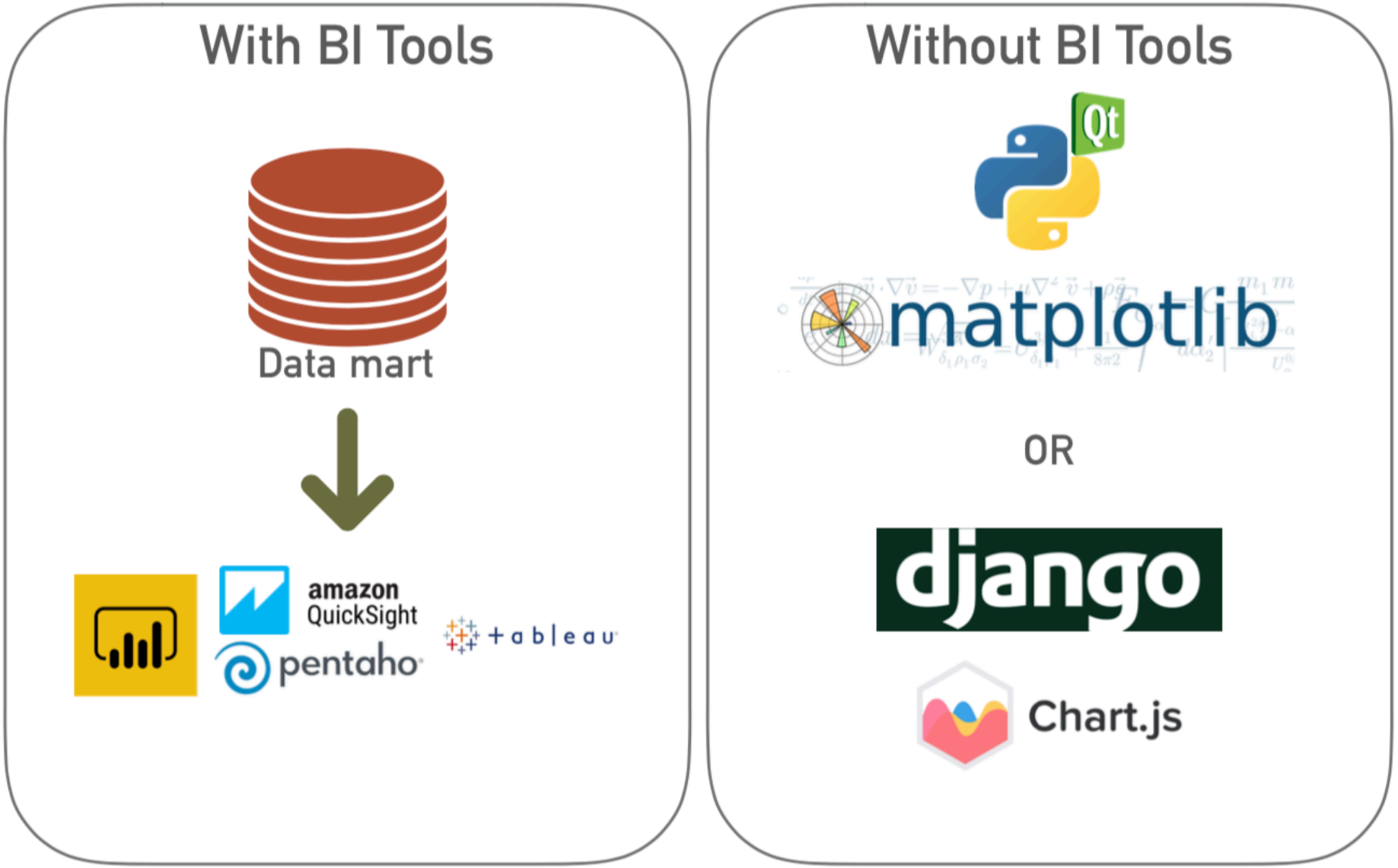
Notifications/Alerts: **Chatbot**


```
import slack

sc = slack.WebClient(token=SLACK_TOKEN)

sc.chat_postMessage(
    channel=CHANNEL_NAME,
    text='Hello!'
)
```

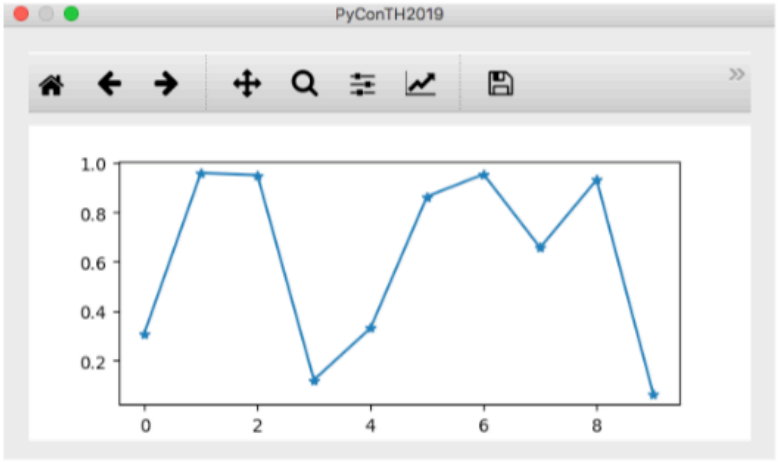
Notifications/Alerts: **Chatbot**

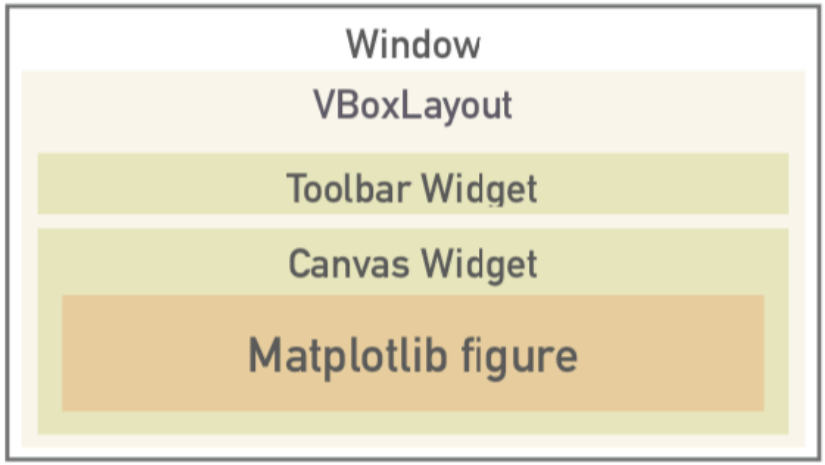

```
import requests
import json
import sys

headers = {
    'Content-Type': 'application/json',
    'Authorization': 'Bearer TOKEN'
}
data = {
    'messages': [ { 'type': 'text', 'text': 'Hello!' } ]
}

requests.post('https://api.line.me/v2/bot/message/broadcast',
                    headers=headers, data=json.dumps(data))
```

Dashboards


With BI Tools
Data mart
amazon
QuickSight
pentaho
+ableau
Without BI Tools
Qt
matplotlib
OR
django
Chart.js

Dashboards: PyQt + Matplotlib



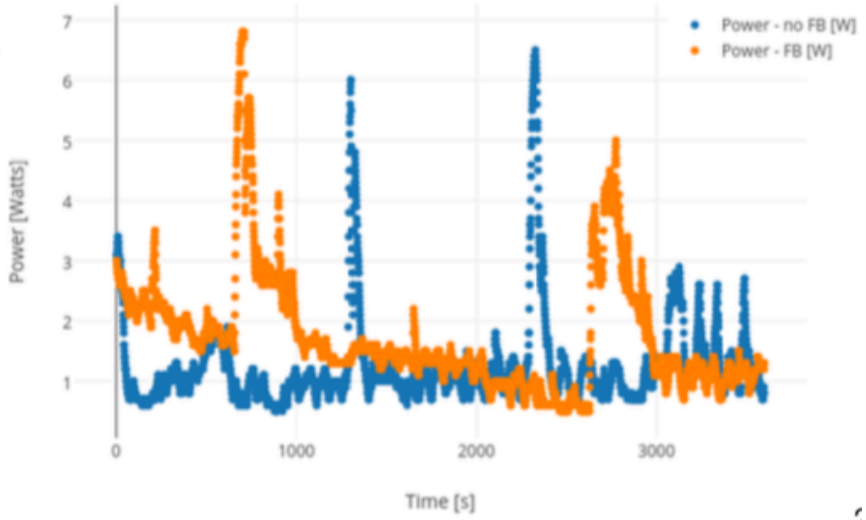



```
# Inside __init__
self.figure = plt.figure()
self.canvas = FigureCanvas(self.figure)
self.toolbar = NavigationToolbar(self.canvas, self)

layout = QVBoxLayout()
layout.addWidget(self.toolbar)
layout.addWidget(self.canvas)
self.setLayout(layout)
```

Dashboards: Other Visualization Packages

- **Seaborn**: A Python visualisation library based on Matplotlib which provides a higher level interface for drawing attractive statistical graphics. (https://stanford.edu/~mwaskom/software/seaborn/)

- **Plotly**: An online analytics and data visualisation tool. Plotly's Python package can be used to make interactive graphs directly from Pandas Data Frames. (https://plot.ly/pandas)



Dashboards: Django + Chart.js

Bar charts
- Vertical
- Horizontal
- Multi axis
- Stacked
- Stacked groups

Line charts
- Basic
- Multi axis
- Stepped
- Interpolation
- Line styles
- Point styles
- Point sizes

Area charts
- Boundaries (line)
- Datasets (line)
- Stacked (line)
- Radar

Other charts
- Scatter
- Scatter - Multi axis
- Doughnut
- Pie
- Polar area
- Radar
- Combo bar/line

Linear scale
- Step size
- Min & max
- Min & max (suggested)

Logarithmic scale
- Line
- Scatter

Time scale
- Line
- Line (point data)
- Time Series
- Combo

Scale options
- Grid lines display
- Grid lines style
- Multiline labels
- Filtering Labels
- Non numeric Y Axis
- Toggle Scale Type

Legend
- Positioning
- Point style
- Callbacks

Tooltip
- Positioning
- Interactions
- Callbacks
- Border
- HTML tooltips (line)
- HTML tooltips (pie)
- HTML tooltips (points)

Scriptable
- Bar Chart
- Bubble Chart
- Pie Chart
- Line Chart
- Polar Area Chart
- Radar Chart

Advanced
- Progress bar
- Content Security Policy

Chart.js

https://www.chartjs.org/

Summary

- Input/Ingestion
  - coralinedb, requests, beautifulsoup4, selenium, flask, django, line-bot-sdk
- Analysis & Calculation
  - pandas, numpy, sklearn, tensorflow, keras, pytorch, dask, numba, cython
- Output/Visualization
  - django, PyQt, matplotlib, pandas, coralinedb, smtplib, email, slackclient, reportlab

Data Science for Business
Introduction to Python – Part 2

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 2.3

Overview

- Commenting
- Defining Functions
  - Return Values
  - Function Composition & Recursion
- Variable Scope
- Working with Strings
- Dynamic Typing
- Converting Between Types
- String Formatting

Commenting Code

- Comments provide a way to write human-readable documentation for your code. Key part of programming!
- In Python code, anything after a # and continuing to the end of the line is considered to be a comment and is ignored.

```
x = 5*4 # ignore this
```

```
x = 5 + 3 # + 10
```

- Multi-line comments can also be added to Python code, using triple quoted strings (i.e. 3 single or 3 double quote characters):

```
'''
This is a single quoted
multi-line comment.
'''
```

```
"""
This is a double quoted
multi-line comment.
"""
```

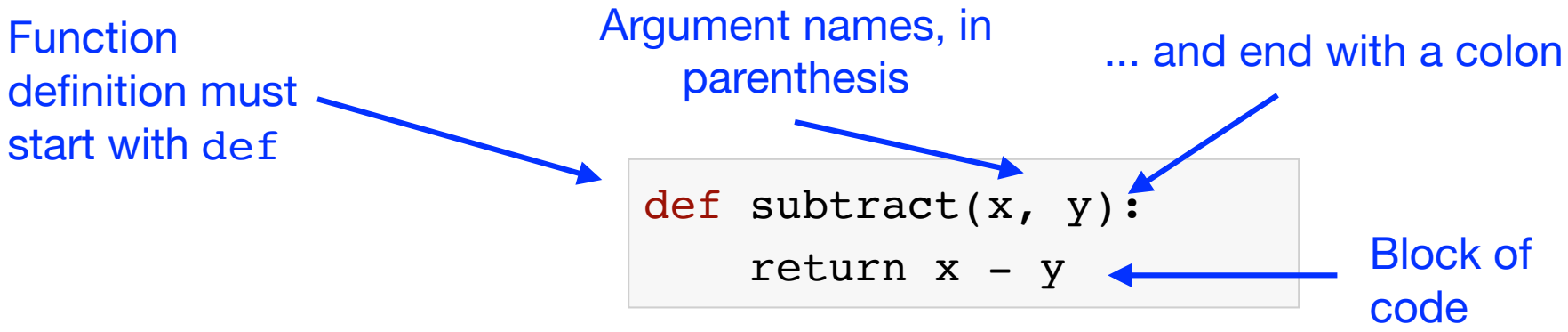
- Note: if you are inside an indented block of code, multi-line comments need to be indented too! Not the case for # comments.

Functions in Python

- Functions in Python represent a block of reusable code to perform a specific task.
- Two basic types of functions:
  - Built-in functions: these usually a part of existing Python packages and libraries.
  - User-defined functions: written by programmers to meet certain requirements of a task or project.
- User-defined functions only need to be written once, and can then potentially be reused multiple times in different applications. They provide a means of making your code more organised and easier to maintain.

Defining Functions

- We create a new user-defined function in Python using the def keyword, followed by a block of code. Specifically we need:
  1. A function name
  2. Zero or more input arguments
  3. An optional output value, specified via return keyword
  4. A block of code



- Call the new function using parenthesis notation:

```
z = subtract(5,3)
```

```
z = subtract(8,12)
```

Defining Functions

- In the simplest case, we can define a function that does not take any input. More often, we will want to pass values to a function as input arguments.

```
def sayhello():
    print("hello!")
```

```
sayhello()
```

```
hello!
```

```
def add(x, y):
    print( x + y )
```

```
add(3, 5)
```

```
8
```

- Some arguments for a function can be optional and do not need to be specified if we provide a default value:

```
def add(x, y=1):
    print( x + y )
```

```
add(3, 5)
```

```
8
```

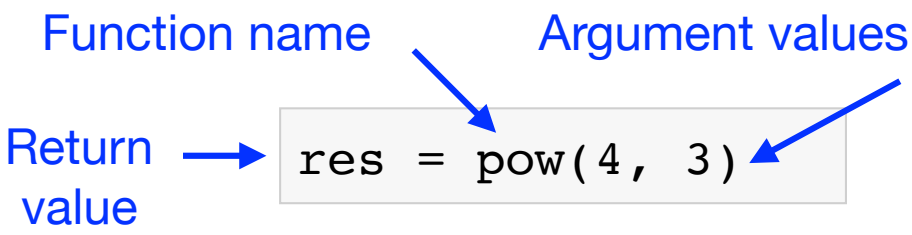
```
add(3)
```

```
4
```

If a value for argument y is not specified, the default value will be y=1

Calling Functions

- Functions are run when you call them with parenthesis notation:



e.g. call function `pow()` to calculate 4 raised to the power of 3

- We can also use keyword arguments that are specified by name.
- Example: One available keyword argument for `print()` is `sep`, which specified what characters should be used to separate multiple values:

```
print(5, 10, 15)
```

```
5 10 15
```

```
print(5, 10, 15, sep="_")
```

```
5_10_15
```

- When non-keyword arguments are used together with standard keyword arguments, keyword arguments must come at the end.

Returning Values

- The type of value returned by a function does not need to be specified in advance.
- Often it is useful to have multiple return statements, one in each branch of a conditional.
- Code that appears after a return statement cannot be reached and will never be executed.

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
```

```
def absolute_value(x):
    if x < 0:
        return -x
    else:
        return x
    return 0
```

Code will never run

- If no return value is specified, a function will return None by default.

```
def square( x ):
    y = x * x
```

```
res = square( 3 )
print(res)
```

```
None
```

Returning Values

- Python allows multiple values to be returned from a single function by separating the values with commas in the return statement.
- Multiple values get returned as a tuple.

```
def min_and_max(values):
    vmin = min(values)
    vmax = max(values)
    return vmin, vmax
```

Two values returned

```
values = [5, 19, 3, 11, 24]
result = min_and_max(values)
print(result)
```

```
(3, 24)
```

Result is a tuple with 2 values

- Unpacking: Multiple variables can be assigned the multiple values returned by the function in a single statement.

```
x, y = min_and_max(values)
print(x)
print(y)
```

Put the 1st returned value in `x`
Put the 2nd returned value in `y`

```
3
24
```

Defining Functions: Examples

- Functions for Celsius to Fahrenheit conversion, and vice-versa:

```
def celsius_to_fahrenheit(c):
   return (9.0/5.0 * c) + 32
```

```
def fahrenheit_to_celsius(f):
   return (f - 32.0) * 5.0 / 9.0
```

```
for ctemp in range(0,30,5):
   print("Celsius", ctemp)
   ftemp = celsius_to_fahrenheit(ctemp)
   print("Fahrenheit", ftemp)
```

```
for ftemp in range(50,80,5):
   print("Fahrenheit", ftemp)
   ctemp = fahrenheit_to_celsius(ftemp)
   print("Celsius", ctemp)
```

```
Celsius 0
Fahrenheit 32.0
Celsius 5
Fahrenheit 41.0
Celsius 10
Fahrenheit 50.0
Celsius 15
Fahrenheit 59.0
Celsius 20
Fahrenheit 68.0
Celsius 25
Fahrenheit 77.0
```

```
Fahrenheit 50
Celsius 10.0
Fahrenheit 55
Celsius 12.777777777777779
Fahrenheit 60
Celsius 15.555555555555555
Fahrenheit 65
Celsius 18.333333333333332
Fahrenheit 70
Celsius 21.11111111111111
Fahrenheit 75
Celsius 23.88888888888889
```

Defining Functions: Examples

- Function for finding the Least Common Multiple (LCM) of two numbers. That is, the smallest positive integer that is perfectly divisible by the two given numbers.

Define function lcm with 2 input parameters

Use while loop to test increasingly larger values

Return value found to be the LCM of x and y

Call our new function

```
def lcm(x, y):
  # choose the greater number
  if x > y:
    greater = x
  else:
    greater = y
  # keep increasing until we find the answer
  while True:
    if(greater % x == 0) and (greater % y == 0):
      answer = greater
      break
    greater += 1
  return answer
```

```
print( lcm(5,7) )
```

```
35
```

```
print( lcm(12,30) )
```

```
60
```

Function Composition & Recursion

- You can call one function from inside another. Several simple functions can be combined to create more complex ones.

```
def square(x):
    return x*x
```

```
def negative(x):
    return -x
```

```
def calc_score(x, y):
    a = square(x)
    b = negative(y)
    return a + b
```

```
calc_score( 3, 4 )
```

```
5
```

- Recursive functions repeatedly call themselves either directly or indirectly in order to loop.

```
def mysum( l ):
    if len(l)==0:
        return 0
    return l[0] + mysum(l[1:])
```

```
mysum( [1, 2, 3] )
```

```
6
```

Example recursively sums a list of numbers.
What's actually happening here:

```
mysum([1, 2, 3])
mysum([2, 3])
mysum([3])
mysum([])
```

Variable Scope

- Scope: refers to the ability to access certain variables in a certain part of our code.
- Code written at the top level (i.e. not in a nested block) is global. These variables are accessible everywhere. Variables defined in a function are local, and are accessible only in that function.

This variable is **global**, it is accessible everywhere

This variable is **local**, it is only accessible in its own function

```
gvar = "This is global"

def myfunction():
  lvar = "This is local"
  print("global_var:", global_var)
  print("lvar:", lvar)
```

```
myfunction()
```

```
gvar: This is global
lvar: This is local
```

We cannot access the local variable outside the function

```
print("gvar:", gvar)
print("lvar:", lvar)
```

```
gvar:  This is global
NameError: name 'lvar' is not defined
```

Variable Scope

- Global variables can be accessed in a function, but normally cannot be modified.

```
x = 20

def myfunction():
    x = 15

myfunction()
print(x)
```

```
20
```

We are creating a new local variable with the same name!

No change to the original!

- We can use the `global` statement to tell Python that a function plans to change one or more global variables.

```
x = 20

def myfunction():
   global x
    x = 15

myfunction()
print(x)
```

```
15
```

Tell Python we plan to modify the global variable `x` in this function

Value of global variable has been changed

Strings Revisited

- Recall Python strings can be defined using either single or double quotes.
- Python also has block strings for multi-line text, defined using triple quotes (single or double).
- Escape sequences: backslashes are used to introduce special characters.

Escape	Meaning
\n	Newline character
\t	Tab character
\r	Return character (Windows)
\\	Backslash - same as one '\'

```
mytext = "this is some text"
```

```
mytext = 'this is some text'
```

```
s = """School of CS,
UCD,
Belfield"""
```

```
s
```

```
'School of CS,\nUCD,\nBelfield'
```

```
address = "UCD\tBelfield"
address
```

```
'UCD\tBelfield'
```

```
address = "UCD\tBelfield"
print(address)
```

```
UCD  Belfield
```

Working With Strings

- Strings can be viewed as sequences of characters of length N.

```
s = "BELFIELD"
```

- As such, we can apply many standard list operations and functions to Python strings.
- Characters and substrings can be accessed using square bracket notation just like lists.

```
s[2]
```

```
L
```

Access a character by index (position)

```
s[1:4]
```

```
ELF
```

Create substrings via slicing

```
len(s)
```

```
8
```

Length of the string i.e. number of characters

- Strings can be concatenated together using + operator

```
t = "ucd" + "_" + "belfield"
t
```

```
'ucd_belfield'
```

String Functions

- Strings have associated functions to perform basic operations.

Syntax
`<string_variable>.<function>(argument1, argument2, ...)`

- Example of string manipulation functions - case conversion:

```
s = "Hello World"
s.upper()
```

```
'HELLO WORLD'
```

```
s = "Hello World"
s.lower()
```

```
'hello world'
```

```
s = "Hello World"
s.swapcase()
```

```
'hELLO wORLD'
```

```
s = "Hello World"
t = s.upper()
print(s)
```

```
'Hello World'
```

```
print(t)
```

```
'HELLO WORLD'
```

These string manipulation functions make a copy of the original string, they do not change the original string.

String Functions - Find & Replace

- Strings have associated functions for finding characters or substrings.

Search for the first occurrence of the specified substring.

```
s = "Hello World"
s.find("World")
```

```
6
```

Returns either the index of the substring, or -1 if not found.

```
s.find("UCD")
```

```
-1
```

Count number of times a substring appears in a string.

```
x = "ACGTACGT"
x.count("T")
```

```
2
```

```
x = "ACGTACGT"
x.count("U")
```

```
0
```

- We can also replace characters or complete substrings. This creates a new copy of the original string.

```
x = "ACGTACGT"
x.replace("T","V")
```

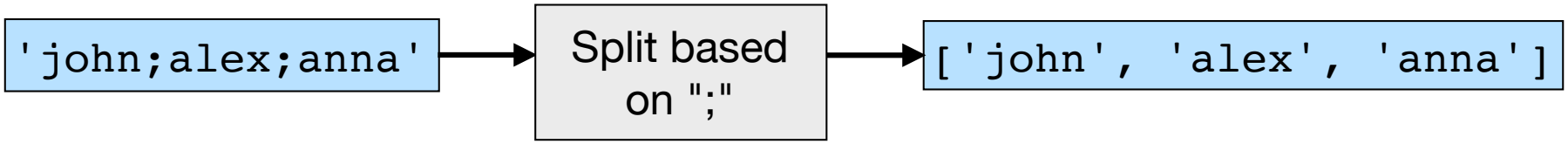
```
'ACGVACGV'
```

```
x = "Hello World"
x.replace(" ","_")
```

```
'Hello_World'
```

String Functions - Split & Join

- Use the `split()` function to separate a string into multiple parts, based on a delimiter - i.e a separator character or substring.



Output is a list containing multiple string values

```
names="john;alex;anna"
names.split(";")
```

```
['john', 'alex', 'anna']
```

```
data = "5,6,11,12"
data.split(",")
```

```
['5', '6', '11', '12']
```

- Use the `join()` function to concatenate a list of strings into a single new string. All values in the list must be strings.

```
<separator>.join(list)
```

```
l = ["dublin","cork","galway"]
"$".join(l)
```

```
'dublin$cork$galway'
```

Dynamic Typing

- Python uses a dynamic typing model for variables:
  - Variables do not need to be declared in advance.
  - Variables do not have a type associated with them, values do.

```
x = 2
x = "some text"
x = True
```

We can change the type of a variable by simply assigning it a new value

- Python uses strong dynamic typing
  - Applying operations to incompatible types is not permitted.
  - May need to remember the type of value your variables contain!

```
1 + "hello"
```

```
Traceback (most recent call last):
  File "", line 1, in ?
TypeError: unsupported operand type(s) for +: 'int' and ‘str'
```

Cannot add an integer to a string!

Converting Between Types

- Since mixing incompatible types is not permitted, we use built-in conversion functions to change a value between basic types.

Use the `str()` function to convert any value to a string

```
str(27)
```

```
'27'
```

```
str(0.45)
```

```
'0.45'
```

We can also convert strings to numeric values using `int()` and `float()`

```
s = "145"
int(s)
```

```
145
```

```
s = "1.325"
float(s)
```

```
1.325
```

- Not all strings can be converted to numeric values…

```
int("UCD")
```

```
ValueError: invalid literal for int() with base 10: 'UCD'
```

```
float("ax0.353")
```

```
ValueError: could not convert string to float: 'ax0.353'
```

Converting Between Types

- Often use the string `split()` function in conjunction with type conversion when parsing simple data files...

```
data = "0.19,1.3,4.5,3,12"
parts = data.split(",")
print( parts )
```

```
['0.19', '1.3', '4.5', '3', '12']
```

Call `split()` to divide the original string into a list of strings

```
values = []
for s in parts:
    values.append( float(s) )
```

Convert each sub-string to a float value

```
print( values )
```

```
[0.19, 1.3, 4.5, 3.0, 12.0]
```

```
type( values[0] )
```

```
<class 'float'>
```

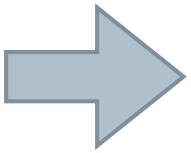
String Formatting

- In Python, we can concatenate multiple variables of different types into a single string, using the % operator. The format string provides the recipe to build the string, containing zero or more placeholders.

Syntax
`"<format string>" % (<var1>,<var2>,...,<varN>)`

- The placeholders get substituted for the list of values that you provide after the % symbol. The number of placeholders in the format string must equal the number of values!

```
" %s was born in %s in %d " % ( "John", "Dublin", 1985 )
```



```
" John was born in Dublin in 1985 "
```

String Formatting

- Special placeholder codes are used when building a format string.
- Each placeholder should correspond to the type of the value that will replace it.

Building format strings

Code	Variable Type
`%d`	Integer
`%f`	Floating point
`%.Nf`	Float (*N* decimal places)
`%s`	String (or any value)
`%%`	The '%' symbol

`\t`	Tab character
`\n`	Newline character

```
x = 45
y = 0.34353
z = "text"
s = "%d and %.2f and some %s" % (x,y,z)
print( s )
```

```
'45 and 0.34 and some text'
```

```
s2 = "%f => %.0f or %.4f" % (y,y,y)
print( s2 )
```

```
0.343530 => 0 or 0.3435
```

Data Science for Business
Next Steps in Python – Part 3

Asst. Prof. Teerapong Leelanupab (Ph.D.)
Faculty of Information Technology
King Mongkut's Institute of Technology Ladkrabang (KMITL)

Week 2.4

Overview

- File Input/Output
- Error Handling
  - Python Error Messages
  - Exceptions
- Python Modules
- Built-in modules
- Basic Mathematical and Random Functions
- Accessing Files and Directories
- Writing Python Scripts
- Command Line Arguments

File Input/Output

- Files are special types of variables in Python, which are created using the `open()` function. Remember to `close()` the file when you are finished!

```
f = open( "<filepath>", "<action>" )
     .....
f.close()
```

"r": read
"w": write
(default is read)

- **Reading files:** After opening a file to read, you can use several functions to access plain-text data:

`read()`	read the full file
`readline()`	read a full line from a file
`readlines()`	read all lines from a file into a list

Need to strip line endings

```
f = open("test.txt", "r")
lines = f.readlines()
f.close()
for line in lines:
    line = line.strip()
    print(line)
```

Read all lines from a file into a list

Example: Reading Files

- Read a list of names and student numbers, storing the information in a dictionary.

Input: students.txt

```
17211426,Stephanie Gale
16212133,Jill Doyle
13388136;Pat Gilbert
17211824,Daryl Bishop
16216364,Carlos Alvarado
17211833,Alison Rogers
17212834,Neil Smith
13312141,Sandra Wright
```

```
register = {}
fin = open("students.txt","r")
lines = fin.readlines()
fin.close()
for line in lines:
   line = line.strip()
   parts = line.split(",")
   student_id = int(parts[0])
   fullname = parts[1]
   register[student_id] = fullname
```

Need to strip line endings

- Display the new contents of the dictionary:

```
for sid in register:
   print( "%d -> %s" % (sid, register[sid]) )
```

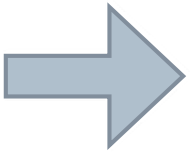
```
17211426 -> Stephanie Gale
16212133 -> Jill Doyle
13388136 -> Pat Gilbert
17211824 -> Daryl Bishop
16216364 -> Carlos Alvarado
17211833 -> Alison Rogers
17212834 -> Neil Smith
13312141 -> Sandra Wright
```

Working with Files

- **Writing files:** After opening a file to write, use the `write()` function to output strings to the file.

```
names = ["Mark","Lisa","Alice","Bob"]
f = open("out.txt","w")
for name in names:
   f.write( name )
   f.write( "\n" )
f.close()
```

Need to explicitly move to next line



out.txt

```
Mark
Lisa
Alice
Bob
```

- Note: By default Python will overwrite an existing file with the same name if it already exists.
- To add data to the end of an existing file, use append mode "a" when opening the file:

```
f = open("out.txt","a")
```

Indicates open in append mode

Example: Writing Files

- Read a list of lines from one file, write the contents back out to a second file with an additional prefix.

Open two files: one to read ("r"), one to write ("w")

```
fin = open("sample.txt","r")
fout = open("modified.txt","w")
for line in fin.readlines():
   fout.write("Copy: ")
   fout.write(line)
fin.close()
fout.close()
```

Note that the lines already end with a new line character

Input: sample.txt

```
County Dublin
County Galway
County Limerick
County Louth
County Wexford
```

Output: modified.txt

```
Copy: County Dublin
Copy: County Galway
Copy: County Limerick
Copy: County Louth
Copy: County Wexford
```

Writing Files + String Formatting

- We can write a variety of Python variables into a text file on multiple lines.
- Note we must convert values to strings before calling `write()`
- We can do this either using type conversion or using string formatting.

```
year = 2013
d = {"a":3.0, "b":4.5, "c":9.87}
fout = open("data.txt","w")
fout.write( str(year) + "\n" )
for key in d:
  fout.write(key + " " + str( d[key] ) + "\n")
fout.close()
```

```
year = 2013
d = {"a":3.0, "b":4.5, "c":9.87}
fout = open("data.txt","w")
fout.write( "%d\n" % year )
for key in d:
  fout.write("%s,%.1f\n" % (key, d[key]))
fout.close()
```

Output: data.txt

```
2013
a,3.0
b,4.5
c,9.9
```

Python Error Messages

- A key programming task is debugging when a program does not work correctly or as expected.
- If Python finds an error in your code, it raises an exception.
  e.g. We try to convert incompatible types
  e.g. We try to read a non-existent file
  Also... When we have invalid syntax in our code (a "typo")

```
number = int("UCD")
```

```
Traceback (most recent call last):
  File "test.py", line 1, in <module>
    number = int("UCD")
ValueError: invalid literal for int() with base 10: 'UCD'
```

Where the error occurred

Type of exception that has occurred

Text describing the error

Python Error Messages

```
d = {"Ireland": "Dublin"}
d["France"]
```

```
Traceback (most recent call last):
  File "test2.py", line 2, in <module>
    d["France"]
KeyError: 'France'
```

Where the error occurred

Type of exception that has occurred

```
def showuser(username):
  print(user_name)

showuser("bob")
```

```
Traceback (most recent call last):
  File "test3.py", line 4, in <module>
    showuser("bob")
  File "test3.py", line 2, in showuser
    print(user_name)
NameError: name 'user_name' is not defined
```

Error originated here

Exception Handling

- By default, an exception will terminate a script or notebook.
- We can handle errors in a structured way by "catching" exceptions. We plan in advance for errors that might occur...

General Format

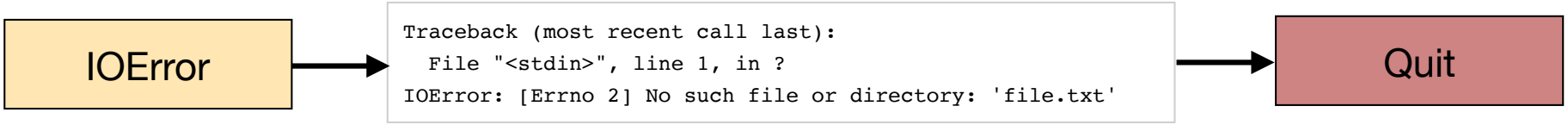
```
try:
   <block of code>
except <errorType>:
   <error handling block>
```

```
try:
    f = open("file.txt","r")
except IOError:
    print("Got error, but continuing anyway")
```

Code where error might occur

Error handling code

Without exception handling



With exception handling

More Complex Exception Handling

- We can get details about the specific cause of the exception - i.e. the error message.

```
try:
    x = int("ucd")
except ValueError as e:
    print("Error:",e)
```

```
Error: invalid literal for int()
with base 10: 'ucd'
```

- Try statements can include an optional `finally` clause that always executes regardless of whether an exception occurs.

```
try:
    x = int(some_string)
except ValueError:
    print("Conversion error")
finally:
    print("Always print this")
```

- Multiple except clauses: A try statement can check for several different exception types in sequence.

```
try:
    x = int(some_string)
    answer = x/y
except ValueError:
    print("Conversion error")
except ZeroDivisionError:
    print("Dividing by 0!")
```

Example: Exception Handling

- Common file input tasks required handling the case where we try to read from a file path that does not exist...

```
file_path = "/home/user/data.csv"
try:
    fin = open(file_path,"r")
    content = fin.read()
    print(content)
    fin.close()
except IOError:
    print("Unable to read from file", file_path)
finally:
    print("Process complete")
```

- If `file_path` does not exist, the except code block will be run:

```
Unable to read from file /home/user/data.csv
Process complete
```

Python Modules

- Module: A single file of Python code, often containing functions and variables related to a particular programming task.
- Accessing functions in a module requires first importing the module for use in the current Python environment. Two different ways to do this.

```
import <module_name>
```

Import the whole module in its entirety

```
from <module_name> import <something>
```

Import a subset of functionality i.e. just certain functions or variables which we require

- Examples of imports:

```
import math
import sys, os
```

Import whole modules. Note we can import multiple modules on each line.

```
from sys import exit
from math import sqrt, log
```

Import a subset of functionality. This can be one or more functions or variables.

Python Modules

- If we have imported an entire module, we then prefix the function names with the module name followed by a dot (i.e. dot notation).

```
import math
x = math.sqrt(9)
print(x)
```

```
3
```

If we forget to import the module...

```
x = math.sqrt(9)
```

```
NameError: name 'math' is not defined
```

- If we have imported a subset of a module, we can access all of those functions or variables without requiring the module name as a prefix.

```
from math import sqrt, log
x = sqrt(9)
y = log(2)
print(x+y)
```

```
3.6931471805599454
```

Now if we include the prefix in the call, it won't work...

```
x = math.sqrt(9)
```

```
NameError: name 'math' is not defined
```

Built-in Modules

- The Python standard library contains a large number of built-in modules for performing different tasks:

Name	Description
sys	Program control, command line arguments (e.g. exit, argv)
math	Basic mathematical functions (e.g. sqrt, exp, sin, cos)
os	File/directory operations (e.g. listdir, mkdir, rmdir)
random	Generate pseudo-random numbers (e.g. random, randint)
re	Provides regular expression matching and replacement operations

Full list of standard modules: https://docs.python.org/3/library

Basic Mathematical Functions

- Python has a `math` module that provides most basic mathematical functions. Before we can use it, we have to import it...

```
import math
x = math.sqrt(9)
print(x)
```

- We can then call various familiar functions:

```
ratio = signal_power / noise_power
decibels = 10 * math.log10(ratio)
```

```
radians = 0.6
height = math.sin(radians)
```

- We can also access variables contained in the module:

```
degrees = 45
radians = degrees / 360.0 * 2 * math.pi
answer = math.sin(radians)
```

`pi` is a variable defined in the math module

- Alternatively we could have imported the subset we required:

```
from math import pi, sin
degrees = 45
radians = degrees / 360.0 * 2 * pi
answer = sin(radians)
```

now no prefix required to access `pi` variable

Random Number Generation

- The `random` module provides functions that generate pseudorandom numbers - i.e. not truly random because they are generated by a deterministic computation, but are generally indistinguishable from them.
- The function `random()` returns a random float between 0.0 and 1.0. Each time we call it, we get the next number from a series.

```
import random
for i in range(4):
    y = random.random()
    print(y)
```

```
0.21660381103801063
0.10168268009500758
0.5845753014438958
0.4436497677624016
```

Prefix functions with random. after importing!

- The module contains many other functions - e.g. `randint()` returns a random integer from the specified range.

```
import random
for i in range(4):
    y = random.randint(10,20)
    print(y)
```

```
20
14
20
15
```

e.g. return a random integer, from 10 to 20 inclusive.

Accessing Files and Directories

- The built-in `os` module provides comprehensive functionality for working with files and directories.

Get the current working directory

```
import os
print( os.getcwd() )
```

```
/home/user/Downloads
```

Prefix functions with os. after importing!

Change the current working directory

```
os.chdir("/usr/local")
print( os.getcwd() )
```

```
/home/user/Downloads
```

Get list of files in specified directory

```
os.listdir("/home/user/Documents")
```

```
['letter.doc','names.xls','results.doc']
```

Delete a file

```
os.remove("letter.doc")
```

Create a directory

```
os.mkdir("python")
```

Writing Python Scripts

- As well as IPython Notebooks, we will frequently need to write `.py` files either as stand-alone scripts or to create new modules.
- Many editors available for creating Python script files
  e.g. PyDev, PyCharm, Gedit, Sublime Text, Textmate, Notepad++
- Basic steps:
  1. Write your script in a text editor.
  2. Save your script as a `.py` file - e.g. hello.py
  3. In the terminal, change to the directory containing the script.
  4. Run python, passing the script filename as the first argument:

File: hello.py

```
for i in range(3):
    print("Hello world")
```

```
~> python hello.py
Hello world
Hello world
Hello world
~>
```

Debugging Python Scripts

- Debugging becomes a little more complicated when working with scripts. Need to be able to read output of Python error messages!
- Follow the traceback provided by Python to identify the original source of the error in the script...

File: squares.py

```
1  def add_squares( x, y ):
2     sx = x * x
3     sy = y * y
4     return sx + sy
5
6  result1 = add_squares( 3, 4 )
7  print("Result 1 = %d" % result1)
8
9  result2 = add_squares( 3, "9" )
10 print("Result 2 = %d" % result2)
```

```
~> python squares.py
Result 1 = 25
Traceback (most recent call last):
  File "squares.py", line 9, in <module>
    result2 = add_squares( 3, "9" )
  File "squares.py", line 3, in add_squares
    sy = y * y
TypeError: can't multiply sequence by non-
int of type 'str'
```

Error message is `TypeError: can't multiply sequence by non-int of type 'str'`

Error originated in `line 3, in add_squares   sy = y * y`

Command Line Arguments

- Many programs allow command-line arguments to be specified when they are run.
- A command-line argument is the information that follows a program's name on the terminal / command line, when it is executed.
- These arguments are used to pass information (e.g. file paths, options etc). to the program.

```
myprog argument1 argument2 argument3
```

```
cd /home/alice/code
```

```
find /home/alice -type f -name README
```

```
python myscript.py
```

Command Line Arguments

- Command-line arguments are used to pass information when you start a Python script from the terminal.

```
python name.py fred lisa john
```

script name

user-specified arguments

- The `argv` variable in the `sys` module contains the list of command line arguments passed to the current script.

File: name.py

```
import sys

print("Got %d arguments" % len(sys.argv))
print("Script name is %s" % sys.argv[0])

for name in sys.argv[1:]:
  print("Hello %s" % name )
```

```
python name.py fred lisa john
```

```
Received 4 arguments
Script name is name.py
Received parameter fred
Received parameter lisa
Received parameter john
```

Contents of `sys.argv` above is

`['name.py', 'fred', 'lisa', 'john']`

Using Scripts as Modules

A single script file corresponds to a single module. You can import any script into another script.

File: fibo.py

```
# Build Fibonacci series up to n
def calc_fib(n):
    series = []
    a = 0
    b = 1
    while b < n:
        series.append(b)
        temp = a
        a = b
        b = temp + b
    return series
```

Exclude the .py file extension from module name when importing

File: testfib.py

```
import fibo
series = fibo.calc_fib(4)
print(series)
```