pyreports

Release 1.2.1

Matteo Guadrini

Oct 05, 2021

CONTENTS:

1 Report workflow **3**

2 Features **5**
- 2.1 Installation . . . 5
- 2.2 Managers . . . 6
- 2.3 Executors . . . 9
- 2.4 Reports . . . 13
- 2.5 Data tools . . . 17
- 2.6 pyreports example . . . 20
- 2.7 pyreports package . . . 26
- 2.8 io . . . 37
- 2.9 core . . . 42

3 Indices and tables **47**

Python Module Index **49**

Index **51**

pyreports is a python library that allows you to create complex reports from various sources such as databases, text files, ldap, etc. and perform processing, filters, counters, etc. and then export or write them in various formats or in databases.

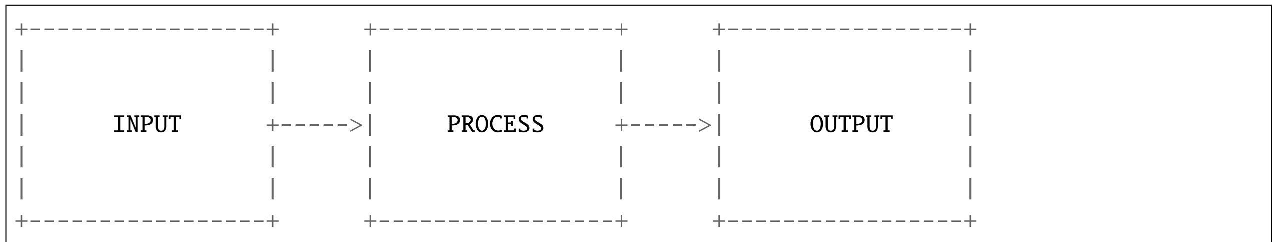
You can use this library for complex reports, or to simply filter data into datasets divided by topic. Furthermore, it is possible to export in various formats, such as csv, excel files or write directly to the database (*mysql*, *mssql*, *postgresql* and more).

CHAPTER

ONE

REPORT WORKFLOW

This package provides tools for receiving, processing and exporting data. Mostly, it follows this workflow.



CHAPTER

TWO

FEATURES

- Capture any type of data
- Export data in many formats
- Data analysis
- Process data with filters and maps
- Some functions will help you to process averages, percentages and much more

2.1 Installation

Here are the installation instructions

2.1.1 Requirements

pyreports is written in python3 (3.6 and higher).

Here are all the external libraries necessary for the proper functioning of the library:

- tablib
- ldap3
- pymssql
- mysql-connector-python
- psycopg2-binary

2.1.2 Installation

```
$ pip install --user pyreports
```

2.2 Managers

The manager objects are responsible for managing inputs or outputs. We can have three macro types of managers: *database*, *file* and *ldap*.

2.2.1 Type of managers

Each type of manager is managed by micro types; Below is the complete list:

1. **Database**
    1. sqllite (SQLlite)
    2. mssql (Microsoft SQL)
    3. mysql (MySQL or MariaDB)
    4. postgresql (PostgreSQL or EnterpriseDB)
2. **File**
    1. file (standard text file or log)
    2. csv (Comma Separated Value file)
    3. json (JSON file)
    4. yaml (YAML file)
    5. xlsx (Microsoft Excel file)
3. **LDAP**
    1. ldap (Active Directory Server, OpenLDAP, FreeIPA, etc.)

Note: The connection arguments of a `DatabaseManager` vary according to the type of database being accessed. Look at the manuals and documentation of each type of database to find out more.

```
import pyreports

# DatabaseManager object
sqllite_db = pyreports.manager('sqllite', database='/tmp/mydb.db')
mssql_db = pyreports.manager('mssql', server='mssql1.local', database='test', user='dba',
↪ password='dba0000')
mysql_db = pyreports.manager('mysql', host='mysql1.local', database='test', user='dba',␣
↪password='dba0000')
postgresql_db = pyreports.manager('postgresql', host='postgresql1.local', database='test
↪', user='dba', password='dba0000')

# FileManager object
file = pyreports.manager('file', '/tmp/log.log')
csv = pyreports.manager('csv', '/tmp/csv.csv')
json = pyreports.manager('json', '/tmp/json.json')
yaml = pyreports.manager('yaml', '/tmp/yaml.yml')
xlsx = pyreports.manager('xlsx', '/tmp/xlsx.xlsx')
```

(continues on next page)

(continued from previous page)

```
# LdapManager object
ldap = pyreports.manager('ldap', server='ldap.local', username='user', password='password
↪', ssl=False, tls=True)
```

2.2.2 Managers at work

A Manager object corresponds to each type of manager. And each Manager object has its own methods for writing and reading data.

DatabaseManager

Databasemanager have eight methods that are used to reconnect, query, commit changes and much more. Let's see these methods in action below.

Note: The following example will be done on a *mysql* type database, but it can be applied to any database because DB-API 2.0 is used.

```
import pyreports

# DatabaseManager object
mysql_db = pyreports.manager('mysql', host='mysql1.local', database='test', user='dba',␣
↪password='dba0000')

# Reconnect to database
mysql_db.reconnect()

# Query: CREATE
mysql_db.execute("CREATE TABLE cars(id SERIAL PRIMARY KEY, name VARCHAR(255), price INT)
↪")
# Query: INSERT
mysql_db.execute("INSERT INTO cars(name, price) VALUES('Audi', 52642)")
# Query: INSERT (many)
new_cars = [
  ('Alfa Romeo', 42123),
  ('Aston Martin', 78324),
  ('Ferrari', 129782),
]
mysql_db.executemany("INSERT INTO cars(name, price) VALUES(%s, %s)", new_cars)
# Commit changes
mysql_db.commit()
# Query: SELECT
mysql_db.execute('SELECT * FROM cars')
# View description and other info of last query
print(mysql_db.description, mysql_db.lastrowid, mysql_db.rowcount)
# Fetch all data
print(mysql_db.fetchall())                    # Dataset object
# Fetch first row
print(mysql_db.fetchone())                    # Dataset object
# Fetch select N row
```

(continues on next page)

(continued from previous page)

```
print(mysql_db.fetchmany(2))                # Dataset object
print(mysql_db.fetchmany())                 # This is same fetchone() method
# Query: SHOW
mysql_db.execute("SHOW TABLES")
print(mysql_db.fetchall())                  # Dataset object

# Call store procedure
mysql_db.callproc('select_cars')
mysql_db.callproc('select_cars', ['Audi'])  # Call with args
print(mysql_db.fetchall())                  # Dataset object
```

Note: Whatever operation is done, the return value of the `fetch*` methods return Dataset objects.

FileManager

FileManager has two simple methods: *read* and *write*. Let's see how to use this manager.

```
import pyreports

# FileManager object
csv = pyreports.manager('csv', '/tmp/cars.csv')

# Read data
cars = csv.read()       # Dataset object

# Write data
cars.append(['Audi', 52642])
csv.write(cars)
```

LdapManager

LdapManager is an object that allows you to interface and get data from a directory server via the ldap protocol.

```
import pyreports

# LdapManager object
ldap = pyreports.manager('ldap', server='ldap.local', username='user', password='password
↪', ssl=False, tls=True)

# Rebind connection
ldap.rebind()

# Query: get data
# This is Dataset object
users = ldap.query('DC=test,DC=local', '(&(objectClass=user)(objectCategory=person))', [
↪'name', 'mail', 'phone'])
if users:
    print(users)
```

(continues on next page)

(continued from previous page)

```
# Close connection
ldap.unbind()
```

Warning: *LdapManager* should only be used for inputs. An ldap manager has no write methods.

2.3 Executors

The **Executor** object is the one who analyzes and processes the data that is instantiated with it. This type of object is the first core object we will see and the basis for all the others.

2.3.1 Executor at work

To instantiate an *Executor* object, you need two things: the mandatory one, a **Dataset** and the other optional is a **header**, which represents the data. Let's see how to instantiate an *Executor* object.

```
import pyreports

# Create a data source
mydb = pyreports.manager('mysql', host='mysql1.local', database='test', user='dba',␣
↪password='dba0000')

# Get data
mydb.execute('SELECT * FROM salary')
employees = mydb.fetchall()                 # return Dataset object

# Create Executor object
myex = pyreports.Executor(employees)        # The employees object already has a header,␣
↪as it was created by a database manager
```

Note: If I wanted to apply a header different from the name of the table columns, perhaps because they are not very speaking or full of underscores, I would have to instantiate the object as follows: `myex = pyreports.Executor(employees, header=['name', 'surname', 'salary'])`. If you wanted to remove the header instead, just set it as `None`: `myex = pyreports.Executor(employees, header=None)`

The *Executor* is a flexible object. It is not related to the *pyreports* library. An *Executor* can also be instantiated via its own Dataset or from a list of tuples (Python primitives used to instantiate a Dataset object. It is also equal to the return value of a database object)

```
import pyreports
import tablib

# Create my Dataset object
mydata = tablib.Dataset()
mydata.append(['Arthur', 'Dent', 55000])
mydata.append(['Ford', 'Prefect', 65000])
```

(continues on next page)

(continued from previous page)

```
# Create Executor object: same result for both
myex = pyreports.Executor(mydata, header=['name', 'surname', 'salary'])
myex = pyreports.Executor([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪header=['name', 'surname', 'salary'])

# Set header after creation
myex.headers(['name', 'surname', 'salary'])
```

Filter data

One of the main functions of working with data is to filter it. The *Executor* object has a filter method for doing this. This method accepts a list of values that must correspond to one of the values of a row in the Executor's Dataset.

Another way to filter the data of an *Executor* object is to pass a callable that takes a single argument and returns something. The return value will be called by the `bool` class to see if it is `True` or `False`. This callable will be called to every single value of the row of the Executor's Dataset.

Finally, it is possible to declare the name of a single return column, if not all columns are needed.

Note: You can pass both a list of values and a function to filter the data.

```
# Filter data by list
myex.filter([55000, 65000, 75000])                        # Filter data only for␣
↪specified salaries

# Filter data by callable
myex.filter(key=str.istitle)                              # Filter data only for␣
↪string contains Title case

def big_salary(salary):
    if not isinstance(salary, int):
        return False
    return True if salary >= 65000 else False             # My custom function

myex.filter(key=big_salary)                               # Filter data with a salary␣
↪greater than or equal to 65000

# Filter data by column
myex.filter(column='salary')                              # Filter by column: name
myex.filter(column=2)                                     # Filter by column: index

# Filter data by list, callable and column
myex.filter([55000, 65000, 75000], str.istitle, 'salary')   # Filter for all three␣
↪methods
```

Warning: If the filters are not applied, the result will be an empty Executor object. If you want to reapply a filter, you will have to reset the object, using the `reset()` method. See below.

Map (modify) data

The *Executor* object is provided with a method to modify the data in real time. The `map` method accepts a mandatory argument, i.e. a callable that accepts a single argument and an optional one that accepts the name of the column or the number of its index.

```
# Define my function for increase salary; isn't that amazing!
def salary_increase(salary):
    if isinstance(salary, int):
        if salary <= 65000:
            return salary + 10000
    return salary

# Let's go! Increase salary today!
myex.map(salary_increase)

# Now, return only salary columns
myex.map(salary_increase, column='salary')
```

Warning: If the function you are passing to the *map* method returns nothing, `None` will be substituted for the original value. If you are using special conditions make sure your function always returns to its original value.

Get data

An *Executor* is not a data object. It is an object that contains data for processing, filters and etc. Once an instance of an *Executor* object is created, the original data is saved so that it can be retrieved.

So there is a way to retrieve and print the current and original data.

```
# Get data
myex.get_data()                              # Return current Dataset object
myex.origin                                  # Return original Dataset object
print(myex.get_data())                       # Print Dataset with current data

# Assign result to variable
my_dataset = myex.get_data()                 # Return Dataset object

# Create a new executor
new_ex = pyreports.Executor(myex.get_data())  # New Executor object with current data
new_ex = myex.clone()                        # New Executor object with original data
```

Note: If you want to clone the original data contained in an Executor object, use the `clone` method.

It is possible through this object, to restore the data source after the modification or the applied filter.

```
# Restore data
myex.reset()                              # Reset data to origin
print(myex.get_data())
```

Attention: Once the object is reset, any changes made will be lost, unless the object has been cloned.

Work with columns

Since the *Executor* object is based on a Dataset object, it is possible to work not only with rows but also with columns. Let's see how to select a single column.

```
# Select column
myex.select_column(1)                # Select column by index number (surname)
myex.select_column('surname')        # Select column by name (surname)
```

We can also add columns as long as they are the same length as the others, otherwise, we will receive an `InvalidDimension` exception.

```
# Add column with values
myex.add_column('floor', [1, 2])

# Add column with function values
def stringify_salary(row):
    return f'$ {row[2]}'

myex.add_column('str_salary', stringify_salary)
```

Note: The function passed to the `add_column` method must have a single argument representing the row (the name *"row"* is a convention). You can use this argument to access data from other columns.

It is also possible to delete a column.

```
# Delete column
myex.del_column('floor')
```

Count

The *Executor* object contains data. You may need to count rows and columns. The object supports the protocol for counting through the built-in `len` function, which will return the current number of rows.

```
# Count columns
myex.count_columns()        # Return number of columns

# Count rows
myex.count_rows()           # Return number of rows
len(myex)                   # Return number of rows
```

Iteration

The *Executor* object supports the python iteration protocol (return of generator object). This means that you can use it in a for loop or in a list comprehension.

```
# For each row in Executor
for row in myex:
    print(row)

# List comprehension
my_list_of_rows = [row for row in myex]
```

2.4 Reports

The package it is provided with a **Report** object and a **ReportBook** object.

The *Report* object provides an interface for a complete workflow-based report (see *Report workflow*).

The *ReportBook* object, on the other hand, is a list of *Report* objects.

This will follow the workflows of each *Report* it contains, except for the output, which can be saved in a single Excel file.

2.4.1 Report at work

The *Report* object provides an interface to the entire workflow of a report: it accepts an input, processes the data and provides an output. To instantiate an object, you basically need three things:

- **input**: a *Dataset* object, mandatory.
- **filter**, **map function** or/and **column**: they are the same objects you would use in an *Executor* object, optional.
- **output**: a *FileManager* object, optional.

```
import pyreports
import tablib

# Instantiate a simple Report object
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
myrep = pyreports.Report(mydata)

# View report
myrep           # repr(myrep)
print(myrep)    # str(myrep)
```

Advanced Report instance

The *Report* object is very complex. Instantiating it as above makes little sense, because the result will be identical to the input dataset. This object enables a series of features for data processing.

```
import pyreports
import tablib

# Instantiate a Report object
salary55k = pyreports.manager('csv', '/tmp/salary55k.csv')
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
report_only_55k = pyreports.Report(mydata, filters=[55000], title='Report salary 55k',␣
↪output=salary55k)

# View report
myrep           # <Report object, title=Report salary 55k>
```

The example above, creates a *Report* object that filters input data only for employees with a salary of 55k. But we can also edit the data on-demand and then filter it, as follows in the next example.

Note: You can also pass a function to the `filters` argument, as for an *Executor* object.

```
import pyreports
import tablib

# My custom function for modifying salary data
def stringify_salary(salary):
    if isinstance(salary, int):
        return f'$ {salary}'
    else:
        return salary

# Instantiate a Report object
salary55k = pyreports.manager('csv', '/tmp/salary55k.csv')
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
report_only_55k = pyreports.Report(mydata,
                                    filters=['$ 55000'],
                                    map_func=stringify_salary,
                                    title='Report salary 55k',
                                    output=salary55k)

# View report
myrep           # <Report object, title=Report salary 55k>
```

Note: It is also possible to declare a counter of the processed lines by setting `count=True`. Moreover, as for an Executor object, you can specify a single return `column` using the column argument; ex. `column='surname'`.

Execute Report

Once a *Report* object has been instantiated, you can execute the filters and editing functions (map) set during the creation of the object.

```
# Apply filters and map function
report_only_55k.exec()

# Print result
print(report_only_55k)

# Adding count after creation
report_only_55k.count = True
report_only_55k.exec()
print(report_only_55k)
```

Warning: Once a filter or map function is applied, it will not be possible to go back. If you want to change filters after call the `exec` method, you need to re-instantiate the object.

Export

Once the `exec` method is called, and then once the data is processed, we can export the data based on the output set when instantiating the object.

Note: If the output has not been specified, calling the export method will print the data to stdout.

```
# Save report on /tmp/salary55k.csv
report_only_55k.export()

# Unset output
report_only_55k.output = None
report_only_55k.export()            # This print the data on stdout

# Set output
report_only_55k.output = salary55k
report_only_55k.export()            # Save report on /tmp/salary55k.csv
```

2.4.2 ReportBook at work

The *ReportBook* object is a collection (list) of *Report* objects. This basically allows you to collect multiple reports in a single container object. The main advantage is the ability to iterate over each *Report* and access its properties.

```
import pyreports
import tablib

# Instantiate the Report objects
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
```

(continues on next page)

(continued from previous page)

```
report_only_55k = pyreports.Report(mydata, filters=[55000], title='Report salary 55k')
report_only_65k = pyreports.Report(mydata, filters=[65000], title='Report salary 65k')

# Create a ReportBook
salary = pyreports.ReportBook([report_only_55k, report_only_65k])

# View ReportBook
salary          # repr(salary)
print(salary)   # str(salary)
```

Note: The ReportBook object supports the `title` property, as follows: `pyreports.ReportBook(title='My report book')`

Export reports

The *ReportBook* object has an `export` method. This method not only saves *Report* objects to its output, but first executes the `exec` method of each *Report* object it contains.

Warning: As for *Report* objects, even a *ReportBook* object once the export method has been called, it will need to be instantiated again if you want to reset the data to the source, before applying the filters and map functions.

```
# Export a ReportBook
salary.export()         # This run exec() and export() on each Report object

# Export each Report on one file Excel (xlsx)
salary.export('/tmp/salary_report.xlsx')
```

Add and remove report

Being a container, the *ReportBook* object can be used to add and remove *Report* object.

```
# Create an empty ReportBook
salary = pyreports.ReportBook(title='Salary report')

# Add a Report object
salary.add(report_only_55k)
salary.add(report_only_65k)

# Remove last Report object added
salary.remove()                     # Remove report_only_65k object
salary.remove(0)                    # Remove report_only_55k object, via index
```

Count reports

The *ReportBook* object supports the protocol for the built-in `len` function, to count the *Report* objects it contains.

```
# Count object
len(salary)
```

Iteration

The *ReportBook* object supports the python iteration protocol (return of generator object). This means that you can use it in a for loop or in a list comprehension.

```
# For each report in ReportBook
for report in salary:
    print(report)

# List comprehension
my_list_of_report = [report for report in salary]
```

Merge

ReportBook objects can be joined together, using the + operator.

```
# ReportBook
book1 = pyreports.ReportBook([report1, report2])
book2 = pyreports.ReportBook([report3, report4])
# Merge ReportBook
tot_book = book1 + book2
tot_book = book1.__add__(book2)

print(tot_book)

# ReportBook None
#   Report1
#   Report2
#   Report3
#   Report4
```

2.5 Data tools

The package comes with utility functions to work directly with *Datasets*. In this section we will see all these functions contained in the **datatools** module.

2.5.1 Average

average function calculates the average of the numbers within a column.

```
import pyreports

# Build a dataset
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],
↪headers=['name', 'surname', 'salary'])

# Calculate average
print(pyreports.average(mydata, 'salary'))  # Column by name
print(pyreports.average(mydata, 2))         # Column by index
```

Attention: All values in the column must be `float` or `int`, otherwise a `ReportDataError` exception will be raised.

2.5.2 Most common

The **most_common** function will return the value of a specific column that is most recurring.

```
import pyreports

# Build a dataset
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],
↪headers=['name', 'surname', 'salary'])
mydata.append(('Ford', 'Prefect', 65000))

# Get most common
print(pyreports.most_common(mydata, 'name'))  # Ford
```

2.5.3 Percentage

The **percentage** function will calculate the percentage based on a filter (Any) on the whole *Dataset*.

```
import pyreports

# Build a dataset
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],
↪headers=['name', 'surname', 'salary'])
mydata.append(('Ford', 'Prefect', 65000))

# Calculate percentage
print(pyreports.percentage(mydata, 65000))  # 66.66666666666666 (percent)
```

2.5.4 Counter

The **counter** function will return a Counter object, with inside it the count of each element of a specific column.

```
import pyreports

# Build a dataset
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
mydata.append(('Ford', 'Prefect', 65000))

# Create Counter object
print(pyreports.counter(mydata, 'name'))  # Counter({'Arthur': 1, 'Ford': 2})
```

2.5.5 Aggregate

The **aggregate** function aggregates multiple columns of some *Dataset* into a single *Dataset*.

> **Warning:** The number of elements in the columns must be the same. If you want to aggregate columns with a different number of elements, you need to specify the argument `fill_empty=True`. Otherwise, an `InvalidDimension` exception will be raised.

```
import pyreports

# Build a datasets
employee = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
places = tablib.Dataset([('London', 'Green palace', 1), ('Helsinky', 'Red palace', 2)],␣
↪headers=['city', 'place', 'floor'])

# Aggregate column for create a new Dataset
new_data = pyreports.aggregate(employee['name'], employee['surname'], employee['salary'],
↪ places['city'], places['place']))
print(new_data.headers)     # ['name', 'surname', 'salary', 'city', 'place']
```

2.5.6 Merge

The **merge** function combines multiple *Dataset* objects into one.

> **Warning:** The datasets must have the same number of columns otherwise an `InvalidDimension` exception will be raised.

```
import pyreports

# Build a datasets
employee1 = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
employee2 = tablib.Dataset([('Tricia', 'McMillian', 55000), ('Zaphod', 'Beeblebrox',␣
↪65000)], headers=['name', 'surname', 'salary'])
```

(continues on next page)

(continued from previous page)

```
# Merge two Dataset object into only one
employee = pyreports.merge(employee1, employee2)
print(len(employee))     # 4
```

2.5.7 Chunks

The **chunks** function divides a *Dataset* into pieces from *N* (`int`). This function returns a generator object.

```
import pyreports

# Build a datasets
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
mydata.append(*[('Tricia', 'McMillian', 55000), ('Zaphod', 'Beeblebrox', 65000)])

# Divide data into 2 chunks
new_data = pyreports.chunks(mydata, 2)      # Generator object
print(list(new_data))     # [[('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)], [('Tricia
↪', 'McMillian', 55000), ('Zaphod', 'Beeblebrox', 65000)]]
```

Note: If the division does not result zero, the last tuple of elements will be a smaller number.

2.6 pyreports example

Example scripts using `pyreports` module.

2.6.1 Basic usage

In this section you will find examples that represent the entire reporting workflow, relying on the **Manager* objects as input and output, and the *Executor* object for the process part.

Database to file

In this example, we extract the data from a mysql database, filter it by error code and finally export it to a csv.

```
import pyreports

# INPUT

# Select source: this is a DatabaseManager object
mydb = pyreports.manager('mysql', host='mysql1.local', database='login_users', user='dba
↪', password='dba0000')

# Get data
mydb.execute('SELECT * FROM site_login')
```

(continues on next page)

(continued from previous page)

```
site_login = mydb.fetchall()                    # return Dataset object

# PROCESS

# Filter data
error_login = pyreports.Executor(site_login)    # accept Dataset object
error_login.filter([400, 401, 403, 404, 500])

# OUTPUT

# Save report: this is a FileManager object
output = pyreports.manager('csv', '/home/report/error_login.csv')
output.write(error_login.get_data())
```

Note: A reflection on this example could be: "Why don't I apply the filter directly in the SQL syntax?" The answer is simple. The advantage of using an *Executor* object is that from general data I can filter or modify (*map* function or with my custom function) without affecting the original Dataset. So much so that I could do several different Executors, process them and then re-merge them into a single Executor, which would be difficult to do with SQL syntax.

File to Database

In this example I have a json file as input, received from a web server, I process it and write to the database.

```
import pyreports

# INPUT

# Return json from GET request on web server: this is a FileManager object
web_server_result = pyreports.manager('json', '/home/report/users.json')
# Get data
users = web_server_result.read()                            # return Dataset object

# PROCESS

# Filter data
user_int = pyreports.Executor(users)                        # accept Dataset object
user_int.filter(key=lambda record: if record == 'INTERNAL') # My filter is a function
user_ext = pyreports.Executor(users)
user_ext.filter(key=lambda record: if record == 'EXTERNAL')

# OUTPUT

# Save report: this is a DatabaseManager object
mydb = pyreports.manager('mysql', host='mysql1.local', database='users', user='dba',␣
↪password='dba0000')

# Write to database
mydb.executemany("INSERT INTO internal_users(name, surname, employeeType) VALUES(%s, %s,
↪%s)", list(user_int))
```

(continues on next page)

(continued from previous page)

```
mydb.executemany("INSERT INTO external_users(name, surname, employeeType) VALUES(%s, %s,
↪%s)", list(user_ext))
mydb.commit()
```

Combine inputs

In this example, we will take two different inputs, and combine them to export an excel file containing the data processing of the two sources.

```
import pyreports

# INPUT

# Config Unix application file: this is a FileManager object
config_file = pyreports.manager('yaml', '/home/myapp.yml')
# Console admin: this is a DatabaseManager object
mydb = pyreports.manager('mssql', server='mssql1.local', database='admins', user='sa',␣
↪password='sa0000')
# Get data
admin_app = config_file.read()                    # return Dataset object: three column␣
↪(name, shell, login)
mydb.execute('SELECT * FROM console_admins')
admins = mydb.fetchall()                          # return Dataset object: three column␣
↪(name, shell, login)

# PROCESS

# Filter data
all_console_admins = pyreports.Executor(admins) # accept Dataset object
all_console_admins.filter(config_file['shell']) # filter by shells

# OUTPUT

# Save report: this is a FileManager object
output = pyreports.manager('xlsx', '/home/report/all_admins.xlsx')
output.write(all_console_admins.get_data())
```

Simple report

In this example, we use a Report type object to create and filter the data through a function and save it in a csv file, printing the number of lines in total.

```
import pyreports

OFFICE_FILTER = 'Customer'

# Function: filter by office
def filter_by_office(value):
    if value == OFFICE_FILTER:
        return True
```

(continues on next page)

(continued from previous page)

```
# Connect to database
mydb = pyreports.manager('postgresql', host='pssql1.local', database='users', user='admin
↪', password='pwd0000')
mydb.execute('SELECT * FROM employees')
all_employees = mydb.fetchall()
# Output to csv
output = pyreports.manager('csv', f'/home/report/office_{OFFICE_FILTER}.csv')
# All customer employees: Report object
one_office = pyreports.Report(all_employees,
                              filters=filter_by_office,
                              title=f'All employees in {OFFICE_FILTER}',
                              count=True,
                              output=output)
# Run and save report
one_office.export()
print(one_office.count)     # Row count
```

2.6.2 Advanced usage

From here on, the examples will be a bit more complex; we will process the data in order to modify it, filter it, combine it and merge it before exporting or parsing it in another object.

Report apache log

In this example we will analyze and capture parts of a web server log. For each error code present in the log, we will create a report that will be inserted in a book, where each sheet will contain the details of the error code. In the last sheet, there will be an element counter for every single error present in the report.

```
import pyreports
import tablib
import re

# Get apache log data: this is a FileManager object
apache_log = pyreports.manager('file', '/var/log/httpd/error.log').read()
# apache log format: regex
regex = '([(\d\.)]+) - - \[(.*?)\] "(.*?)" (\d+) - "(.*?)" "(.*?)"'

# Function than receive Dataset and return a new Dataset
def format_dataset_log(data_input):
    data = tablib.Dataset(headers=['ip', 'date', 'operation', 'code', 'client'])
    for row in data_input:
        log_parts = re.match(regex, row[0]).groups()
        new_row = list(log_parts[:4])
        new_row.append(log_parts[5])
        data.append(new_row)
    return data

# Create a collection of Report objects
all_apache_error = pyreports.ReportBook(title='Apache error on my site')
```

(continues on next page)

(continued from previous page)

```
# Create a Report object based on error code
apache_error_log = format_dataset_log(apache_log)
all_error = set(apache_error_log['code'])
for code in all_error:
    all_apache_error.add(pyreports.Report(apache_error_log, filters=[code], title=f
↪'Error {code}'))

# Count all error code
counter = pyreports.counter(apache_error_log, 'code')
# Append new Report on ReportBook with error code counters
error_counter = tablib.Dataset(counter.values(), headers=counter)
all_apache_error.add(pyreports.Report(error_counter))

# Save ReportBook on Excel
all_apache_error.export('/home/report/apache_log_error_code.xlsx')
```

We now have a script that parses and breaks an apache httpd log file by error code.

Report e-commerce data

In this example, we combine data from different e-commerce databases. In addition, we will create two reports: one for the sales, the other for the warehouse. Then once saved, we will create an additional report that combines both of the previous ones.

```
import pyreports

# Get data from database: a DatabaseManager object
mydb = pyreports.manager('postgresql', host='pssql1.local', database='ecommerce', user=
↪'reader', password='pwd0000')
mydb.execute('SELECT * FROM sales')
sales = mydb.fetchall()
mydb.execute('SELECT * FROM warehouse')
warehouse = mydb.fetchall()

# filters
household = ['plates', 'glass', 'fork']
clothes = ['shorts', 'tshirt', 'socks']

# Create sales Report objects
sales_by_household= pyreports.Report(sales, filter=household, title='household sold items
↪')
sales_by_clothes = pyreports.Report(sales, filter=clothes, title='clothes sold items')

# Create warehouse Report objects
warehouse_by_household= pyreports.Report(warehouse, filter=household, title='household␣
↪items in warehouse')
warehouse_by_clothes = pyreports.Report(warehouse, filter=clothes, title='clothes items␣
↪in warehouse')

# Create a ReportBook objects
```

(continues on next page)

(continued from previous page)

```
sales_book = pyreports.ReportBook([sales_by_household, sales_by_clothes], filter='Total␣
↪sold')
warehouse_book = pyreports.ReportBook([warehouse_by_household, warehouse_by_clothes],␣
↪filter='Total remained')

# Save reports
sales_book.export('/home/report/sales.xlsx')
warehouse_book.export('/home/report/warehouse.xlsx')

# Other report: combine two book
all = sales_book + warehouse_book
all.export('/home/report/all.xlsx')

# Now print to stdout all data
all.export()
```

Command line report

In this example, we're going to create a script that doesn't save any files. We will read from a database, modify the data so that it is more readable and print it in standard output. We will also see how to use our script with other command line tools.

```
import pyreports

# Get data from database: a DatabaseManager object
mydb = pyreports.manager('sqllite', database='/var/myapp/myapp.db')
mydb.execute('SELECT * FROM performance')
performance = mydb.fetchall()

# Transform data for command line reader
cmd = pyreports.Executor(performance)

def number_to_second(seconds):
    if isinstance(seconds, int):
        ret = float(int)
        return f'{ret:.2f} s'
    else:
        return seconds

cmd.map(number_to_second)

# Print data
print(cmd.get_data())
```

Now we can read the db directly from the command line.

```
$ python performance.py
$ python performance.py | grep -G "12.*"
```

Note: The examples we can give are almost endless. This library has such flexible python objects that we can adapt

them to any use case. You can also use it as a simple database data reader.

2.6.3 Use cases

As you may have noticed, there are many use cases for this library. The `manager` objects are so flexible that you can read and write data from any source. Furthermore, thanks to the `Executor` objects you can filter and modify the data on-demand when you want and restore it at a later time, and then channel it into the `Report` objects and then into the `ReportBook` collection objects.

Below, I'll list other use cases common to both package users and developers:

- Export LDAP users and insert them into a database
- Read a log file and write it into a database
- Find out which LDAP users are present in a web server log file
- Backup configuration files by exporting them in yaml format (passwd, httpd.conf, etc)
- Calculate access rates of a database
- Count how many times an ip address is present in a log file

I could go on indefinitely; anything you can think of about a file, a database and an LDAP server and you need to manipulate or verify the data, this is the library for you.

2.7 pyreports package

The package includes python modules for creating reports, from input to output to data processing.

2.7.1 pyreports modules

io

The *io* module contains all the classes and functions needed to interface with inputs and outputs.

Contains all input management.

class `pyreports.io.`**`Connection`**(**args*, ***kwargs*)

Bases: `abc.ABC`

Connection base class

`__init__`(**args*, ***kwargs*)

Connection base object.

`__weakref__`

list of weak references to the object (if defined)

class `pyreports.io.`**`CsvFile`**(*filename*)

Bases: *`pyreports.io.File`*

CSV file class

`read`(***kwargs*)

Read csv format

Returns Dataset object

write(*data*)
Write data on csv file

Parameters **data** – data to write on csv file

Returns None

class pyreports.io.**DatabaseManager**(*connection:* pyreports.io.Connection)
Bases: object

Database manager class for SQL connection

__init__(*connection:* pyreports.io.Connection)
Database manager object for SQL connection

Parameters **connection** – Connection based object

__repr__()
Representation of DatabaseManager object

Returns string

__weakref__
list of weak references to the object (if defined)

callproc(*proc_name*, *params=None*)
Calls the stored procedure named

Parameters

- **proc_name** – name of store procedure
- **params** – sequence of parameters must contain one entry for each argument that the procedure expects

Returns Dataset object

commit()
This method sends a COMMIT statement to the server

Returns None

execute(*query*, *params=None*)
Execute query on database cursor

Parameters

- **query** – SQL query language
- **params** – parameters of the query

Returns None

executemany(*query*, *params*)
Execute query on database cursor with many parameters

Parameters

- **query** – SQL query language
- **params** – list of parameters of the query

Returns None

fetchall()
Fetches all (or all remaining) rows of a query result set

Returns Dataset object

fetchmany(*size=1*)
Fetches the next set of rows of a query result

Parameters **size** – the number of rows returned

Returns Dataset object

fetchone()
Retrieves the next row of a query result set

Returns Dataset object

reconnect()
Close and start connection

Returns None

class pyreports.io.**ExcelFile**(*filename*)
Bases: *pyreports.io.File*

Excel file class

read(***kwargs*)
Read xlsx format

Returns Dataset object

write(*data*)
Write data on xlsx file

Parameters **data** – data to write on yaml file

Returns None

class pyreports.io.**File**(*filename*)
Bases: abc.ABC

File base class

__init__(*filename*)
File base object

Parameters **filename** – file path

__weakref__
list of weak references to the object (if defined)

abstract read(***kwargs*)
Read with format

Returns Dataset object

abstract write(*data*)
Write data on file

Parameters **data** – data to write on file

Returns None

class pyreports.io.**FileManager**(*file: pyreports.io.File*)
Bases: object

File manager class for various readable file format

__init__(*file: pyreports.io.File*)
File manager object for various readable file format

Parameters **file** – file object

__repr__()

Representation of FileManager object

Returns string

__weakref__

list of weak references to the object (if defined)

read(***kwargs*)

Read file

Returns Dataset object

write(*data*)

Write data on file

Parameters **data** – data to write on file

Returns None

class pyreports.io.**JsonFile**(*filename*)

Bases: *pyreports.io.File*

JSON file class

read(***kwargs*)

Read json format

Returns Dataset object

write(*data*)

Write data on json file

Parameters **data** – data to write on json file

Returns None

class pyreports.io.**LdapManager**(*server*, *username*, *password*, *ssl=False*, *tls=True*)

Bases: object

LDAP manager class

__init__(*server*, *username*, *password*, *ssl=False*, *tls=True*)

LDAP manager object

Parameters

- **server** – fqdn server name or ip address
- **username** – username for bind operation
- **password** – password of the username used for bind operation
- **ssl** – disable or enable SSL. Default is False.
- **tls** – disable or enable TLS. Default is True.

__repr__()

Representation of LdapManager object

Returns string

__weakref__

list of weak references to the object (if defined)

query(*base_search*, *search_filter*, *attributes*)
Search LDAP element on subtree base search directory

Parameters

- **base_search** – distinguishedName of LDAP base search
- **search_filter** – LDAP query language
- **attributes** – list of returning LDAP attributes

Returns Dataset object

rebind(*username*, *password*)
Re-bind with specified username and password

Parameters

- **username** – username for bind operation
- **password** – password of the username used for bind operation

Returns None

unbind()
Unbind LDAP connection

Returns None

class pyreports.io.**MSSQLConnection**(**args*, ***kwargs*)
Bases: *pyreports.io.Connection*

Connection microsoft sql class

class pyreports.io.**MySQLConnection**(**args*, ***kwargs*)
Bases: *pyreports.io.Connection*

Connection mysql class

class pyreports.io.**PostgreSQLConnection**(**args*, ***kwargs*)
Bases: *pyreports.io.Connection*

Connection postgresql class

class pyreports.io.**SQLliteConnection**(**args*, ***kwargs*)
Bases: *pyreports.io.Connection*

Connection sqlite class

class pyreports.io.**TextFile**(*filename*)
Bases: *pyreports.io.File*

Text file class

read(***kwargs*)
Read with format

Returns Dataset object

write(*data*)
Write data on file

Parameters **data** – data to write on file

Returns None

class pyreports.io.**YamlFile**(*filename*)

Bases: *pyreports.io.File*

YAML file class

read(***kwargs*)

Read yaml format

Returns Dataset object

write(*data*)

Write data on yaml file

Parameters **data** – data to write on yaml file

Returns None

pyreports.io.**create_database_manager**(*dbtype*, **args*, ***kwargs*)

Creates a DatabaseManager object

Parameters **dbtype** – type of database connection

Returns DatabaseManager

pyreports.io.**create_file_manager**(*filetype*, *filename*)

Creates a FileManager object

Parameters

- **filetype** – type of file
- **filename** – path of file

Returns FileManager

pyreports.io.**create_ldap_manager**(*server*, *username*, *password*, *ssl=False*, *tls=True*)

Creates a LdapManager object

Parameters

- **server** – fqdn server name or ip address
- **username** – username for bind operation
- **password** – password of the username used for bind operation
- **ssl** – disable or enable SSL. Default is False.
- **tls** – disable or enable TLS. Default is True.

pyreports.io.**manager**(*datatype*, **args*, ***kwargs*)

Creates manager object based on datatype

Parameters

- **datatype** – type of manager
- **args** – various positional arguments
- **kwargs** – various keyword arguments

Returns Manager object

core

The *core* module contains all the classes that refer to the creation and manipulation of data.

Contains all business logic and data processing.

class pyreports.core.Executor(*data*, *header=None*)

Bases: `object`

Executor receives, processes, transforms and writes data

__init__(*data*, *header=None*)

Create Executor object

Parameters

- **data** – everything type of data
- **header** – list header of data

__iter__()

Iterate over dataset

Returns next value

__len__()

Count data

Returns integer

__str__()

Pretty representation of Executor object

Returns string

__weakref__

list of weak references to the object (if defined)

add_column(*column*, *value*)

Add column to data

Parameters

- **column** – column name
- **value** – list value for column, or function with no arguments that returns a value

Returns None

clone()

Clone Executor object

Returns executor

count_columns()

Count all column

Returns integer

count_rows()

Count all rows

Returns integer

del_column(*column*)

Delete column

Parameters `column` – column name

Returns None

`filter`(*flist=None*, *key=None*, *column=None*)

Filter data through a list of strings (equal operator) and/or function key

Parameters

- `flist` – list of strings
- `key` – function that takes a single argument and returns data
- `column` – select column name or index number

Returns None

`get_data`()

Get dataset

Returns dataset

`headers`(*header*)

Set header

Parameters `header` – header of data

Returns None

`map`(*key*, *column=None*)

Apply function to data

Parameters

- `key` – function that takes a single argument
- `column` – select column name or index number

Returns None

`reset`()

Reset data to original data

Returns None

`select_column`(*column*)

Filter dataset by column

Parameters `column` – name or index of column

Returns Dataset object

`class pyreports.core.Report`(*input_data: tablib.core.Dataset*, *title=None*, *filters=None*, *map_func=None*, *column=None*, *count=False*, *output: Optional[*pyreports.io.FileManager*] = None*)

Bases: `object`

Report represents the workflow for generating a report

`__init__`(*input_data: tablib.core.Dataset*, *title=None*, *filters=None*, *map_func=None*, *column=None*, *count=False*, *output: Optional[*pyreports.io.FileManager*] = None*)

Create Report object

Parameters

- `input_data` – Dataset object
- `title` – title of Report object

- **filters** – list or function for filter data
- **map_func** – function for modifying data
- **column** – select column name or index
- **count** – count rows
- **output** – FileManager object

`__repr__()`
Representation of Report object

Returns string

`__str__()`
Pretty representation of Report object

Returns string

`__weakref__`
list of weak references to the object (if defined)

`exec()`
Create Executor object to apply filters and map function to input data

Returns None

`export()`
Process and save data on output

Returns if count is True, return row count

`send`(*server*, *_from*, *to*, *cc=None*, *bcc=None*, *subject=None*, *body=''*, *auth=None*, *_ssl=True*, *headers=None*)
Send saved report to email

Parameters

- **server** – server SMTP
- **_from** – email address 'from:'
- **to** – email address 'to:'
- **cc** – email address 'cc:'
- **bcc** – email address 'bcc:'
- **subject** – email subject. Default is report title
- **body** – email body
- **auth** – authorization tuple "(user, password)"
- **_ssl** – boolean, if True port is 465 else 25
- **headers** – more header value "(header_name, key, value)"

Returns None

class pyreports.core.**ReportBook**(*reports=None*, *title=None*)
Bases: `object`

ReportBook represent a collection of Report's object

`__add__`(*other*)
Add report object

Parameters **other** – Report object

Returns ReportBook

`__init__`(*reports=None*, *title=None*)
Create a ReportBook object

Parameters

- **reports** – Report's object list
- **title** – title of report book

`__iter__`()
Return report iterator

Returns iterable object

`__len__`()
Number of Report objects

Returns int

`__repr__`()
Representation of ReportBook object

Returns string

`__str__`()
Pretty representation of ReportBook object

Returns string

`__weakref__`
list of weak references to the object (if defined)

`add`(*report:* pyreports.core.Report)
Add report object

Parameters **report** – Report object

Returns None

`export`(*output=None*)
Save data on report output or an Excel workbook

Parameters **output** – output path for report export

Returns None

`remove`(*index: Optional[int] = None*)
Remove Report object, last added or index specified

Parameters **index** – report number to remove

Returns None

`send`(*server*, *_from*, *to*, *cc=None*, *bcc=None*, *subject=None*, *body=''*, *auth=None*, *_ssl=True*, *headers=None*)
Send saved report to email

Parameters

- **server** – server SMTP
- **_from** – email address 'from:'
- **to** – email address 'to:'
- **cc** – email address 'cc:'

- **bcc** – email address 'bcc:'
- **subject** – email subject. Default is report title
- **body** – email body
- **auth** – authorization tuple "(user, password)"
- **_ssl** – boolean, if True port is 465 else 25
- **headers** – more header value "(header_name, key, value)"

Returns None

datatools

The *datatools* module contains all utility functions for data processing.

Contains all functions for data processing.

`pyreports.datatools.`**`aggregate`**`(`**columns*, *fill_empty: bool = False*, *fill_value=None*`)`
Aggregate in a new Dataset the columns

Parameters

- **columns** – columns added
- **fill_empty** – fill the empty field of data with "fill_value" argument
- **fill_value** – fill value for empty field if "fill_empty" argument is specified

Returns Dataset

`pyreports.datatools.`**`average`**`(`*data*, *column*`)`
Average of list of integers or floats

Parameters

- **data** – Dataset object
- **column** – column name or index

Returns float

`pyreports.datatools.`**`chunks`**`(`*data*, *length*`)`
Yield successive n-sized chunks from data

Parameters

- **data** – Dataset object
- **length** – n-sized chunks

Returns generator

`pyreports.datatools.`**`counter`**`(`*data*, *column*`)`
Count all row value

Parameters

- **data** – Dataset object
- **column** – column name or index

Returns Counter

`pyreports.datatools.`**`merge`**`(`**datasets*`)`
Merge two or more dataset in only one

Parameters **datasets** – Dataset object collection

Returns Dataset

`pyreports.datatools.`**`most_common`**`(`*data*, *column*`)`

The most common element in a column

Parameters

- **data** – Dataset object
- **column** – column name or index

Returns Any

`pyreports.datatools.`**`percentage`**`(`*data*, *filter_*`)`

Calculating the percentage according to filter

Parameters

- **data** – Dataset object
- **filter** – filter

Returns float

exception

The *exception* module contains all the classes that represent explicit package exceptions.

Contains all custom exception.

exception `pyreports.exception.`**`ReportDataError`**

Bases: *pyreports.exception.ReportException*

exception `pyreports.exception.`**`ReportException`**

Bases: `Exception`

`__weakref__`

list of weak references to the object (if defined)

exception `pyreports.exception.`**`ReportManagerError`**

Bases: *pyreports.exception.ReportException*

2.8 io

In this section, you will find information on how to add new types of `*Connection` objects, `*File` objects, or `*Manager` objects.

2.8.1 Connection

Each `*Connection` object inherits from the abstract `Connection` class, which forces each type of connection object to accept these arguments when creating the object:

- `args`, various positional arguments
- `kwargs`, various keyword arguments

Besides this, the class must have a `connect` and a `close` method, respectively to connect to the database and one to close the connection, respectively.

```
class Connection(ABC):

    """Connection base class"""

    def __init__(self, *args, **kwargs):
        """Connection base object."""

        self.connection = None
        self.cursor = None
        self.args = args
        self.kwargs = kwargs

    @abstractmethod
    def connect(self):
        pass

    @abstractmethod
    def close(self):
        pass
```

Example `Connection` based class:

```
class SQLliteConnection(Connection):

    """Connection sqlite class"""

    def connect(self):
        self.connection = sqlite3.connect(*self.args, **self.kwargs)
        self.cursor = self.connection.cursor()

    def close(self):
        self.connection.close()
        self.cursor.close()
```

Warning: All connections are DBAPI 2.0 compliant. If you need to create your own, it must adhere to these APIs.

2.8.2 File

The `File` is the abstract class that the other `*File` classes are based on. It contains only the `file` attribute, where the path of the file is saved during the creation of the object and two methods: `read` to read the contents of the file (must return a Dataset object) and `write` (accept a Dataset) and writes to the destination file.

```
class File(ABC):

    """File base class"""

    def __init__(self, filename):
        """File base object

        :param filename: file path
```

(continues on next page)

(continued from previous page)

```
        """
        self.file = filename

    @abstractmethod
    def write(self, data):
        """Write data on file

        :param data: data to write on file
        :return: None
        """
        pass

    @abstractmethod
    def read(self, **kwargs):
        """Read with format

        :return: Dataset object
        """
        pass
```

Example `File` based class:

```
class CsvFile(File):

    """CSV file class"""

    def write(self, data):
        """Write data on csv file

        :param data: data to write on csv file
        :return: None
        """
        if not isinstance(data, tablib.Dataset):
            data = tablib.Dataset(data)
        with open(self.file, mode='w') as file:
            file.write(data.export('csv'))

    def read(self, **kwargs):
        """Read csv format

        :return: Dataset object
        """
        with open(self.file) as file:
            return tablib.Dataset().load(file, **kwargs)
```

2.8.3 Alias

When creating a `Connection` or `File` class, if you want to use the `manager` function to create the returning `*Manager` object, you need to create an alias. There are two dicts in the `io` module, which represent the aliases of these objects. If you have created a new `Connection` class, you will need to enter your alias in the `DBTYPE` *dict* while for File-type classes, enter it in the `FILETYPE` *dict*. Here is an example: `'ods': ODSFile`

2.8.4 Manager

Managers are classes that represent an input and output manager. For example, the `DatabaseManager` class accepts a `Connection` object and implements methods on these types of objects representing database connections.

class pyreports.io.DatabaseManager(*connection: pyreports.io.Connection*)

Database manager class for SQL connection

callproc(*proc_name*, *params=None*)

Calls the stored procedure named

Parameters

- **proc_name** – name of store procedure
- **params** – sequence of parameters must contain one entry for each argument that the procedure expects

Returns Dataset object

commit()

This method sends a COMMIT statement to the server

Returns None

execute(*query*, *params=None*)

Execute query on database cursor

Parameters

- **query** – SQL query language
- **params** – parameters of the query

Returns None

executemany(*query*, *params*)

Execute query on database cursor with many parameters

Parameters

- **query** – SQL query language
- **params** – list of parameters of the query

Returns None

fetchall()

Fetches all (or all remaining) rows of a query result set

Returns Dataset object

fetchmany(*size=1*)

Fetches the next set of rows of a query result

Parameters **size** – the number of rows returned

Returns Dataset object

`fetchone()`
Retrieves the next row of a query result set

Returns Dataset object

`reconnect()`
Close and start connection

Returns None

Manager function

Each `*Manager` class has associated a function of type `create_<type of manager>_manager(*args, **kwargs)`. This function will then be used by the `manager` function to create the corresponding `*Manager` object based on its alias.

For example, the `DatabaseManager` class has associated the `create_database_manager` function which will be called by the `manager` function to create the object based on the type of alias passed.

`pyreports.io.manager(datatype, *args, **kwargs)`
Creates manager object based on datatype

Parameters

- **datatype** – type of manager
- **args** – various positional arguments
- **kwargs** – various keyword arguments

Returns Manager object

```
def create_database_manager(dbtype, *args, **kwargs):
    """Creates a DatabaseManager object

    :param dbtype: type of database connection
    :return: DatabaseManager
    """
    # Create DatabaseManager object
    connection = DBTYPE[dbtype](*args, **kwargs)
    return DatabaseManager(connection=connection)
```

2.8.5 Example

Here we will see how to create your own `*Connection` class to access a specific database.

```
import pyreports
import DB2

# class for connect DB2 database
class DB2Connection(pyreports.io.Connection):

    def connect(self):
        self.connection = DB2.connect(*self.args, **self.kwargs)
        self.cursor = self.connection
```

(continues on next page)

(continued from previous page)

```
    def close(self):
        self.connection.close()
        self.cursor.close()

# Create an alias for DB2Connection object
pyreports.io.DBTYPE['db2'] = DB2Connection

# Create my DatabaseManager object
mydb2 = pyreports.manager(dsn='sample', uid='db2inst1', pwd='ibmdb2')
```

2.9 core

In this section, we will see how to expand and modify *pyreports core* objects.

2.9.1 Expand Executor

It is possible that in some particular case, it is necessary to have custom methods not included in the objects at our disposal. This concept extends to python in general, but we will focus on this library.

Custom map method

The `map` method of the `Executor` class accepts a function as an argument that it will call for each element of each row of the `Dataset` included in the `Executor` object.

```
    def map(self, key, column=None):
        """Apply function to data

        :param key: function that takes a single argument
        :param column: select column name or index number
        :return: None
        """
        if callable(key):
            ret_data = tablib.Dataset(headers=self.data.headers)
            for row in self:
                # Apply function to data
                new_row = list()
                for field in row:
                    new_row.append(key(field))
                ret_data.append(new_row)
            self.data = ret_data
        else:
            raise ValueError(f"{key} isn't function object")
        # Return all data or single column
        if column and self.data.headers:
            self.data = self.select_column(column)
```

There may be a need to apply the function on the entire row. Personalization could be done like this:

```
import pyreports
import tablib

# Define my Executor class
class MyExecutor(pyreports.Executor):

    # My custom map method
    def map(self, key, column=None):
        if callable(key):
            ret_data = tablib.Dataset(headers=self.data.headers)
            for row in self:
                # Apply function to data
                ret_data.append(key(row))
            self.data = ret_data
        else:
            raise ValueError(f"{key} isn't function object")
        # Return all data or single column
        if column and self.data.headers:
            self.data = self.select_column(column)

# Test my map
exec = MyExecutor([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)], header=['name
↪', 'surname', 'salary'])

# Function than accept row (iterable)
def stringify(row):
    return [str(item) for item in row]

exec.map(stringify)
```

Add method

You can also add new functionality to the `Executor` object. We are going to add a method to view the data content of an `Executor`.

```
import pyreports

# Define my Executor class
class MyExecutor(pyreports.Executor):

    def __str__(self):
        return self.data


# Print data
exec = MyExecutor([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)], header=['name
↪', 'surname', 'salary'])
print(exec)
```

2.9.2 Expand Report

The `Report` object is really versatile. It is a representation of the report's workflow in full. However, there may be greater needs. For example, before saving the output, make a certain request or save the output before and after processing.

To "break" the working process of the *Report* object, you need to expand it and re-implement its methods.

Save the origin

As anticipated, sometimes we need to save the data before it is processed. To do this, we need to implement a new method to augment or modify the workflow. In this way, we are going to run this worklow: `[INPUT] -> [SAVE ORIGIN] -> [PROCESS] -> [OUTPUT]`

```
import pyreports
import tablib
import os

# Define my Executor class
class MyReport(pyreports.Report):

    def save_origin(self):
        # Save origin in origin file
        if self.output:
            self.output.write(self.data)
            os.rename(self.output.file, 'origin_' + self.output.file)
        # Process report
        self.export()

# Test MyReport
salary55k = pyreports.manager('csv', '/tmp/salary55k.csv')
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],␣
↪headers=['name', 'surname', 'salary'])
report_only_55k = MyReport(mydata, filters=[55000], title='Report salary 55k',␣
↪output=salary55k)

# My workflow report: [INPUT] -> [SAVE ORIGIN] -> [PROCESS] -> [OUTPUT]
report_only_55k.save_origin()
```

Always print

Another highly requested feature is to save and print at the same time. Much like the Unix `tee` shell command, we will implement the new functionality in our custom *Report* object.

```
import pyreports
import tablib

# Define my Executor class
class MyReport(pyreports.Report):

    def tee(self):
        # Print data...
```

(continues on next page)

(continued from previous page)

```
        print(self)
        # ...and save!
        self.export()

# Test MyReport
salary55k = pyreports.manager('csv', '/tmp/salary55k.csv')
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],
↪headers=['name', 'surname', 'salary'])
report_only_55k = MyReport(mydata, filters=[55000], title='Report salary 55k',
↪output=salary55k)

# Print and export
report_only_55k.tee()
```

2.9.3 Extend ReportBook

The `ReportBook` object is a collection of `Report` type objects. When you iterate over an object of this type, you get a generator that returns the *Report* objects it contains one at a time.

Note: Nothing prevents that you can also insert the `MyReport` classes created previously. They are also subclasses of `Reports`.

Book to dict

One of the features that might interest you is to export a *ReportBook* as if it were a dictionary.

```
import pyreports
import tablib


# Instantiate the Report objects
mydata = tablib.Dataset([('Arthur', 'Dent', 55000), ('Ford', 'Prefect', 65000)],
↪headers=['name', 'surname', 'salary'])
report_only_55k = pyreports.Report(mydata, filters=[55000], title='Report salary 55k')
report_only_65k = pyreports.Report(mydata, filters=[65000], title='Report salary 65k')

class MyReportBook(pyreports.ReportBook):

    def to_dict(self):
        return {report.title: report for report in self if report.title}

# Test my book
salary = MyReportBook([report_only_55k, report_only_65k])
salary.to_dict()            # {'Report salary 55k': <Report object, title=Report salary
↪55k>, 'Report salary 65k': <Report object, title=Report salary 65k>}
```

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

pyreports.core, 32
pyreports.datatools, 36
pyreports.exception, 37
pyreports.io, 26

INDEX

Symbols

__add__() (*pyreports.core.ReportBook method*), 34
__init__() (*pyreports.core.Executor method*), 32
__init__() (*pyreports.core.Report method*), 33
__init__() (*pyreports.core.ReportBook method*), 35
__init__() (*pyreports.io.Connection method*), 26
__init__() (*pyreports.io.DatabaseManager method*), 27
__init__() (*pyreports.io.File method*), 28
__init__() (*pyreports.io.FileManager method*), 28
__init__() (*pyreports.io.LdapManager method*), 29
__iter__() (*pyreports.core.Executor method*), 32
__iter__() (*pyreports.core.ReportBook method*), 35
__len__() (*pyreports.core.Executor method*), 32
__len__() (*pyreports.core.ReportBook method*), 35
__repr__() (*pyreports.core.Report method*), 34
__repr__() (*pyreports.core.ReportBook method*), 35
__repr__() (*pyreports.io.DatabaseManager method*), 27
__repr__() (*pyreports.io.FileManager method*), 29
__repr__() (*pyreports.io.LdapManager method*), 29
__str__() (*pyreports.core.Executor method*), 32
__str__() (*pyreports.core.Report method*), 34
__str__() (*pyreports.core.ReportBook method*), 35
__weakref__ (*pyreports.core.Executor attribute*), 32
__weakref__ (*pyreports.core.Report attribute*), 34
__weakref__ (*pyreports.core.ReportBook attribute*), 35
__weakref__ (*pyreports.exception.ReportException attribute*), 37
__weakref__ (*pyreports.io.Connection attribute*), 26
__weakref__ (*pyreports.io.DatabaseManager attribute*), 27
__weakref__ (*pyreports.io.File attribute*), 28
__weakref__ (*pyreports.io.FileManager attribute*), 29
__weakref__ (*pyreports.io.LdapManager attribute*), 29

A

add() (*pyreports.core.ReportBook method*), 35
add_column() (*pyreports.core.Executor method*), 32
aggregate() (*in module pyreports.datatools*), 36
average() (*in module pyreports.datatools*), 36

C

callproc() (*pyreports.io.DatabaseManager method*), 27, 40
chunks() (*in module pyreports.datatools*), 36
clone() (*pyreports.core.Executor method*), 32
commit() (*pyreports.io.DatabaseManager method*), 27, 40
Connection (*class in pyreports.io*), 26
count_columns() (*pyreports.core.Executor method*), 32
count_rows() (*pyreports.core.Executor method*), 32
counter() (*in module pyreports.datatools*), 36
create_database_manager() (*in module pyreports.io*), 31
create_file_manager() (*in module pyreports.io*), 31
create_ldap_manager() (*in module pyreports.io*), 31
CsvFile (*class in pyreports.io*), 26

D

DatabaseManager (*class in pyreports.io*), 27, 40
del_column() (*pyreports.core.Executor method*), 32

E

ExcelFile (*class in pyreports.io*), 28
exec() (*pyreports.core.Report method*), 34
execute() (*pyreports.io.DatabaseManager method*), 27, 40
executemany() (*pyreports.io.DatabaseManager method*), 27, 40
Executor (*class in pyreports.core*), 32
export() (*pyreports.core.Report method*), 34
export() (*pyreports.core.ReportBook method*), 35

F

fetchall() (*pyreports.io.DatabaseManager method*), 27, 40
fetchmany() (*pyreports.io.DatabaseManager method*), 27, 40
fetchone() (*pyreports.io.DatabaseManager method*), 28, 41
File (*class in pyreports.io*), 28
FileManager (*class in pyreports.io*), 28
filter() (*pyreports.core.Executor method*), 33

G

get_data() (*pyreports.core.Executor method*), 33

H

headers() (*pyreports.core.Executor method*), 33

J

JsonFile (*class in pyreports.io*), 29

L

LdapManager (*class in pyreports.io*), 29

M

manager() (*in module pyreports.io*), 31, 41
map() (*pyreports.core.Executor method*), 33
merge() (*in module pyreports.datatools*), 36
module
    pyreports.core, 32
    pyreports.datatools, 36
    pyreports.exception, 37
    pyreports.io, 26
most_common() (*in module pyreports.datatools*), 37
MSSQLConnection (*class in pyreports.io*), 30
MySQLConnection (*class in pyreports.io*), 30

P

percentage() (*in module pyreports.datatools*), 37
PostgreSQLConnection (*class in pyreports.io*), 30
pyreports.core
    module, 32
pyreports.datatools
    module, 36
pyreports.exception
    module, 37
pyreports.io
    module, 26

Q

query() (*pyreports.io.LdapManager method*), 29

R

read() (*pyreports.io.CsvFile method*), 26
read() (*pyreports.io.ExcelFile method*), 28
read() (*pyreports.io.File method*), 28
read() (*pyreports.io.FileManager method*), 29
read() (*pyreports.io.JsonFile method*), 29
read() (*pyreports.io.TextFile method*), 30
read() (*pyreports.io.YamlFile method*), 31
rebind() (*pyreports.io.LdapManager method*), 30
reconnect() (*pyreports.io.DatabaseManager method*), 28, 41
remove() (*pyreports.core.ReportBook method*), 35
Report (*class in pyreports.core*), 33
ReportBook (*class in pyreports.core*), 34
ReportDataError, 37
ReportException, 37
ReportManagerError, 37
reset() (*pyreports.core.Executor method*), 33

S

select_column() (*pyreports.core.Executor method*), 33
send() (*pyreports.core.Report method*), 34
send() (*pyreports.core.ReportBook method*), 35
SQLliteConnection (*class in pyreports.io*), 30

T

TextFile (*class in pyreports.io*), 30

U

unbind() (*pyreports.io.LdapManager method*), 30

W

write() (*pyreports.io.CsvFile method*), 26
write() (*pyreports.io.ExcelFile method*), 28
write() (*pyreports.io.File method*), 28
write() (*pyreports.io.FileManager method*), 29
write() (*pyreports.io.JsonFile method*), 29
write() (*pyreports.io.TextFile method*), 30
write() (*pyreports.io.YamlFile method*), 31

Y

YamlFile (*class in pyreports.io*), 30