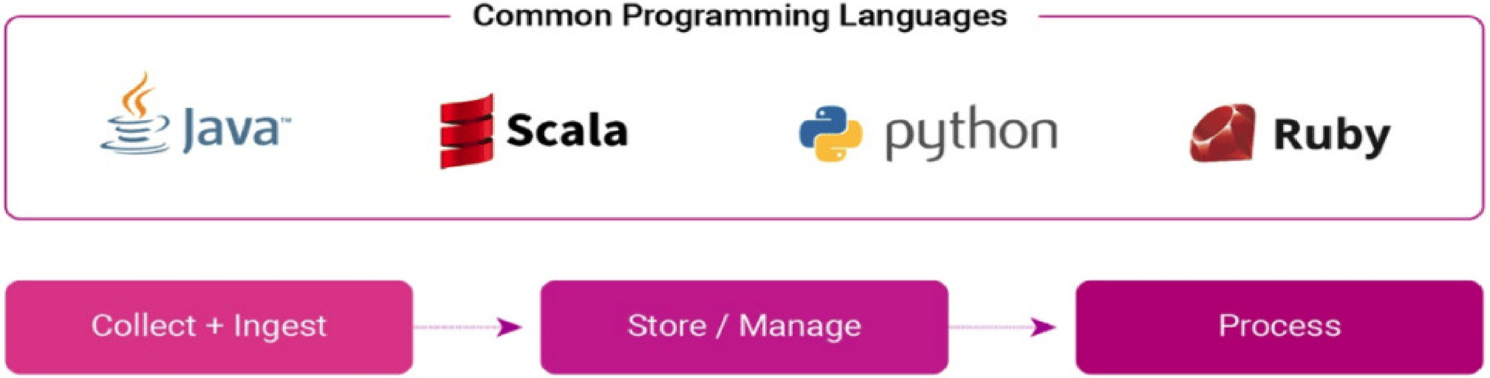
Introducción & Best Practices con Python

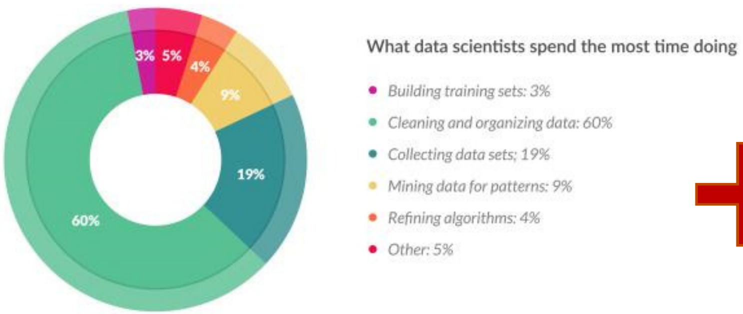
Juan Luis Rivero

Innova-tsn

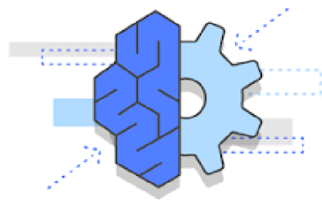
Proyectos de Big Data

THE DATA PIPELINE



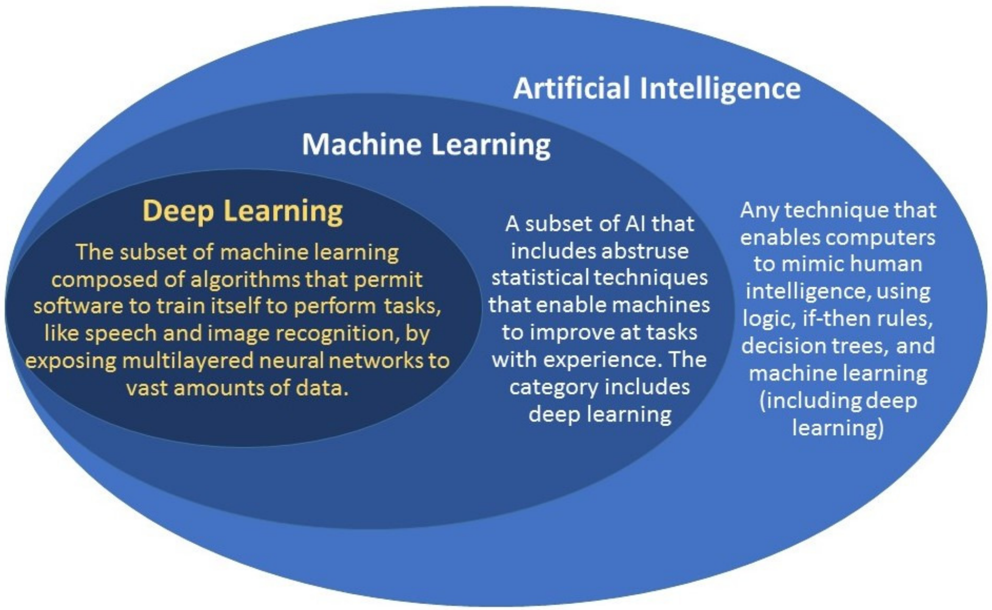


+ amazon web services +



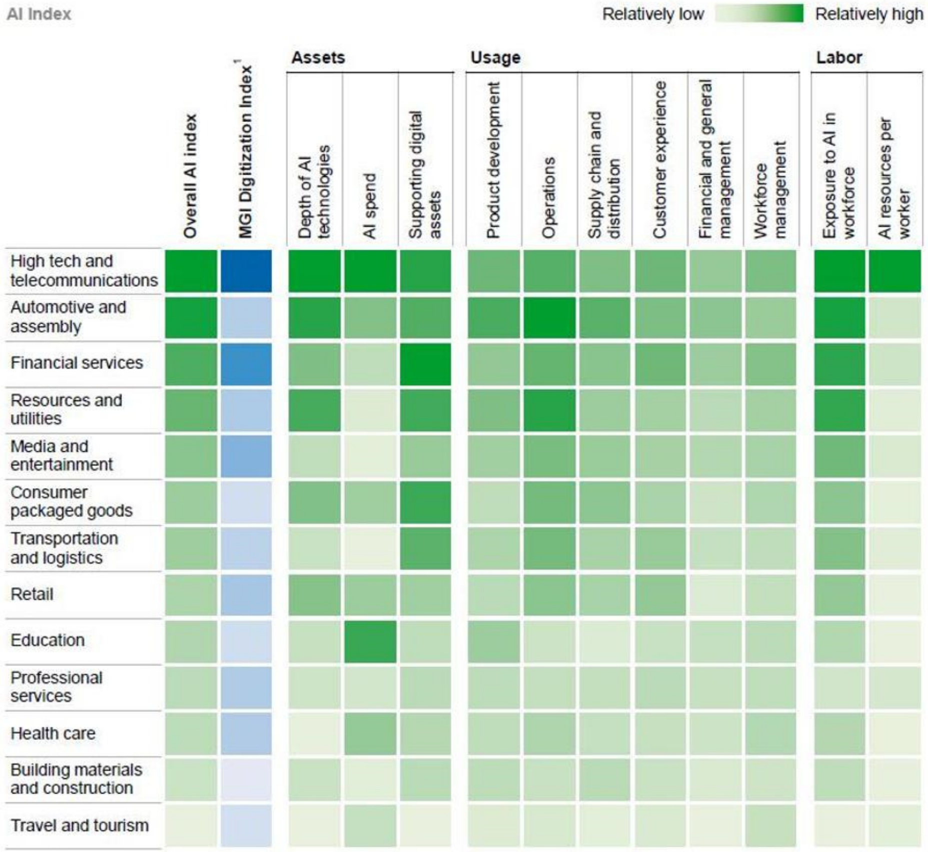
El Valor de los Datos

AI Project



Adopción AI

AI adoption is occurring faster in more digitized sectors and across the value chain

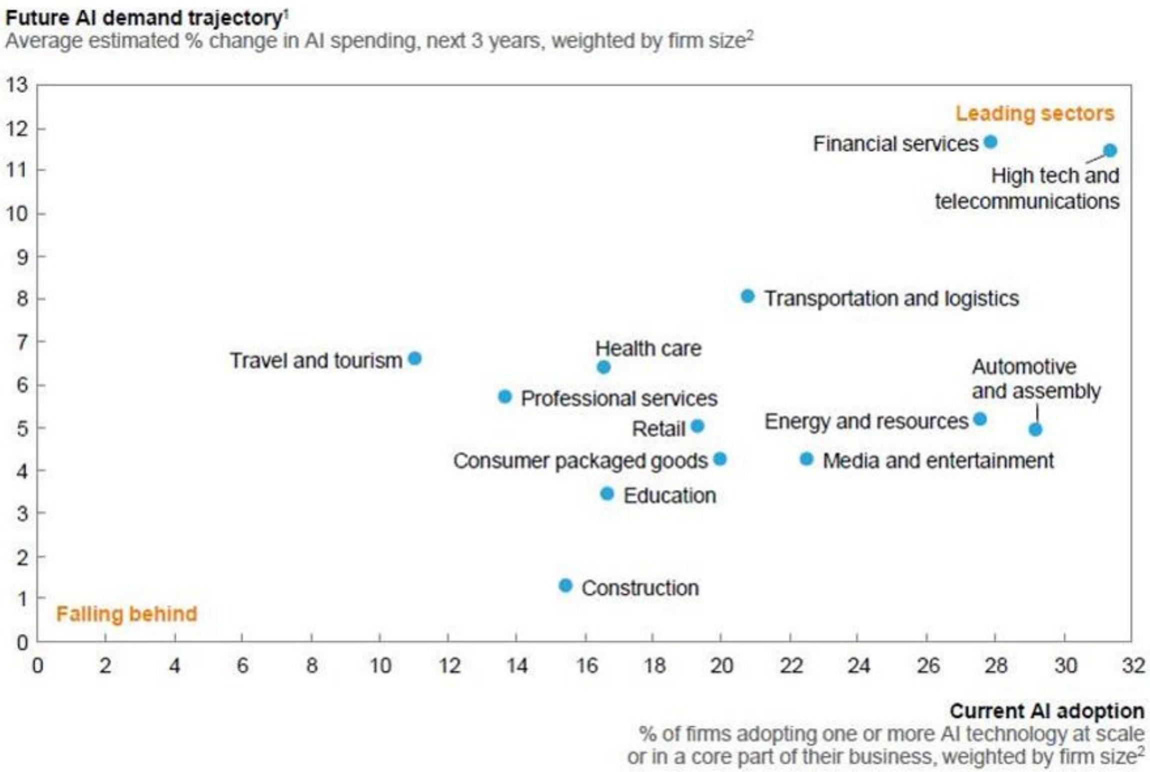


1 The MGI Digitization Index is GDP weighted average of Europe and United States. See Appendix B for full list of metrics and explanation of methodology.

SOURCE: McKinsey Global Institute AI adoption and use survey; *Digital Europe: Pushing the frontier, capturing the benefits*, McKinsey Global Institute, June 2016; *Digital America: A tale of the haves and have-mores*, McKinsey Global Institute, December 2015; McKinsey Global Institute analysis

Adopción AI por sectores

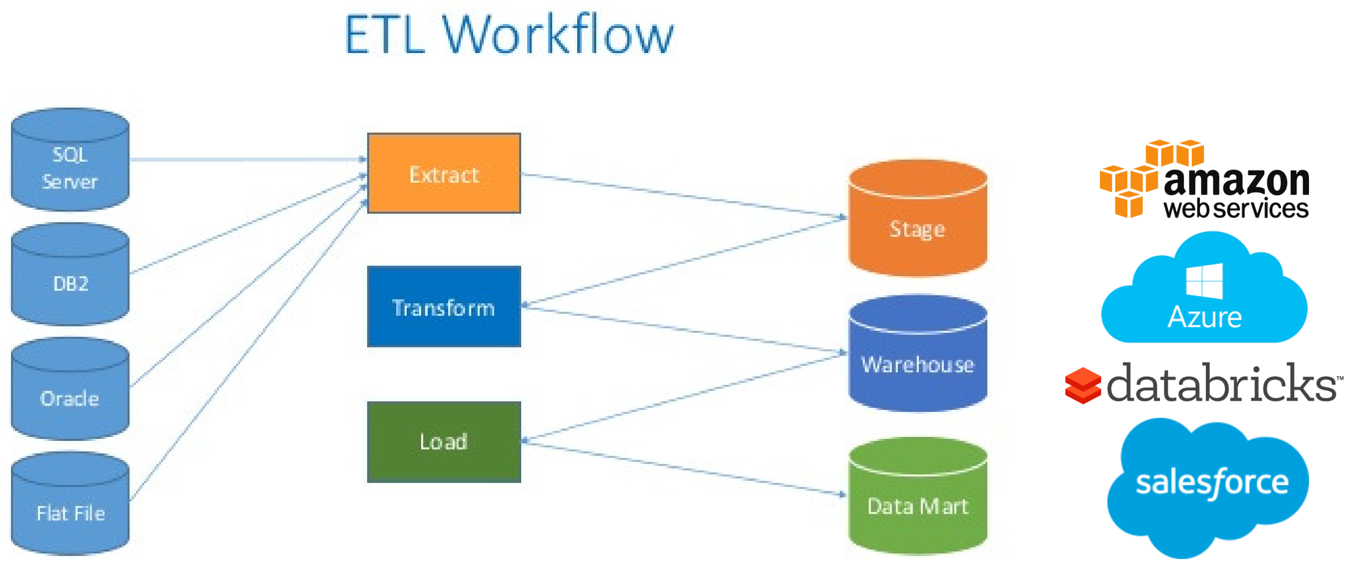
Sectors leading in AI adoption today also intend to grow their investment the most



1 Based on the midpoint of the range selected by the survey respondent.
2 Results are weighted by firm size. See Appendix B for an explanation of the weighting methodology.

SOURCE: McKinsey Global Institute AI adoption and use survey; McKinsey Global Institute analysis

Fases de los Datos



Extract / Expand

El objetivo de este proceso es la agregación de los datos de entrada en un almacén común o almacenamiento intermedio. Normalmente no se realizan transformaciones.

Formatos para este almacenamiento intermedio

- JSON: No almacena datos sobre los tipos (excepto Strings que se encierran entre "").
- CSV: Dependiendo de cómo se van a consumir los datos en fases sucesivas. Tampoco almacena datos sobre tipos.
- PARQUET: Formato columnar de almacenamiento en proyectos basados en arquitecturas Hadoop.

Extract / Expand

PROBLEMAS

- Librerias de acceso a las distintas fuentes (principalmente Bases de Datos).
- Desbordamiento de Memoria.

Extract / Expand

PROBLEMAS

- Librerias de acceso a las distintas fuentes (sobretodo Bases de Datos)

```
import sqlalchemy
Import pyhive, hive
## Motor SQLAlchemy para la base de datos SQL Server
axEngine = sqlalchemy.create_engine(dbc.ax['dialect'] + '+' + dbc.ax['driver'] + '://' +
dbc.ax['username'] + ':' + dbc.ax['password'] + '@' + dbc.ax['host'] + '/' + dbc.ax['database'] +
dbc.ax['parameters’])
```

pandas.read_csv

pandas.**read_csv**(*filepath_or_buffer, sep=', ', delimiter=None, header='infer', names=None, index_col=None, usecols=None, squeeze=False, prefix=None, mangle_dupe_cols=True, dtype=None, engine=None, converters=None, true_values=None, false_values=None, skipinitialspace=False, skiprows=None, nrows=None, na_values=None, keep_default_na=True, na_filter=True, verbose=False, skip_blank_lines=True, parse_dates=False, infer_datetime_format=False, keep_date_col=False, date_parser=None, dayfirst=False, iterator=False, chunksize=None, compression='infer', thousands=None, decimal=b'.', lineterminator=None, quotechar='"', quoting=0, escapechar=None, comment=None, encoding=None, dialect=None, tupleize_cols=None, error_bad_lines=True, warn_bad_lines=True, skipfooter=0, skip_footer=0, doublequote=True, delim_whitespace=False, as_recarray=None, compact_ints=None, use_unsigned=None, low_memory=True, buffer_lines=None, memory_map=False, float_precision=None*) [source]

Read CSV (comma-separated) file into DataFrame

Also supports optionally iterating or breaking of the file into chunks.

pandas.DataFrame.to_csv

DataFrame.**to_csv**(*path_or_buf=None, sep=', ', na_rep='', float_format=None, columns=None, header=True, index=True, index_label=None, mode='w', encoding=None, compression=None, quoting=None, quotechar='"', line_terminator='\n', chunksize=None, tupleize_cols=None, date_format=None, doublequote=True, escapechar=None, decimal='.'*) [source]

Write DataFrame to a comma-separated values (csv) file

pandas.read_sql

pandas.**read_sql**(*sql, con, index_col=None, coerce_float=True, params=None, parse_dates=None, columns=None, chunksize=None*) [source]

Read SQL query or database table into a DataFrame.

pandas.read_sql_table

pandas.**read_sql_table**(*table_name, con, schema=None, index_col=None, coerce_float=True, parse_dates=None, columns=None, chunksize=None*) [source]

Read SQL database table into a DataFrame.

Given a table name and a SQLAlchemy connectable, returns a DataFrame. This function does not support DBAPI connections.

pandas.read_json

pandas.**read_json**(*path_or_buf=None, orient=None, typ='frame', dtype=True, convert_axes=True, convert_dates=True, keep_default_dates=True, numpy=False, precise_float=False, date_unit=None, encoding=None, lines=False, chunksize=None, compression='infer'*) [source]

Convert a JSON string to pandas object

pandas.read_parquet

pandas.**read_parquet**(*path, engine='auto', **kwargs*) [source]

Load a parquet object from the file path, returning a DataFrame.

pandas.DataFrame.to_parquet

DataFrame.**to_parquet**(*fname, engine='auto', compression='snappy', **kwargs*) [source]

Write a DataFrame to the binary parquet format.

New in version 0.21.0.

Extract / Expand

PROBLEMAS

- **Desbordamiento de Memoria**

```
csv_buffer = StringIO()
gz_buffer = BytesIO()
axConnection = axEngine.connect()
rowsPending = True
while rowsPending:
   sqlSentence = "WITH Results_SQL AS (SELECT " + "ROW_NUMBER() OVER " + "(ORDER BY " + orderBy + ") as\
               RowNum, * "  + "FROM " + entityName + ") " + "SELECT * FROM Results_SQL WHERE  RowNum > “\
               + str(offset) + " AND RowNum <=  " + str(offset + chunkSize)
   chunk = pd.io.sql.read_sql(sqlSentence, axConnection)
   if (offset == 0):
      chunk.to_csv(csv_buffer, sep = ';',compression = 'gzip',encoding = 'utf8')
   else:
      chunk.to_csv(csv_buffer, sep = ';',compression = 'gzip',encoding = 'utf8',mode= 'a',header = False)
   offset += chunkSize
   if (len(chunk) < chunkSize):
      rowsPending = False
   del chunk
   csv_buffer.seek(0)
   with gzip.GzipFile(mode='w', fileobj=gz_buffer) as gz_file:
      gz_file.write(bytes(csv_buffer.getvalue(), 'utf-8’))
s3_resource = boto3.resource("s3")
s3_resource.Object(ENTITY_BUCKET, ENTITY_KEY).put(Body=gz_buffer.getvalue())
```

Transform

Es el proceso que más tiempo consume. Incluye muchos tipos de operaciones y transformaciones de acuerdo al negocio del Usuario.

pandas.Series.map

`Series.map`(*arg, na_action=None*) [source]

Map values of Series using input correspondence (which can be a dict, Series, or function)

Parameters:	**arg** : *function, dict, or Series* **na_action** : *{None, 'ignore'}* If 'ignore', propagate NA values, without passing them to the mapping function
Returns:	**y** : *Series* same index as caller

pandas.Series.apply

`Series.apply`(*func, convert_dtype=True, args=(), **kwds*) [source]

Invoke function on values of Series. Can be ufunc (a NumPy function that applies to the entire Series) or a Python function that only works on single values

Parameters:	**func** : *function* **convert_dtype** : *boolean, default True* Try to find better dtype for elementwise function results. If False, leave as dtype=object **args** : *tuple* Positional arguments to pass to function in addition to the value **Additional keyword arguments will be passed as keywords to the function**
Returns:	**y** : *Series or DataFrame if func returns a Series*

pandas.DataFrame.apply

`DataFrame.apply`(*func, axis=0, broadcast=False, raw=False, reduce=None, args=(), **kwds*) [source]

Applies function along input axis of DataFrame.

Objects passed to functions are Series objects having index either the DataFrame's index (axis=0) or the columns (axis=1). Return type depends on whether passed function aggregates, or the reduce argument if the DataFrame is empty.

Parameters:	**func** : *function* Function to apply to each column/row **axis** : *{0 or 'index', 1 or 'columns'}, default 0* • 0 or 'index': apply function to each column • 1 or 'columns': apply function to each row **broadcast** : *boolean, default False* For aggregation functions, return object of same size with values propagated **raw** : *boolean, default False* If False, convert each row or column into a Series. If raw=True the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance **reduce** : *boolean or None, default None* Try to apply reduction procedures. If the DataFrame is empty, apply will use reduce to determine whether the result should be a Series or a DataFrame. If reduce is None (the default), apply's return value will be guessed by calling func an empty Series (note: while guessing, exceptions raised by func will be ignored). If reduce is True a Series will always be returned, and if False a DataFrame will always be returned. **args** : *tuple* Positional arguments to pass to function in addition to the array/series

pandas.DataFrame.applymap

`DataFrame.applymap`(*func*) [source]

Apply a function to a DataFrame that is intended to operate elementwise, i.e. like doing map(func, series) for each series in the DataFrame

Parameters:	**func** : *function* Python function, returns a single value from a single value
Returns:	**applied** : *DataFrame*

Transform

PROBLEMAS

- Consumo excesivo de tiempo en mapeos.
- Desbordamiento de Memoria en transformaciones.

Transformaciones I

Transformaciones sencillas

```
# 7º column
bonus['LastUpdate'] = iebonus_parse['modifiedon'].map(lambda x: utc2Local(str(x)) if str(x) != 'nan' else np.nan)

# 8º column
bonus['AdmissionId'] = iebonus_parse['ie_admissionid']
```

Transformaciones II

Transformación complicada

```
# 12º column ( Fill it after building Historic)
def approved(x):
if x.AdmissionId in enrol_sales_dead:
   sales_dead_line = enrol_sales_dead[x.AdmissionId]
   times = []
   for item in histo_sales_dead[x.BonusId]:
      times.append(item)
      times = sorted(times)
      oldest = min(times)
      youngest = max(times)
      if sales_dead_line < oldest:
          date_to_see = oldest
      elif sales_dead_line > youngest:
          date_to_see = youngest
      else:
          for i in range(len(times) - 1):
            if sales_dead_line >= times[i] and sales_dead_line < times[i+1]:
                date_to_see = times[i]
                break
          bonus_date = x.BonusDate
          if bonus_date <= date_to_see:
              return 'Yes'
          else:
              return 'No'
       else:
         return np.nan
 bonus['ApprovedAtSalesDeadline'] = bonus.apply(lambda x: approved(x),axis=1)
```

Transformaciones IV

Transformación curiosa (SQL vs Python)

```
def calculate_experience(intervals):

    sorted_intervals = sorted(intervals, key=lambda tup: tup[0])
    merged = []
    total_experience = 0

    for higher in sorted_intervals:
        if not merged:
            merged.append(higher)
        else:
            lower = merged[-1]
            # test for intersection between lower and higher:
            # we know via sorting that lower[0] <= higher[0]
            if higher[0] <= lower[1]:
                upper_bound = max(lower[1], higher[1])
                merged[-1] = (lower[0], upper_bound)  # replace by merged interval
            else:
                merged.append(higher)
    for mer in merged:
        total_experience += mer[1] - mer[0]
    return round(float(total_experience/365.0),2)
```

Transformaciones

Desbordamiento de Memoria

En muchas transformaciones puede ser necesario convertir un parte de un Dataframe a otras estructuras de datos debido al desbordamiento de memoria en operaciones de “join”.

pandas.DataFrame.to_dict

`DataFrame.to_dict(orient='dict', into=<class 'dict'>)` [source]

Convert DataFrame to dictionary.

Parameters:

orient : *str {'dict', 'list', 'series', 'split', 'records', 'index'}*

Determines the type of the values of the dictionary.

- dict (default) : dict like {column -> {index -> value}}
- list : dict like {column -> [values]}
- series : dict like {column -> Series(values)}
- split : dict like {index -> [index], columns -> [columns], data -> [values]}
- records : list like [{column -> value}, ... , {column -> value}]
- index : dict like {index -> {column -> value}}

New in version 0.17.0.

Abbreviations are allowed. *s* indicates *series* and *sp* indicates *split*.

into : *class, default dict*

The collections.Mapping subclass used for all Mappings in the return value. Can be the actual class or an empty instance of the mapping type you want. If you want a collections.defaultdict, you must pass it initialized.

New in version 0.21.0.

Returns:

result : *collections.Mapping like {column -> {index -> value}}*

```
df = pd.DataFrame({'Click_Id':['A','B','C','D','E'],'Count':[100,200,300,400,250]})
df.set_index(‘Click_Id’)[‘Count’].to_dict(‘index’)
df.set_index(‘Count’)[‘Click_Id’].to_dict(‘index’)
```

Transformaciones

Ejemplo Diccionario

```
# Read CBD
cbd = read_file(TRANSFORM_BUCKET,CBD_KEY,cbd_col,1)
if cbd is None:
    return -1
cbd_dict = defaultdict()
for key in cbd.index:
    cbd_dict[cbd.loc[key,'SalesTeamId’]] = (cbd.loc[key,'CommercialRegionId’],\
                                            cbd.loc[key,'CommercialRegion'],\
                                            cbd.loc[key,'CenterId’],\
                                            cbd.loc[key,'CenterName'],\
                                            cbd.loc[key,'CommercialBusinessUnitId’],\
                                            cbd.loc[key,'CommercialBusinessUnit'],\
                                            cbd.loc[key,'SalesTeam'])

del cbd
```

Ejemplo I

Tuplas, Listas

pandas.Series.tolist

`Series.tolist()` [source]

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

pandas.Index.tolist

`Index.tolist()` [source]

Return a list of the values.

These are each a scalar type, which is a Python scalar (for str, int, float) or a pandas scalar (for Timestamp/Timedelta/Interval/Period)

https://docs.python.org/3.6/tutorial/datastructures.html

Tuplas, Listas

- Ambas utilizan las mismas estructuras subyacentes de datos.
- Sus diferencias son:
  - Las Listas son Arrays dinámicos: son mutables y se pueden redimensionar.
  - Las Tuplas son Arrays estáticos: son inmutables y su contenido no se puede cambiar una vez creado.
- Las Tuplas se usan para describir propiedades múltiples propiedades de una cosa que no cambia y las Listas se pueden usar para almacenar colecciones de datos de objetos diversos.
- Ambas pueden almacenar diferentes tipos de datos, aunque ello produce algún "overhead" y reduce la optimización. El "overhead" se reduce con datos del mismo tipo.

Diccionarios y Conjuntos

- Se usan cuando los datos no tienen un orden. Los Diccionarios se basan en (clave,valor) y los Conjuntos (Set) sólo en clave.
- Cuando alcanzan su punto crítico se almacenamiento se redimensionan. El tamaño menor es 8 buckets. El redimensionamiento es de 4x hasta los 50,000 elementos, después es de 2x. La redimensión solo sucede con un "insert".
- Los Diccionarios y Conjuntos proven una fantástica vía para almacenar datos sique se pueden indexer por clave.

Arrays

NumPy

http://www.numpy.org/

Pandas

python + NumPy + R

Pandas = best of Python + numpy + R

Python	- Easy syntax - Good for prototyping ("...but slow") - Helpful community
Numpy	- Fast, memory-efficient calcs - Well-tested algorithms
R	- DataFrame column labels - Indexes to align rows

https://pandas.pydata.org/

Tunning: Magic commands

- Jupyter tiene “Magic” commands que suministran funcionalidad adicional sobre el Código Python.
- “Magic” commands comienzan con % (para ejecuciones sobre una línea) o %% (para ejecuciones sobre la celda entera)
- conda install -c anaconda line_profiler
- conda install -c chroxvi memory_profiler

Arrays. Ejemplo II

Tunning. Ejemplo III

jupyter

Tunning. Ejemplo III

Iteración sobre las filas, o qué no hacer

- Pandas está construido sobre NumPy, diseñado para manipulación de vectores (los bucles son ineficientes)

- El método de Pandas iterrows suministra una tupla (Index, Series) sobre las que iterar, pero es muy lento.

Tunning. Ejemplo III

Mejor usando “apply”

- apply aplica una función a lo largo de un eje específico (filas o columnas)

- Más eficiente que iterrows, pero sigue requiriendo un bucle sobre las filas

- Se debe usar cuando no hay forma de vectorizer una función.

Tunning. Ejemplo III

Mejora de 2.24x

- Sigue haciendo muchas tareas repetitivas
- %lprun -f haversine df.apply(lambda row: haversine(40.671,\
  -73.985, row['latitude'], row['longitude']), axis=1)

```
Total time: 0.0337621 s
File: <ipython-input-2-d2b5c58ef814>
Function: haversine at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           def haversine(lat1, lon1, lat2, lon2):
     3      1631         2747      1.7      3.5      MILES = 3959
     4      1631        24089     14.8     30.4      lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
     5      1631         3132      1.9      4.0      dlat = lat2 - lat1
     6      1631         2145      1.3      2.7      dlon = lon2 - lon1
     7      1631        29595     18.1     37.4      a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
     8      1631        12244      7.5     15.5      c = 2 * np.arcsin(np.sqrt(a))
     9      1631         3220      2.0      4.1      total_miles = MILES * c
    10      1631         1958      1.2      2.5      return total_miles
```

Tunning. Ejemplo III

Vectorización

- Las unidades básicas de Pandas son:
  - **Series** es un array unidimensional con etiqueta de eje
  - **DataFrame** es un array bi-dimensional con etiquetas de ejes (filas y columnas)
- Vectorization es el proceso de realizar operaciones sobre arrays en vez de sobre escalares

Tunning. Ejemplo III

Mejora de 93.67x respecto a iterrows y 42.15x respecto a apply

- %lprun -f haversine haversine(40.671, -73.985,\
  df['latitude'], df['longitude’])
- Esta función no hace bucles

```
Total time: 0.00557439 s
File: <ipython-input-2-d2b5c58ef814>
Function: haversine at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           def haversine(lat1, lon1, lat2, lon2):
     3         1            7      7.0      0.1      MILES = 3959
     4         1         1160   1160.0      8.9      lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
     5         1         1285   1285.0      9.8      dlat = lat2 - lat1
     6         1          955    955.0      7.3      dlon = lon2 - lon1
     7         1         7766   7766.0     59.4      a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
     8         1         1371   1371.0     10.5      c = 2 * np.arcsin(np.sqrt(a))
     9         1          519    519.0      4.0      total_miles = MILES * c
    10         1            2      2.0      0.0      return total_miles
```

Tunning. Ejemplo III

NumPy

- NumPy es un paquete fundamental para computación científica en Python
- Las operaciones de NumPy operations se ejecutan en Código C pre-compilado y optimizado
- Suprime todo el overhead en el que incurre Pandas series (indexación, chequeo de tipo de datos, etc)

Tunning. Ejemplo III

Mejora de 550.1x respecto a iterrows y 5.8x respecto a vectorizacion Pandas

- %lprun -f haversine df["distance"] = haversine(40.671, -73.985,\
  df["latitude"].values, df["longitude"].values)
- Esta función no hace bucles

```
Total time: 0.00125355 s
File: <ipython-input-2-d2b5c58ef814>
Function: haversine at line 2

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     2                                           def haversine(lat1, lon1, lat2, lon2):
     3         1            9      9.0      0.3      MILES = 3959
     4         1          721    721.0     24.5      lat1, lon1, lat2, lon2 = map(np.deg2rad, [lat1, lon1, lat2, lon2])
     5         1           60     60.0      2.0      dlat = lat2 - lat1
     6         1           19     19.0      0.6      dlon = lon2 - lon1
     7         1         1884   1884.0     64.1      a = np.sin(dlat/2)**2 + np.cos(lat1) * np.cos(lat2) * np.sin(dlon/2)**2
     8         1          225    225.0      7.7      c = 2 * np.arcsin(np.sqrt(a))
     9         1           17     17.0      0.6      total_miles = MILES * c
    10         1            3      3.0      0.1      return total_miles
```

Pandas Cython

Enhancing Performance

Cython (Writing C extensions for pandas)

For many use cases writing pandas in pure python and numpy is sufficient. In some computationally heavy applications however, it can be possible to achieve sizeable speed-ups by offloading work to cython.

This tutorial assumes you have refactored as much as possible in python, for example trying to remove for loops and making use of numpy vectorization, it's always worth optimising in python first.

This tutorial walks through a "typical" process of cythonizing a slow computation. We use an example from the cython documentation but in the context of pandas. Our final cythonized solution is around 100 times faster than the pure python.

http://pandas.pydata.org/pandas-docs/stable/enhancingperf.html

Pandas: El Zen de la optimización

- Evita los bucles
- Si tienes que usar bucles, usa apply, no funciones iterables
- Si tienes que usar apply, usa Cython
- Vectorización es usualmente major que operaciones escalares
- Operaciones de vectores sobre NumPy arrays son más eficientes que las nativas de Pandas series

Pandas: conclusiones

- Pandas es muy potente
- Muchas maneras de hacer las cosas bien (y de hacerlas mal)
- Comprobar como se ejecuta en Jupyter (% y %%)
- Intentar escalar a problemas mayores
- Leer su código