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 2.0, codenamed *Terrific*.

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

Getting Explore

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

Cloning via terminal:

Make sure you have a newer version of `git`, `python3`, `pip3` and `make` installed on your machine.

1] Clone the repository

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

2] `cd` into the cloned repository

```
cd Explore
```

3] Run the `Explore.py` file

```
python3 Explore.py
```

You now have Explore set-up and running.

Using the `makefile`

`makefile` makes it easy to install Explore on your Linux machine. Using the `makefile` will produce an executable which can be run from anywhere on the system via the terminal.

Navigate to the Explore directory where the `makefile` is present. It is recommended to run `make install` or just `make` to setup Explore on your system in one step. However, the users

have the freedom to use different clauses to initiate only the setup processes which they wish. Here are the different `makefile` clauses:

`make` / `make install`: (Recommended) Installs Explore on the user's machine. This will also install the Explore Package from pypi.org so make sure to have an internet connection.

`make script`: Generates an Explore executable.

`make path`: Adds the Explore directory to the PATH variable.

`make installpackage`: Installs the Explore Package via `pip3`.

`buildpkg`: Generates the Explore Package from the code at `Explore/explore-package/explore_package/__init__.py`

`make installpkg`: Uninstalls the previously installed package and installes the user-made package. (The user needs to run `make buildpkg` first)

`make update`: Updates Explore by pulling the latest changes from the GitHub repository.

`make reset`: Resets the password and the encryption key.

Files in the Source Code

Explore is made up of four files:

- `cipher.py` contains the code for the security layer in Explore.
- `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.

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

- `Explore/explore-package/setup.py` is the script that creates 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 `pip`

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

Login System and Security Layer in Explore

After launching Explore for the first time, the user will be asked to set a password and an encryption key. Explore will save this data for you and you'll just have to enter the previously set password to login into Explore the next time you fire it up. The encryption key is solely for security purposes and can be set to any arbitrary value. The encryption key can be set to a random value by entering `0` when asked for it.

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 giant hash table. Cluster is made up of keys and the data 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 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 spaces. For additional spaces, the pipe character , `|` has been created. `|` will act as nothing but a space character.

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 Explore is set up. The Explore Login System is connected to this `log.txt` file. So, it's not a good idea to delete the `log.txt` file even if it's no longer necessary to the user.

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** and the **Cluster** 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** is it's dataspace, row and column preceded by `b_`

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

There should be atleast 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 23 commands:

> *01. push*
> *02. getmess*
> *03. pop*
> *04. mov*

> *05. sortmess*
> *06. set*
> *07. getcluster*
> *08. rem*
> *09. change*
> *10. count*
> *11. clear*
> *12. csv*
> *13. book*
> *14. getbook*
> *15. get*
> *16. disp*
> *17. clean*
> *18. calc*
> *19. exp*
> *20. export*
> *21. import*
> *22. read*
> *23. dump*
> *24. getlog*
> *25. find*
> *26. info*
> *27. 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 Explore's pipe character `|` for representating 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 is applicable to the **Cluster** 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 can also be performed on the data from the **Cluster**.

05. sortmess

`sortmess` simply sorts the **Mess** in ascending or descending order.

Order for sorting in ascending is represented by **a** or **A** and the order for sorting in descending is represented by **d** or **D**.

> Ascending: `a` or `A`
> Descending: `d` or `D`

```
sortmess <order>
```

Like,

```
:) > push num 30
:) > push num 10
:) > push num 20
:) > getmess
1. 30
2. 10
3. 20
:) > sortmess a
:) > getmess
1. 10
2. 20
3. 30
:) > sortmess d
:) > getmess
1. 30
2. 20
3. 10
```

`sortmess` can sort the mess only if it contains numeric data. `sortmess` can't sort alphabetic data.

06. set

`set` is a **Cluster** related command. It allows the user to set a data item is the **Cluster**. Unlike the **Mess**, data items in the **Cluster** have a distinct key. The data item acts as the value to the user-specified key.

```
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** of 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.

07. 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 *Keys* or *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
```

08. 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
:) > rem name value
:) > getmess
Key : Value

var : hello
```

09. 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** ot the **Book**.

Like,

```
:) > push num 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
```

10. 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
```

11. 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
```

12. csv

`csv` reads a CSV file. CSV stands for *Comma-Separated Values*. `csv` will read the values from the `.csv` file and dislpay 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 <spacing> <file_name_with_complete_or_relative_address>
```

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 1 /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
```

13. 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 on 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,

```
:) > 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
:) > getmess
1. 901.0
```

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

Referencing data from the **Book**:

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

14. 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*.

15. get

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

```
get mess <position>
```

```
get cluster <key>
```

```
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
```

16. 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
```

17. 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
```

18. 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(%). Parenthesis 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)
```

19. 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.

20. 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 2
:) > 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.

For data privacy, Explore also supports data encryption while exporting. The key is to be provided with the `export` command, preceded by `e_`.

```
export <memory_location> <file_name_with_complete_or_relative_address> e_<encryption_key>
```

The encryption key should always be a whole number.

Like,

```
:) > push num 45
:) > push alpha hey
:) > export mess /home/me/Desktop/enc_mess.txt e_60
```

In this example, the **Mess** has been exported into a file called *enc_mess.txt*, encrypted with the encryption key: 60.

(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).

21. 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,

Using `w` :

```
:) > push alpha Terrific
:) > set codename alpha Terrific
:) > getmess
1. Terrific
:) > getcluster
Key : Value

codename : Terrific
:) > import mess w /home/me/Desktop/my_mess.txt
:) > getmess
1. Terrific
2. 3.14
3. something
:) > import cluster w /home/me/Desktop/my_cluster.txt
:) > getcluster
Key : Value

codename : Terrific
name : Explore
ver : 2
```

Using `rw` :

```
:) > push alpha Terrific
:) > set codename alpha Terrific
:) > getmess
1. Prometheus
:) > getcluster
Key : Value

codename : Terrific
:) > 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 : 2
```

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

Encrypted exported file **Mess**/**Cluster** files can be exported back if the correct decryption key is provided with the `import` command, preceded by `d_`.

```
import <memory_location_to_be_imported_into> <import_mode> <exported_file_name_with_complete_or_relative_address> d_<decryption_key>
```

Decryption Key is the same as the Encryption Key.

Like,

```
:) > import mess rw /home/me/Desktop/enc_mess.txt d_60
:) > getmess
1. 45.0
2. hey
```

enc_mess.txt is the same encrypted file which was exported in the previous section.

22. 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...


:) >
```

23. 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 !
```

24. 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
```

`log.txt` is an important file for Explore to run. Therefore, it's not a good idea to delete it even if the user doesn't need it.

25. 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
```

26. info

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

```
info
```

Like,

```
:) > info

Explore v2.0
Codename: Terrific
License: GNU General Public License v3.0
Author: Shreyas Sable
Repository: https://www.github.com/KILLinefficiency/Explore

:) > about

Explore v2.0
Codename: Terrific
License: GNU General Public License v3.0
Author: Shreyas Sable
Repository: https://www.github.com/KILLinefficiency/Explore
```

27. 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.

Installing the Explore Package

The Explore Package is available in the `explore-package` directory as a `.whl` package.

`cd` into `explore-package`

```
cd explore-package
```

Use `pip3` to install the package

```
sudo pip3 install explore_package-2.0-py3-none-any.whl
```

Installing Explore Package from pypi.org :
The Explore Package can be directly installed from pypi.org via pip3.

```
sudo 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

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

```
import explore_package as e
e.invoke("disp Hello World!")
e.invoke("push num 3.14")
e.invoke("set val alpha PI")
e.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.

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

Non-Data-Returning	Data-Returning
push	*count*
pop	*get*
mov	*calc*
sortmess	*find*
set	*info*
rem	*getmess*
change	*getcluster*
disp	
clean	
export	
import	
read	

csv	
book	
getbook	
dump	

Like,

```
import explore_package as e
e.invoke("push num 3.14")
e.invoke("set pi num 3.14")
if e.invoke("get mess 1") == e.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 the `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`)
find	*Dictionary* / *Dictionaries* inside a *List* (`list` and `dict`)
info	*Dictionary* (`dict`)
getmess	*List* (`list`)
getcluster	*Dictionary* (`dict`)

Like,

Python Program

```
import explore_package as e

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

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

# Getting Explore's information
information = e.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': 2.0, 'Codename': 'Terrific', 'License': 'GNU General Public License v3.0', 'Author':
'Shreyas Sable', 'Repository': 'https://www.github.com/KILLinefficiency/Explore'}

pi
Terrific
```

Using Formatted Strings

The user can get a data item into the Python Program from either the **Mess** or the **Cluster** 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 handy.

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

Like,

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

The output on running would be:

```
1. 9.8

Key : Value

name : Explore
```

run()

`run()` lets the user to 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

```
import explore_package as e
e.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

```
import explore_package as e
e.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 and contact me at shreyas.sable2166@gmail.com.

Shreyas Sable