Explore Documentation

https://www.github.com/KILLinefficiency/Explore

Introduction

Explore is a REPL data manager for small scale to intermediate scale purposes. Explore has a CLI but also comes with a Python Package which allows programmers to integrate Explore into their Python Programs.

Explore is distributed under the GNU General Public License v3.0.

Explore is currently at version 3.0, codenamed *Kal-El*.

This documentation covers Explore on a **Linux Distribution**.

Getting Explore

Explore is available in source code form only at the official Explore repository and can be cloned from GitHub.
You can either clone the repository directly from the GitHub page or by using the terminal.

Requirements

Make sure that you have newer version of the following installed:

> `git`
>
> `python3`
>
> `pip3`
>
> `node`
>
> `docker` (Optional: For Dockerfile.)
>
> `bash` (Optional: If you decide to use the installer.)

Cloning

```
git clone https://www.github.com/KILLinefficiency/Explore
```

Installing on Linux and WSL

Installation on Linux can be done either manually or automatically via a script.

Manual Installation (not recommened for new users):

1] `cd` into the cloned repository.

```
cd Explore
```

2] Install the dependencies using `pip3`.

```
pip3 install -r requirements.txt
```

3] Run `Explore.py` file.

```
python3 Explore.py
```

4] For using the server, run `server.js` file.

```
node server.js
```

Automatic Installation (recommended):

1] `cd` into the cloned repository.

```
cd Explore
```

2] Run `install.sh`.

```
./install.sh
```

3] Restart the terminal.

This will install Explore at `~/.Explore`.

Explore can be launched by running `explore` in the terminal.

Explore Server can be launched by running `explore-server` in the terminal.

Installing on Windows

1] `cd` into the cloned repository.

```
cd Explore
```

2] Install the dependencies using `pip`.

```
pip install -r requirements.txt
```

3] Run `Explore.py` file.

```
python Explore.py
```

4] For using the server, run `server.js` file.

```
node server.js
```

Files in the Source Code

Explore is made up of seven files:

- `lib.py` contains a few Explore standard functions required for it's functioning.
- `shell.py` is an important file which contains the code responsible for the executions of commands. The code for the login screen is also present in `shell.py`. `shell.py` uses all the standard functions defined in `lib.py`.
- `Explore.py` is the main file run by the Python interpreter. It just launches the `shell.py` file.
- `limit.py` contains the functions related to the `limit` command.
- `error.py` contains standard Explore errors.
- `memory_structures.py` contains functions realted the the working of the **Mess** and the **Cluster**.
- `server.js` launches the server.

Apart from these files, the repository also contains file for Explore's Python module.

- `Explore/explore-package/setup.py` is the Python script that generates a `.whl` package.
- `Explore/explore-package/explore_package/__init__.py` is the main file that contains the entire code for module.

To generate a `.whl` package:

1] `cd` into `Explore/explore-package`

```
cd Explore/explore-package
```

2] Run the `setup.py` file

```
python3 setup.py bdist_wheel
```

3] A `.whl` file will be generated in the newly created `dist` directory.

The `.whl` can be installed using `pip3`

```
pip3 install dist/<the-.whl-file>.whl
```

or

```
pip3 install dist/*
```

Alternatively this process can also be automated via scripts available in the same directory.

Scripts for automating this process:

- `gen-package.sh` : For generating the Explore package.
- `install-package.sh` : For installing the generated Explore package.
- `rem-package.sh` : For removing the genereted package and also uninstalling it from the system.

Memory Locations in Explore

Memory Locations:

> - Mess
> - Cluster
> - Book

Explore uses three memory locations to store it's data. The primary being the **Mess** and the secondary being the **Cluster**. The third memory location, the **Book** is however a non-rewritable memory location which stores parsed data.

The working the **Mess** is very similar to the working of the Stack and is in fact based on the idea of Stack.

The **Cluster** on the other side acts as a hash table. Cluster is made up of keys and the data is assigned to the keys.

The **Book** is non-rewritable, meaning the data once written to it cannot be changed. All the parsed data goes to the Book.

Data Types in Explore

Data Types:

> - num
> - alpha

Explore currently has two data types. Explore uses `num` to store numeric values and `alpha` to store alphabetic (string) values (including alphanumeric values).

Spaces in Explore

Explore allows the usage of spaces. However Explore expects you to use only one space for everything. Additional spaces in Explore will be reduced down to only one space. For additional spaces, the pipe character , `|` has to be used. `|` will act as nothing but a space character.

Like,

Using `disp` to display something:

```
:) > disp Hello World
Hello World
:) > disp Hello          World
Hello World
```

(Does not work.)

```
:) > disp Hello||||||||||World
Hello          World
```

(Works.)

This works not only with `disp` but with all the commands.

Case Insensitivity in Explore

Explore allows case insensitivity only for the head of the command.

Like,

```
disp hello world
```

```
DISP hello world
```

```
Disp hello world
```

```
dISP hello world
```

```
DiSp hello world
```

```
dIsP hello world
```

All of the given commands give the same output.

```
hello world
```

The *log.txt* file

Explore saves all of your commands in a text file simply called `log.txt`. This file is generated in the Explore directory as soon as Explore is set up.

Escaping the *log.txt* file

Explore will by-default save all the commands in `log.txt`. However this can be prevented by prefixing the Explore command with a space ( ). By doing so, the user's command will not be saved in the `log.txt` file.

Referring data in and from the Memory Locations

The data in the **Mess**, the **Cluster** and the **Book** have a special reference convention. Data from the **Mess** requires it's *Position* for access while data from the **Cluster** requires it's *Key* for access. Many commands in Explore use referencing to access a particular value.

Reference to a data item in the **Mess** is it's *Position* preceded by `x_`

```
x_(position-of-the-data-item-in-the-Mess)
```

Reference to a data item in the **Cluster** is it's *Key* preceded by `y_`

```
y_(key-of-the-data-item-in-the-Cluster)
```

Reference to a data item in the **Book** for a parsed text file is it's dataspace and its position preceded by `b_`

```
b_(name_of_dataspace)->(position_of_the_data_item)
```

Reference to a data item in the **Book** for a parsed CSV file is it's dataspace, row and column preceded by `b_`

```
b_(name_of_dataspace)->(row_number)->(column_number)
```

The **Book** uses the arrow character (`->`) for it's referencing.

There should be a space ( ) after and before the reference.
Like,

Valid:

```
calc x_3 + 1 + y_val
```

Invalid:

```
calc x_3+1+y_val
```

The Prompt

Explore uses the smily face emoticon `:)` as the default prompt. However the user can change the prompt emoticon to a different emoticon.

Explore Commands

Explore works with the following 28 commands:

> *01. push*
> *02. getmess*
> *03. pop*
> *04. mov*
> *05. set*
> *06. getcluster*
> *07. rem*
> *08. change*
> *09. count*
> *10. clear*
> *11. csv*

> *12. book*
> *13. getbook*
> *14. get*
> *15. disp*
> *16. clean*
> *17. calc*
> *18. exp*
> *19. export*
> *20. import*
> *21. read*
> *22. dump*
> *23. getlog*
> *24. find*
> *25. limit*
> *26. server*
> *27. info*
> *28. bye*

This documentation covers all of them one by one.

01. push

`push` is the most basic operation when it comes to the **Mess**.
The syntax for `push` is self-explanatory.

`push` simply pushes some data into the **Mess**. The command is followed by the data type and the data:

```
push <data_type> <data>
```

Like,

```
:) > push num 3.14
:) > push alpha pi
```

The contents of the **Mess** can be displayed using the `getmess` command. Explore will display the data according the sequence of pushing.

```
:) > getmess

1. 3.14
2. pi
```

`push` can take alphabetic data contaning spaces too.

```
:) > push alpha something with spaces
:) > getmess

1. 3.14
2. pi
3. something with spaces
```

Instead of using just spaces, the user can also use Explore's pipe character `|` for spaces.

```
:) > push alpha something|with|spaces
:) > getmess

1. 3.14
2. pi
3. something with spaces
```

Explore will, by-default, push the data item to the last position. However, the user can also push the data item to a specific position.

Positions in Explore start from 1.

```
push <data_type> <data> <position>
```

Like,

```
:) > push num 9.8 2
:) > getmess

1. 3.14
2. 9.8
3. pi
4. something with spaces
```

Mathematical expressions can also be passed to `push`.
Like,

```
:) > push num 22/7
:) > getmess

1. 3.142857142857143
```

The user can also push the data from within the **Mess** by referencing it.

Like,

```
:) > push num 3.14
:) > getmess

1. 3.14

:) > push num x_1
:) > getmess

1. 3.14
```

```
2. 3.14
```

The same operation of referencing is applicable to the **Cluster** and the **Book** too.

02. getmess

`getmess` displays all the contents of the **Mess** along with their positions.

```
getmess
```

Like,

```
:) > getmess

1. 3.14
2. 9.8
3. pi
4. something with spaces
```

Here, the positions of the data items are succeeded by the data items.
For example, 1 is the position of *3.14*, 2 is the position of *9.8* and so on.

03. pop

`pop` removes the last data item from the **Mess**.

```
pop
```

Like,

```
:) > getmess

1. 3.14
2. 9.8
3. pi
4. something with spaces

:) > pop
:) > getmess

1. 3.14
2. 9.8
3. pi
```

`pop`, by-default, removes the last data item from the **Mess**. However, the user can remove a data item from anywhere in the **Mess** by specifying it's position.

```
pop <position>
```

Like,

```
:) > getmess

1. 3.14
2. 9.8
3. pi

:) > pop 2
:) > getmess

1. 3.14
2. pi
```

`pop` also supports multiple item deletion.

```
pop <position_of_item_1> <position_of_item_2> ...
```

Like,

```
:) > getmess

1. 3.14
2. pi

:) > push num 9.8
:) > pop 1 3
:) > getmess

1. pi
```

04. mov

`mov` comes handy when the user wants to replace a data item with another one in the **Mess**. `mov` requires a data type, data item and the position of the data item to be replaced.

```
mov <data_type> <data> <position>
```

Like,

```
:) > getmess

1. 3.14
2. pi

:) > mov num 9.8 1
:) > mov alpha g 2
:) > getmess

1. 9.8
```

```
2. g
```

The user can also move the data from within the **Mess**.

Like,

```
:) > push alpha ninepointeight
:) > push num 9.8
:) > getmess

1. ninepointeightfour
2. 9.8

:) > mov num x_2 1
:) > getmess

1. 9.8
2. 9.8
```

The same operation of referencing can also be performed on the data from the **Cluster** and the **Book** too.

05. set

`set` is a **Cluster** related command. It allows the user to set a data item in the **Cluster**. Unlike the **Mess**, data items in the **Cluster** have a distinct key. The data item in the **Cluster** is made up of *Keys* and their *Values*.

```
set <key> <data_type_of_the_value> <value>
```

Keys in Explore do not have a data type. The data type has to be specified for the value.

```
:) > set pi num 3.14
```

The contents of the **Cluster** can be displayed by using the `getcluster` command.

```
:) > set pi num 3.14
:) > set something alpha anything
:) > getcluster

Key : Value

pi : 3.14
something : anything
```

Mathematical expressions can also be passed to `set`.
Like,

```
:) > set pi num 22/7
:) > getcluster

Key : Value

pi: 3.142857142857143
```

The user can set the values from the **Mess**, the **Cluster** or from the **Book**.
Like,

```
:) > push num 3.14
:) > set value alpha pi
:) > set pi num x_1
:) > set name alpha y_value
:) > getcluster

Key : Value

value : pi
pi : 3.14
name : pi
```

Keys cannot contain spaces and a *Key* can be assigned to one value at a time only.

06. getcluster

`getcluster` displays the contents of the **Cluster**. The contents include both the *Keys* and the *Values*.
By default `getcluster` displays all the *Keys* and *Values* present in the **Cluster**. However, `getcluster` can also be used to access the *Keys* or the *Values* only.

```
getcluster
```

```
getcluster keys
```

```
getcluster values
```

Like,

```
:) > getcluster

Key : Value

pi : 3.14
something : anything

:) > getcluster keys

Key :
```

```
pi :
something :

:) > getcluster values

: Values

: 3.14
: anything
```

07. rem

`rem` removes a data item from the **Cluster**. By default, `rem` will remove the last data item from the **Cluster**.

```
rem
```

Like,

```
:) > getcluster

Key : Value

pi : 3.14
something : anything

:) > rem
:) > getcluster

Key : Value

pi : 3.14
```

However, `rem` can also be used to remove a specific data item from the **Cluster**. For this, the user needs to specify the *Key* of the data item to be removed.

```
rem <key>
```

Like,

```
:) > set g num 9.8
:) > getcluster

Key : Value

pi : 3.14
g : 9.8

:) > rem pi
```

```
:) > getcluster

Key : Value

g : 9.8
```

`rem` also supports multiple item deletion.

```
rem <key1> <key2> ...
```

Like,

```
:) > set name alpha explore
:) > set value num 3.14
:) > set var alpha hello
:) > getcluster

Key : Value

name : explore
value : 3.14
var : hello

:) > rem name value
:) > getcluster

Key : Value

var : hello
```

08. change

`change` is a **Cluster** related operation. In Explore, the user cannot assign a *Value* to an existing *Key*. This means that using `set` to reassign a value won't work.
In this case, `change` comes handy. As the name suggests, `change` is used to change a *Value* of an existing *Key*.

```
change <existing_key> <new_datatype_for_the_new_value> <new_value>
```

Like,

```
:) > getcluster

Key : Value

g : 9.8

:) > change g num 10
:) > getcluster
```

```
Key : Value

g : 10
```

The *Value* can also be changed by referring it from within the **Cluster**, the **Mess** or the **Book**.

Like,

```
:) > push num 3.14
:) > getmess

1. 3.14

:) > set pi alpha threpointonefour
:) > set pi_val num 22/7
:) > getcluster

Key : Value

pi : threepointonefour
pi_val : 3.142857142857143

:) > change pi num x_1
:) > getcluster

Key : Value

pi : 3.14
pi_val : 3.142857142857143

:) > change pi num y_pi_val
:) > getcluster

Key : Value

pi : 3.142857142857143
pi_val : 3.142857142857143
```

09. count

`count` displays the number of data items in the **Mess**, the **Cluster** and the **Book** as specified by the user.

```
count mess
```

```
count cluster
```

Like,

```
:) > push num 3.14
```

```
:) > push num 9.8
:) > push alpha some|random|text
:) > set version num 1
:) > set ten num 1010
:) > getmess

1. 3.14
2. 9.8
3. some random text

:) > getcluster

Key : Value

version : 1
ten : 1010

:) > count mess
3
:) > count cluster
2
```

`count` also works on the **Book**.

```
count book
```

10. clear

Sometimes when you are working hard, you make a lot of mess on the screen. `clear` clears the screen and returns the Explore Prompt back.

```
clear
```

11. csv

`csv` reads a CSV file. CSV stands for *Comma-Separated Values*. `csv` will read the values from a `.csv` file (or any text file with the same text format) and display it in a tabular form.

```
csv <file_name_with_complete_or_relative_address>
```

The default spacing for the `csv` command is 4 tabs. The user can optionally change this spacing.

```
csv config tab <required_tabs>
```

The default field separator for the `csv` command is `,`. The user can optionally change this.

```
csv config fs <required_field_separator>
```

Like,

```
:) > read /home/me/Desktop/file.csv

abc,def,ghi,jkl
123,456,789,012
345,678,901,234
567,890,123,456
789,012,345,678


:) > csv /home/me/Desktop/file.csv

    abc             def             ghi             jkl
1.  123             456             789             012
2.  345             678             901             234
3.  567             890             123             456
4.  789             012             345             678

:) > csv config tab 1
:) > csv /home/me/Desktop/file.csv

    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
3.  567 890 123 456
4.  789 012 345 678
```

```
:) > read /home/me/Desktop/file1.csv

abc;def;ghi;jkl
123;456;789;012
345;678;901;234
567;890;123;456
789;012;345;678

:) > csv config fs ;
:) > csv /home/me/Desktop/file1.csv

    abc             def             ghi             jkl
1.  123             456             789             012
2.  345             678             901             234
3.  567             890             123             456
4.  789             012             345             678
```

12. book

As seen in the previous section, `csv` can read and display the data from a CSV file in a tabular form, however `csv` does not provide any medium to access the parsed data.
The data from a CSV file can be parsed and accessed using `book`.

`book` currently supports parsing of text file (reading lines from a text file) and CSV files only.

As the name suggests, this parsed data goes into the **Book**. Inside the **Book**, the parsed data is

stored individually as per the parsed files into units called *Data Spaces*.

Data Space name and the file address *cannot* contain any spaces.

```
book <file_type> <data_space_name> <file_name_with_full_or_relative_address>
```

Like,

Consider a file `list.txt`

```
Debian
Arch Linux
Fedora
Linux Mint
Ubuntu
Manjaro
Gentoo
```

For text files:

Referencing data from the **Book**:

```
b_(name_of_dataspace)->(item_position)
```

```
:) > book text distros /home/me/Desktop/list.txt

Debian
Arch Linux
Fedora
Linux Mint
Ubuntu
Manjaro
Gentoo


:) > disp b_distros->2
Arch Linux
:) > disp b_distros->7
Gentoo
:) > disp b_distros->4
Linux Mint
```

For CSV files:

Referencing data from the **Book**:

```
b_(name_of_dataspace)->(row_number)->(column_number)
```

```
:) > book csv info /home/me/Desktop/file.csv

    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
```

```
3.  567 890 123 456
4.  789 012 345 678

:) > push num b_info->3->3
:) > push alpha b_info-1->2
:) > getmess

1. 901.0
2. def
```

The following example parses the CSV file and then access the data using referencing. The data at the *3rd row*, *3rd column* and the data of the *1st row*, *2nd column* of the *Data Space* called *info* is accessed and push to the **Mess**.

13. getbook

`getbook` displays all the *Data Spaces* from the **Book**.

```
getbook
```

Like,

```
:) > book csv info /home/me/Desktop/file.csv

    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
3.  567 890 123 456
4.  789 012 345 678

:) > book csv data /home/me/Desktop/file.csv

    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
3.  567 890 123 456
4.  789 012 345 678

:) > getbook

info:
    abc         def         ghi         jkl
    123         456         789         012
    345         678         901         234
    567         890         123         456
    789         012         345         678

data:
    abc         def         ghi         jkl
    123         456         789         012
    345         678         901         234
    567         890         123         456
    789         012         345         678
```

Here the same CSV file has been parsed twice in two different *Data Spaces*.

14. get

`get` accesses the data from the **Mess**, the **Cluster** and the **Book**

```
get mess <position>
```

```
get cluster <key>
```

```
get book <dataspace> <item_position>
```

```
get book <dataspace> <row> <column>
```

Like,

```
:) > push num 9.8
:) > set pi alpha threepointonefour
:) > book csv data file.csv

    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
3.  567 890 123 456
4.  789 012 345 678

:) > get mess 1
9.8
:) > get mess pi
threepointonefour
:) > get book data 3 3
901
```

15. disp

`disp` is similar to `print()` in Python. `disp` displays the text passed to it.

```
disp <text>
```

Like,

```
:) > disp Hello World!
Hello World!
```

`disp` can also be used to display data directly from the **Mess**, the **Cluster** and the **Book**.

```
disp x_<position>
```

```
disp y_<key>
```

```
disp b_(dataspace)->(row)->(column)
```

Like,

```
:) > push num 3.14
:) > set value alpha PI
:) > disp y_value is x_1
PI is 3.14
```

16. clean

`clean` is useful when the user wants to remove all the data items from the **Mess**, **Cluster** or the **Book**.

```
clean mess
```

```
clean cluster
```

```
clean book
```

Like,

```
:) > push num 3.14
:) > push num 9.8
:) > set var1 alpha anything
:) > set var2 alpha something
:) > getmess

1. 3.14
2. 9.8

:) > getcluster

Key : Cluster

var1 : anything
var2 : something

:) > clean mess
:) > getmess
:) > clean cluster
:) > getcluster
:) > book csv data /home/me/Desktop/file.csv
```

```
    abc def ghi jkl
1.  123 456 789 012
2.  345 678 901 234
3.  567 890 123 456
4.  789 012 345 678

:) > getbook

data:
    abc         def         ghi         jkl
    123         456         789         012
    345         678         901         234
    567         890         123         456
    789         012         345         678

:) > clean book
:) > getbook
```

17. calc

`calc` stands for “calculator”. As the name suggests, the `calc` command performs mathematical calculations when a mathematical expression is passed to it.

```
calc <mathematical_expression>
```

Like,

```
:) > calc 5 + 10
15
```

`calc` can perform addition(+), subtraction(-), multiplication(*), division(/) and modulus(%). Parentheses can be used to define an order of precendence.

Like,

```
:) > calc (5 + 10) * 2
30
```

Values from the **Mess**, the **Cluster** and the **Book** can also be referred in `calc`.

Like,

```
:) > push num 45
:) > set var num 55
:) > calc x_1 + y_var
100
:) > calc x_1 + y_var + 100
200
```

`calc` can also be used for checking mathematical equations.
Like,

```
:) > calc 5 == 5
Yes. (1)
:) > calc 10 < 6
No. (0)
```

18. exp

`exp` stands for expression and it changes the prompt expression. The default prompt emoticon is the smily face emoticon `:)` .

```
exp
```

Like,

```
:) > exp
1. :)
2. ;)
3. :|
4. :(
5. :D >
6. :P >
7. :O >
```

The user can change the expression by passing the position of the emoticon from the list displayed after running `exp` .

```
exp <expression_position_number>
```

Like,

```
:) > exp
1. :)
2. ;)
3. :|
4. :(
5. :D >
6. :P >
7. :O >
:) > exp 1
:) > exp 2
;) > exp 3
:| > exp 4
:( > exp 1
:) >
```

The user can see the changed expression on the prompt.

19. export

Explore saves all of it's data items in the **Mess** and in the **Cluster**. The users are free to store as

much as data they wish. However, Explore clears all of it's data once it is closed which means that there will be no data present if Explore is re-launched.

It would be very inconvenient for the user to re-enter all the data items. This is where `export` saves the day.
`export` allows the user to export the data items into a text file for later use.

```
export <memory_location> <file_name_with_complete_or_relative_address>
```

Like,

```
:) > push num 3.14
:) > push alpha something
:) > set name alpha Explore
:) > set ver num 3
:) > export mess /home/me/Desktop/my_mess.txt
:) > export cluster /home/me/Desktop/my_cluster.txt
```

Here, the data items from the **Mess** and the **Cluster** have been exported to *my_mess.txt* and *my_cluster.txt* respectively.

It's not a good idea to edit an exported file manually as it can cause errors while importing it.

(Here, `me` is the username on a Linux machine. The following example shows a Linux File System, but `export` works just fine on Windows and MacOS too).

20. import

Exporting data is of no use if the user can't import it back into Explore. `import` imports the data back into the *Memory Locations* from the exported file.

`import` has 2 import modes:

> `w` : write
>
> `rw` : rewrite

```
import <memory_location_to_be_imported_into> <import_mode> <exported_file_name_with_complete_or_relative_address>
```

Import mode `w` will import the data items into the entered *Memory Location* along with the already existing data items.

Import mode `rw` will import the data items into the entered *Memory Location* but will remove all the already existing data items.

Like,

my_mess.txt and *my_cluster.txt* here are the same files which were exported in the previous section.

Using `w`:

```
:) > push alpha Kal-El
:) > set codename alpha Kal-El
:) > getmess

1. Kal-El

:) > getcluster

Key : Value

codename : Kal-El

:) > import mess w /home/me/Desktop/my_mess.txt
:) > getmess

1. Kal-El
2. 3.14
3. something

:) > import cluster w /home/me/Desktop/my_cluster.txt
:) > getcluster

Key : Value

codename : Kal-El
name : Explore
ver : 3
```

Using `rw`:

```
:) > push alpha Kal-El
:) > set codename alpha Kal-El
:) > getmess

1. Kal-El

:) > getcluster

Key : Value

codename : Kal-El

:) > import mess rw /home/me/Desktop/my_mess.txt
:) > getmess

1. 3.14
2. something

:) > import cluster rw /home/me/Desktop/my_cluster.txt
:) > getcluster

Key : Value

name : Explore
ver : 3
```

21. read

`read` displays the contents of a text file passed to it.

```
read <file_name_with_complete_or_relative_address>
```

Like,

my_text.txt file on Desktop

```
some random text...
```

Explore

```
:) > read /home/me/Desktop/my_text.txt

some random text...


:) >
```

22. dump

`dump` is useful when the user wants to write certain text/data into a file. `dump` will keep stacking the data to the next line as long as it's being used for the same file. The file address *cannot* contain any spaces.

```
dump <text> <file_name_with_complete_or_relative_address>
```

Like,

```
:) > dump hello /home/me/Desktop/hey.txt
:) > read /home/me/Desktop/hey.txt

hello


:) > push num 55
:) > dump x_1 /home/me/Desktop/hey.txt
:) > set name alpha Explore
:) > dump Hey y_name ! /home/me/Desktop/hey.txt
:) > read /home/me/Desktop/hey.txt

hello
55.0
Hey Explore !
```

23. getlog

Explore saves all of the user-entered commands in a file called `log.txt` in the same directoy as that of the main file, `Explore.py`. However, all of the user-entered commands are written to `log.txt` only after Explore is closed.

`getlog` simply reads the `log.txt` file so that the user can browse through the previously entered commands.

```
getlog
```

Like,

Run 1:

```
:) > push num 3.14
:) > set val alpha PI
:) > disp y_val is x_1
PI is 3.14
:) > exit
```

Run 2:

```
:) > getlog
1. push num 3.14
2. set val alpha PI
3. disp y_val is x_1
4. exit
```

If the user deletes the `log.txt` file, it will be automatically created the next time Explore is launched.

24. find

`find` searches for the entered data item. `find` can search data items from the **Mess** and the *Keys*, the *Values* from the **Cluster** and the parsed values for the dataspaces of the **Book**.

```
find <data_type> <data_item_to_be_searched>
```

Like,

```
:) > push alpha pi
:) > push num 3.14
:) > set pi num 3.14
:) > set PI num 3.14
:) > find num 3.14
```

```
Location: Mess   Datatype: num   Position: 2
Location: Cluster    Itemtype: Value     Datatype: num   Key: pi
Location: Cluster    Itemtype: Value     Datatype: num   Key: PI
:) > find alpha pi
Location: Mess   Datatype: alpha     Position: 1
Location: Cluster    Itemtype: Key   Value: 3.14
```

25. limit

`limit` is useful when the user wants to set a limit for the number of times a command is called.

```
limit enable
```

```
limit disable
```

```
limit status
```

Setting a limit

```
limit enable <command_name> <limit>
```

A limit can be set to all the commands at once.

```
limit enable all <limit>
```

`limit enable` cannot take multiple commands at once, so the commands should to be split into multiple `limit enable` commands.

Like,

Invalid

```
:) > limit enable push disp 4
```

Valid

```
:) > limit enable push 4
:) > limit enable disp 4
```

The following example shows the `disp` command but `limit` works with all the commands.

```
:) > limit enable disp 3
:) > disp running for the first time
running for the first time
:) > disp running for the second time
running for the second time
:) > disp last time
last time
```

```
:) > disp i wont show up :(
Command Limit Reached.
```

Here the limit is set to 3, thus the `disp` command becomes executable for only 3 time. For the fourth time, an error message will show up.

Removing a limit

```
limit disable <command_name>
```

`limit disable` can take multiple commands at once.

Like,

```
limit disable <command_1> <command_2> ...
```

Limits from all the commands can be removed at once.

```
limit disable all
```

Removing the limit set to `disp` in the previous section (whole example):

```
:) > limit enable disp 3
:) > disp running for the first time
running for the first time
:) > disp running for the second time
running for the second time
:) > disp last time
last time
:) > disp i wont show up :(
Command Limit Reached.
:) >
:) > limit disable disp
:) > disp i am back again
i am back again
```

Getting information about a limit

```
limit status <command>
```

`limit status` can take multiple commands at once.

Like,

```
limit status <command_1> <command_2> ...
```

The user can get the limit status of all the commands at once.

```
limit status all
```

Using `limit status`:

```
:) > limit status disp

1. Command: disp
Limit Enabled: No
Requests: 0
Limit: 0

:) > limit enable disp 3
:) > limit status disp

1. Command: disp
Limit Enabled: Yes
Requests: 0
Limit: 3

:) > disp try 1
try 1
:) > limit status disp

1. Command: disp
Limit Enabled: Yes
Requests: 1
Limit: 3
```

26. server

`server` allows sharing of data from a device running Explore to one or more device running Explore if all of them are connected to a common network like WiFi.

Let's consider Device1 and Device2, each one running Explore, connected to a common WiFi.

With `server`, it's possible to share the **Mess** and the **Cluster** contents wirelessly from Explore running on Device1 to Explore running on Device2.

The Explore Server undertakes this task.

Running Explore Server

If Explore is installed via the installer, the Explore Server can be launched by running

```
explore-server
```

Alternatively the Explore Server can be launched from the `server.js` file (if the installer was not used).

1] Navigate to the cloned repository.

2] Run

```
node server.js
```

For the sender (Device1), the data will be hosted at

```
0.0.0.0:2166
```

For the reciever (Device2), the data will be hosted at

```
<ip.of.the.sender>:2166
```

Explore uses port **2166**

Running the Explore Server takes up the entire process, so it's better to run it on a separate terminal.

Sending the data (Device1)

The sender has to run just one command inside Explore for sending the data.

```
server update all
```

This will pipe all the data from the **Mess** and the **Cluster** to the Explore Server.

Alternatively, the user can specifically choose the **Mess** or the **Cluster** to be sent by using

```
server update mess
```

and

```
server update cluster
```

Recieving the data (Device2)

The reciever requires the IP address of the sender (Device1).

This can done by running `server connect` inside Explore

```
server connect <ip.of.the.sender>
```

Next, the data can be recieved using `server fetch`

```
server fetch all
```

Alternatively, the user can specifically choose the **Mess** or the **Cluster** to be recieved by using

```
server fetch mess
```

and

```
server fetch cluster
```

Device1

```
:) > push num 200
:) > set status alpha haha data go brrr
:) > getmess

1. 200.0

:) > getcluster

Key : Value

status : haha data go brrr

:) > server update all
```

Device2

```
:) > getmess
:) > getcluster
:) > server connect <ip.of.the.sender>
:) > server fetch all
:) > getmess

1. 200.0

:) > getcluster

Key : Value

status : data go brrr
```

Multiple devices can be connected in such way.

27. info

`info` displays Explore-related information. `about` is an alternative to `info`.

```
info
```

Like,

```
:) > info

Explore v3.0
Codename: Kal-El
License: GNU General Public License v3.0
Author: Shreyas Sable
Repository: https://www.github.com/KILLinefficiency/Explore

:) > about

Explore v3.0
Codename: Kal-El
```

```
License: GNU General Public License v3.0
Author: Shreyas Sable
Repository: https://www.github.com/KILLinefficiency/Explore
```

28. bye

As the name suggests, `bye` closes down the current instance of Explore. The alternatives to `bye` are `bye.`, `exit` and `exit.`.

```
bye
```

Like,

```
:) > bye

Bye.
```

The Explore Package

Explore also offers a Python Package for integrating Explore in Python Programs.
If you have used the installer or have installed dependencies from `requirements.txt`, then the Explore Package is already installed on your system.

Alternatively, the Explore Package can be installed from pypi.org:

```
pip3 install explore_package
```

Usage

The Explore package has two in-built functions, the `invoke()` function and the `run()` function.

> - `invoke()`
> - `run()`

The `invoke()` function runs Explore commands as strings in the Python Program. The `run()` function runs the user-made Explore Scripts in the Python Program.

These two functions are a part of an `Explore` object.

Getting an Explore object

Each Explore object has it's own **Mess**, **Cluster** and **Book**.

```
from explore_package import Explore
```

```
db = Explore() # Object 1
db1 = Explore() # Object 2
```

The object can be named whatever the user wants. Thus, multiple instances of Explore are possible.

invoke()

`invoke()` takes in one string argument. `invoke()` for few Explore commands returns data, while most of them displays it out directly in the Python Program.

```
invoke("<explore_command>")
```

Like,

Python Program

```
from explore_package import Explore

db = Explore()

db.invoke("disp Hello World!")
db.invoke("push num 3.14")
db.invoke("set val alpha PI")
db.invoke("disp y_val is x_1")
```

The output on running would be:

```
Hello World!
PI is 3.14
```

Data-Returning and Non-Data-Returning Values

Data-Returning Commands do not display the output/data directly. The user will need a Python variable to hold the data and then print it out or perform operations on it.

Non-Data-Returning commands give a direct output.
There are 24 Explore Commands in the Explore Package.

Non-Data-Returning and Data-Returning commands in the Explore Package:

Non-Data-Returning	Data-Returning
push	*count*
pop	*get*
mov	*calc*
set	*info*

rem	*getmess*
change	*getcluster*
disp	*getbook*
clean	*find*
export	*limit status*
import	*read*
csv	*server ip*
book	
dump	

Like,

```
from explore_package import Explore
db = Explore()
db.invoke("push num 3.14")
db.invoke("set pi num 3.14")
if db.invoke("get mess 1") == db.invoke("get cluster pi"):
    print("PI is 3.14")
else:
    print("PI is not 3.14")
```

The output on running would be:

```
PI is 3.14
```

Here, both the `get` commands returned data. One `get` command returned data from the **Mess** and the other `get` command returned data from the **Cluster**. The data returned is compared in an `if` statement and since the returned data items were equal, the block inside `if` was executed.

Return Types for Data-Returning Commands

Data-Returning Commnds return data of different data types. It's good to know the Return Types so that it's helpful to the user while performing different operations on the returned data.

All the Return Types are Python's data types.

Command	Return Type
count	*Integer* (`int`)
get	*Float* / *String* (`float` / `str`)
calc	*Float* (`float`)
info	*Dictionary* (`dict`)
getmess	*List* (`list`)

getcluster	*Dictionary* (`dict`)
getbook	*Dictionary* (`dict`)
find	*Dictionary* / *Dictionaries* inside a *List* (`list` and `dict`)
limit status	*Dictionary* (`dict`)
read	*List* (`list`)
server ip	*String* (`str`)

Like,

Python Program

```
from explore_package import Explore

db = Explore()

db.invoke("push num 3.14")
db.invoke("set pi num 3.14")

# Getting a search result for the number 3.14
search = db.invoke("find num 3.14")
print(search)
print() # Printing a blank line...

# Getting Explore's information
information = db.invoke("info")
print(information)
print() # Printing a blank line...

# Getting specific data from the returned data
print(search[1]["Key"])
print(information["CODENAME"])
```

Output on running would be:

```
[{'Location': 'Mess', 'Datatype': 'num', 'Position': 1}, {'Location': 'Cluster', 'Itemtype': 'Val
ue', 'Datatype': 'num', 'Key': 'pi'}]

{'VERSION': 3.0, 'CODENAME': 'Kal-El', 'LICENSE': 'GNU General Public License v3.0', 'AUTHOR': 'S
hreyas Sable', 'REPOSITORY': 'https://www.github.com/KILLinefficiency/Explore'}

pi
Kal-El
```

Using Formatted Strings

The user can get a data item into the Python Program from either the **Mess**, the **Cluster** or the **Book** by using `get`. But there is no standard Explore command to send a data/value from the Python Program into the **Mess** or the **Cluster**.

This is where Python's Formatted Strings come into use.

The user can directly use operations like `push` and `set` by passing the data/value inside a formatted string in `invoke()`.

Like,

```
from explore_package import Explore
db = Explore()
some_value = 9.8
some_name = "Explore"
db.invoke(f"push num {some_value}")
db.invoke(f"set name alpha {some_name}")
print(db.invoke("getmess"))
print()
print(db.invoke("getcluster"))
```

Here, `some_value` and `some_name` are Python variables which are sent into the Explore instance.

The output on running would be:

```
[9.8]

{'name': 'Explore'}
```

run()

`run()` lets the user run Explore Scripts in a Python Program. The user does not have to run all the everyday commands everytime. This task can be automated with the help of Explore Scripts.

An Explore Script can have any extension as long as it's a text file. It's always better to use `.txt` extension for Explore Scripts.

Explore Scripts can only have **Non-Data-Returning** commands in them.

```
run("<file_name_with_complete_or_relative_address>")
```

Like,

Here's a file `script.txt` on the user's Desktop.

script.txt

```
disp hello
```

Python Program

```
from explore_package import Explore
db = Explore()
db.run("/home/me/Desktop/script.txt")
```

The output on running the Python Program:

```
hello
```

Explore Scripting also allows *Commenting*. Explore Script Comments begins with `...`

```
... This is a Comment!
```

Like,

script.txt

```
... Setting the numeric value
push num 3.14

... Setting the name
set name alpha PI

... Displaying it out
disp y_name is x_1
```

Python Program

```
from explore_package import Explore
db = Explore()
db.run("/home/me/Desktop/script.txt")
```

The output on running the Python Program:

```
PI is 3.14
```

Uninstalling the Explore Package

If no longer needed, the Explore Package can be uninstalled with `pip3` .

```
pip3 uninstall explore_package
```

Hey There!

Hello Explorers! I hope you had fun using Explore as much as I had making it. If you like it, feel free to make contributions at https://www.github.com/KILLinefficiency/Explore or contact me at shreyas.sable2166@gmail.com.

Shreyas Sable