Insights
Powered by PA

Python Scripting Guide

July 2024 | Version 10.3

Product Release Date 2024-07-17

Revision Date 2024-07-17

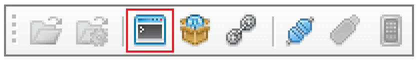
Cellebrite

Contents

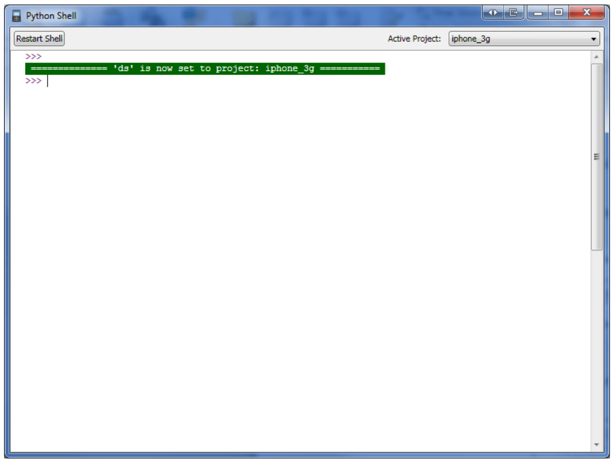
1. Getting started

Inseyets.PA enables an API based on the Physical Analyzer python shell to access, modify, and add data in various aspects.

To open the python shell window, press the Python shell button in the application tool bar, or select **Python** > **Python shell** from the application menu.

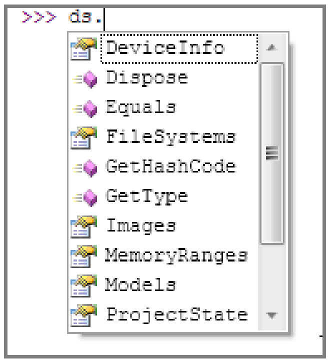


The shell window is displayed.



```
>>>
============== 'ds' is now set to project: iphone_3g ===========
>>> 
```

Since the python shell is based on the IronPython 2.6 interface, it enables autocompletion for the members of each object.



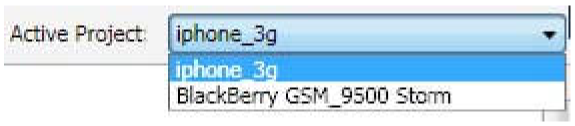
```
>>> ds.
```

2. The DataStore

The **DataStore** object contains access to all the data containers in a single project opened in Inseyets.PA. It is accessible in the shell window as the variable **ds**.

```
>>> ds
DataStore for device IPHONE_PHYSICAL (2 file systems (31772 nodes), 9842
models)
```

2.1. Selecting the active DataStore



Each opened project has its own **DataStore** instance. If more than one project was opened in Inseyets.PA, select the active **DataStore** in the **Active Project** menu at the top right:

A message appears in the shell confirming the change of **DataStore**:

```
============== 'ds' is now set to project: BlackBerry GSM_9500 Storm

===========

>>> ds
DataStore for device BB9500 (3 file systems (1083 nodes), 44 models)
```

2.2. DataStore members

The **DataStore** enables access to the members listed in the following table.

Member	Description
MemoryRanges	Collection of memory ranges storing the original input extraction and manipulations of it. Generated by different plugins.
FileSystems	Collection of Filesystems.
Models	Collection of Analyzed Data models (SMS, Contacts, etc.).
TaggedFiles	Files tagged by the DataFilesHandler option in the post-processing of a chain.

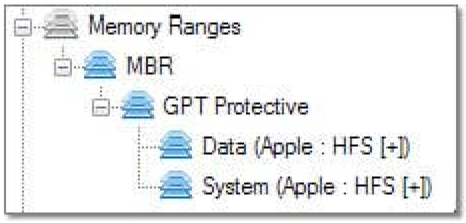
3. MemoryRange

Inseyets.PA's memory streams are stored in objects called **MemoryRanges**. A **MemoryRange** inherits from the **MemoryStream** object and therefore has the standard functions **seek** and **read**.

3.1. MemoryNode

A **MemoryNode** is a **MemoryRange** with an assigned name. It is primarily used when adding a **MemoryRange** to the **DataStore**'s **MemoryRanges** collection (a name is required for the project tree).

3.2. Getting a MemoryNode from the DataStore



MemoryNode objects are stored in their corresponding hierarchical organization – as appears in the project tree. **MemoryNode** names are not required to be unique; two **MemoryNodes** in the same hierarchy can have the same name. They can be accessed in any of the following ways:

- Direct index access from the collection:

```
>>> ds.MemoryRanges[0]
MemoryNode 'MBR' (8120172544b in 1 chunks), 1 child
>>> ds.MemoryRanges[0].Children[0]
MemoryNode 'GPT Protective' (8120168448b in 1 chunks), 2 children
>>> ds.MemoryRanges[0].Children[0].Children [1]
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
```

- Using **GetByName** function – returns an array of all the **MemoryNodes** with the input name:

```
>>> ds.MemoryRanges[0]
MemoryNode 'MBR' (8120172544b in 1 chunks), 1 child
>>> ds.MemoryRanges.GetByName ("GPT Protective")
Array[MemoryNode]((MemoryNode 'GPT Protective' (8120168448b in 1 chunks), 2
children))
>>> res = _
>>> res.Length
1
>>> mem = res [0]
>>> mem
MemoryNode 'GPT Protective' (8120168448b in 1 chunks), 2 children
```

» Using the property **ds.MemoryRanges.All** which returns a collection of all the **MemoryNodes**, regardless of their hierarchical position:

```
>>> ds.MemoryRanges[0]
MemoryNode 'MBR' (8120172544b in 1 chunks), 1 child
>>> ds.MemoryRanges.All
<Data.Store.MemoryRangeCollection+<get_All>d__6 object at 0x000000000000002D
[Data.Store.MemoryRangeCollection+<get_All>d__6]>
>>> res = _
>>> for mem in res:
print mem
MemoryNode 'MBR' (8120172544b in 1 chunks), 1 child
MemoryNode 'GPT Protective' (8120168448b in 1 chunks), 2 children
MemoryNode 'System (Apple : HFS [+])' (713052160b in 1 chunks), 0 children
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
```

3.3. Accessing the data in a MemoryRange

Since a **MemoryRange** inherits from the standard **MemoryStream**, the **Position** property stores the current position inside the stream:

```
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> mem.Position
1024L
>>> hex (_)
'0x400L'
```

Use the seek and read functions to get buffers with data from the **MemoryRange**:

```
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> mem.Position
0L
>>> mem.seek (0x400)
>>> mem.Position
1024L
>>> buf = mem.read (0x400)
>>> hex (len (buf))
'0x400'
```

3.4. Chunk

A **Chunk** object represents a specific continuous range inside a memory range and is constructed of three key properties: **BaseStream** – the **MemoryRange** to which this **Chunk** instance refers, **Offset** – the start offset in the **BaseStream** of the chunk, and **Length** – the length of the chunk. A **Chunk** is constructed in the following way:

```
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> offset = 0x400
>>> length = 0x400
>>> Chunk (mem, offset, length)
Chunk (MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0
children[0x400:0x800] (0x400b)
>>> ch = _
>>> ch.BaseStream
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> ch.Offset
1024L
>>> ch.Length
1024L
```

Chunks are used primarily when generating new **MemoryRanges**.

3.5. Generating a new MemoryRange

A new **MemoryRange** can be generated in two ways:

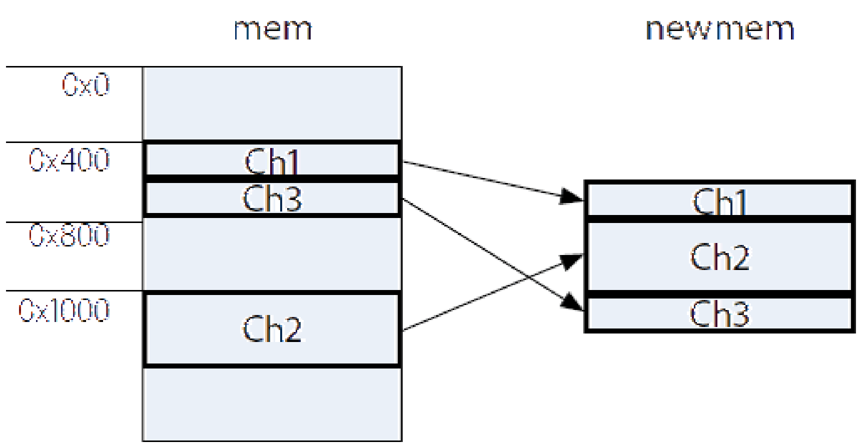
- Generating a **MemoryRange** which is a continuous subrange of a single **MemoryRange,** using the **GetSubRange** function:

```
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> offset = 0x400
>>> length = 0x400
>>> mem.GetSubRange (offset, length)
MemoryRange (1024 bytes in 1 chunks)
```

- Generating a new **MemoryRange** from one or more original **MemoryRanges**, where only fragments of the original **MemoryRanges** are considered and in any order required. Define each such fragment as a **Chunk** object and create a list of chunks in the desired order to create a new **MemoryRange**:

```
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> ch1 = Chunk (mem, 0x400, 0x200)
>>> ch2 = Chunk (mem, 0x1000, 0x400)
>>> ch3 = Chunk (mem, 0x600, 0x200)
>>> ## Append the chunks into a list according to the desired order
>>> chunks = []
>>> chunks.append (ch1)
>>> chunks.append (ch2)
>>> chunks.append (ch3)
>>> ## Construct a MemoryRange from the chunks list
>>> MemoryRange (chunks)
MemoryRange (2048 bytes in 3 chunks)
>>> ## Length should be 0x400 + 0x200 + 0x200 = 0x800
>>> newmem = _
>>> hex (newmem.Length)
'0x800L'
```

The creation of the new range in term of **Chunks** is displayed in the following scheme.



3.6. Generating a MemoryNode and adding it to the DataStore

A **MemoryNode** is a **MemoryRange** with an assigned name. After we have constructed a **MemoryRange**, we can also construct a **MemoryNode**. This is required if we want to add the newly created range into the **DataStore**.

```
>>> ## We have our new MemoryRange
>>> newmem
MemoryRange (2560 bytes in 3 chunks)
>>> ## Create a MemoryNode
>>> memnode_name = "MyMemoryNode"
>>> MemoryNode (memnode_name, newmem)
MemoryNode 'MyMemoryNode' (2560b in 3 chunks), 0 children
>>> ## Or, alternatively
>>> MemoryNode (memnode_name, MemoryRange (chunks))
MemoryNode 'MyMemoryNode' (2560b in 3 chunks), 0 children
```

The newly created **MemoryRange** can be added to the **DataStore**'s **MemoryRanges** collection under any point in the tree.

```
>>> memnode
MemoryNode 'MyMemoryNode' (2560b in 3 chunks), 0 children
>>> ## Add the MemoryNode to the collection
>>> ds.MemoryRanges.Add (memnode)
>>> ## Or as a child to another MemoryNode
>>> mem
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 0 children
>>> mem.Children.Add (memnode)
>>> ## Get the number of children of a MemoryNode
>>> mem.Children.Count
1
>>> ## Print the names of the children nodes
>>> for child in mem.Children:

                print child.Nam
MyMemoryNode
```

A **MemoryNode**’s parent can be accessed using the **Parent** property:

```
>>> memnode
MemoryNode 'MyMemoryNode' (2560b in 3 chunks), 0 children
>>> memnode.Parent
MemoryNode 'Data (Apple : HFS [+])' (7407116288b in 1 chunks), 1 child
```

4. Special object types

In addition to the standard C# objects, Inseyets.PA introduces several objects and enums used as property or value types for filesystems and models. Before we elaborate on how to parse and construct a filesystem or models, let us review these types.

4.1. DeletedState enum

This enum is used to indicate the deleted state of an object. It has three possible values, as listed in the following table.

Type	Description
Unknown	Default value, used for elements when the deleted state is unknown.
Deleted	Used for elements which are known to be deleted.
Intact	Used for elements which are known to be intact.

The enum can be accessed in the shell as **DeletedState** with the values **DeletedState.Unknown**, **DeletedState.Deleted** or **DeletedState.Intact** correspondingly.

4.2. TimeStamp object

The **TimeStamp** object is used to represent date time for filesystem **Nodes** and for content models which have a **TimeStamp** property. There are several ways to initialize a **TimeStamp** object.

» The static function **FromUnixTime** receives the standard UNIX int32 **TimeStamp** format (seconds from epoch 1/1/1970 0:00 (UTC+0)) and returns a **TimeStamp** object:

```
>>> TimeStamp.FromUnixTime (0x4e5f4990)
9/1/2011 9:00:00 AM (UTC+0)
```

» The static function **FromFileTime** receives the int64 FILETIME format and returns a **TimeStamp** object:

```
>>> TimeStamp.FromFileTime (0x1cc68912431a000L)
9/1/2011 10:23:07 AM (UTC+0)
```

» Using an already initialized standard C# DateTime object:

```
>>> ## If we have an initialized DateTime
>>> DateTime (2011, 9, 1, 9, 0, 0)
<System.DateTime object at 0x0000000000000038 [9/1/2011 9:00:00 AM]>
>>> ## Init TimeStamp from it
>>> dt = _
>>> TimeStamp (dt)
9/1/2011 9:00:00 AM
>>> ## Or with indication we know the timezone (here UTC+0)
>>> TimeStamp (dt, True)
9/1/2011 9:00:00 AM (UTC+0)
```

» From an ISO timestamp format (yyyy-mm-dd hh:mm:ss+/-tz) and using **System.Convert.ToDateTime**:

```
>>> TimeStamp (System.Convert.ToDateTime ('2011-09-01 09:00:00-5'), True)
9/1/2011 9:00:00 AM (UTC-5)
```

4.3. MetaData object

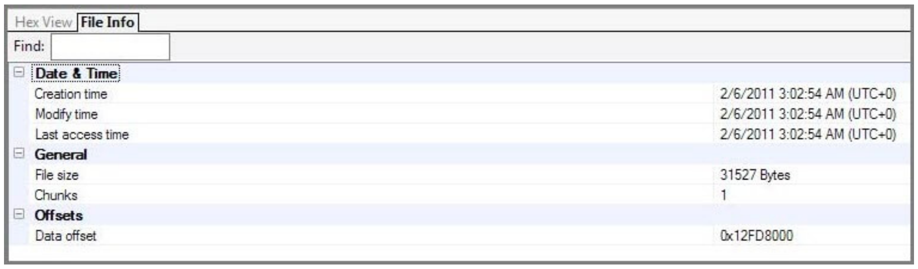
The **MetaData** object allows the construction of metadata pairings of either key-value or key-value-group.

The group indication allows us to distinguish between different groups of metadata key-value pairs; e.g. distinguishing metadata from the filesystem tree (such as datetime, size, attributes) from metadata that was parsed within the file (EXIF information, quicktime or mp4 tags).

We will later describe which objects have a **MetaData** property and how they can be accessed, but after we have access to such an object – let us demonstrate the updating process:

```
>>> ## We have access to a MetaData object md
>>> md
<Data.Core.MetaData object at 0x000000000000002B>
>>> ## Add key-value pairing
>>> md.Add ("MyKey", "MyValue")
MyKey: MyValue
>>> ## Add key-value-group pairing
>>> md.Add ("MyGrKey", "MyGrValue", "MyGroup")
MyGrKey (MyGroup): MyGrValue
```

Filesystem file nodes are viewable in the following way:



Date & Time	
Creation time	2/6/2011 3:02:54 AM (UTC+0)
Modify time	2/6/2011 3:02:54 AM (UTC+0)
Last access time	2/6/2011 3:02:54 AM (UTC+0)
General	
File size	31527 Bytes
Chunks	1
Offsets	
Data offset	0x12FD8000

5. Node, file, directory and link objects

As with **MemoryNodes**, filesystem **Node** objects are more flexible in use than an actual filesystem allows. For example, two nodes stored in the same directory can have the same name and a file **Node** can have child nodes such as a directory or link. This flexibility allows for very comfortable parsing, as we will demonstrate.

5.1. Node types

A **Node** object is defined abstract and therefore may be used with any **Node** type listed in the following table.

Type	Description
Unknown	Default value, used for elements when the deleted state is unknown.
Deleted	Used for elements which are known to be deleted.
Intact	Used for elements which are known to be intact.

The **Node** types are contained in the **NodeType** enum:

```
>>> NodeType
<type 'NodeType'>
>>> NodeType.Directory
Data.Files.NodeType.Directory
>>> NodeType.File
Data.Files.NodeType.File
>>> NodeType.Link
Data.Files.NodeType.Link
>>> NodeType.Embedded
Data.Files.NodeType.Embedded
```

A **NodeType** is a flag enum - a **Node** can be both **File** and **Embedded** at the same time:

```
>>> NodeType.Embedded | NodeType.File
<enum Data.Files.NodeType: File, Embedded>
```

5.2. Creating new Nodes

A new File or Directory **Node** can be created in the following ways:

- Using the **Node** constructor with an assigned **NodeType**:

```
>>> f = Node (NodeType.File)
>>> f
File '/' (0b)
## Or with a name and Node type
>>> f = Node ("MyFile", NodeType.File)
>>> f
File '/MyFile' (0b)
>>> d = Node (NodeType.Directory)
>>> d
Directory '/' (0 children)
>>> f = Node ("MyDir", NodeType.Directory)
>>> f
Directory '/MyDir' (0 children)
```

- Using the specific constructor for each **Node** type:

```
>>> f = File ()
>>> f = File ("MyFile")
>>> f
File '/MyFile' (0b)
>>> d = Directory ()
>>> d = Directory ("MyDir")
>>> d
Directory '/MyDir' (0 children)
```

- The latter is also the way to initialize a new **Link Node**:

```
>>> l = Link ("MyLink")
>>> l
Symbolic link MyLink ->
```

5.3. Node properties

All **Node** types have properties inherited from the abstract **Node** object that are used frequently when constructing a filesystem.

Property	Type	Description
Name	string	**Node** name. Can also be set upon initialization, as detailed above.
Deleted	DeletedState	Indication of the state of a **Node** (intact, deleted or unknown).
AccessTime	TimeStamp	**Node** last access date time.
CreationTime	TimeStamp	**Node** creation date time.
ModifyTime	TimeStamp	**Node** last modification date time.
DeletedTime	TimeStamp	**Node** deletion date time.
Data	MemoryRange	Data MemoryRange related to this **Node** (e.g. a file's data).
MetaData	MetaData	Container for any additional metadata on each **Node**.
Children	NodeCollection	Collection of children nodes related to this **Node** (e.g. files inside a directory).

The **NodeCollection** type is a **Nodes** collection with several additional properties we address later in the FileSystem section.

See Special object types (on page 18) regarding **TimeStamp**, **DeletedState**, and **MetaData** object types.

5.4. Link path

Use the **LinkPath** string property to update the **Link**'s symbolic link path:

```
>>> l = Link ("MyLink")
>>> l.LinkPath = '/dir1/dir1_1/dir1_1_2/file'
>>> l
Symbolic link MyLink -> /dir1/dir1_1/dir1_1_2/file
>>> ## Or, directly during init
>>> l = Link ("MyLink", "/dir1/dir1_1/dir1_1_2/file")
>>> l
Symbolic link MyLink -> /dir1/dir1_1/dir1_1_2/file
```

6. FileSystem

A **FileSystem** object constructs a single filesystem that was either received as an input folder or zip file, or parsed from a dump. A **FileSystem** object contains a hierarchical structure of **Node** objects that represent the structure of the filesystem.

A **FileSystem** object can be added to the **DataStore** after it has been initialized, while its nodes and their hierarchy may be added at a later phase, although we recommend that you construct the entire filesystem and only then to add it to the **DataStore**: as the addition of each **Node** causes a GUI refresh and therefore prolongs the filesystem construction time.

6.1. Creating a new FileSystem and adding it to the DataStore

Use the following code to create a new **FileSystem** object:

```
>>> ## Create a new FileSystem
>>> fs = FileSystem ()
>>> fs.Name = "My FileSystem"
>>> ## Or, init the filesystem already with a name
>>> fs = FileSystem ("My FileSystem")
>>> fs
FileSystem 'My FileSystem' (0 nodes)
>>> ## Add the new FileSystem to the DataStore
>>> ds.FileSystems.Add (fs)
>>> ds.FileSystems
FileSystemCollection {"System (Apple : HFS [+])", "Data (Apple : HFS [+])",
"My FileSystem"}
```

6.2. Accessing a FileSystem in the DataStore

Similar to **ds.MemoryRanges**, the property **ds.FileSystems** allows access using a direct index (**ds.FileSystems [0]**), the **GetByName** function or **ds.FileSystems.All** property. Usage and results are the same as described in MemoryRange (on page 9).

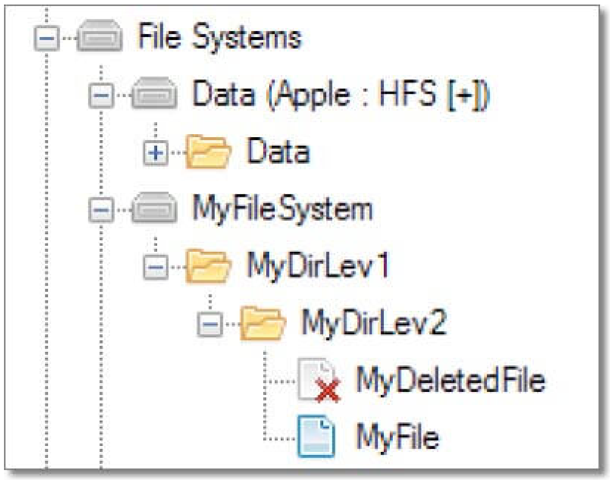
6.3. Constructing an hierarchical FileSystem

In the code below, let us summarize the flow for creating hierarchies in a **FileSystem** object:

```
>>> ## Create a new FileSystem
>>> fs = FileSystem ("My FileSystem")
>>> fs
FileSystem 'My FileSystem' (0 nodes)
>>> ## Generate a file
>>> f = File ()
>>> f.Name = "MyFile"
>>> ## Add a timestamp
>>> f.CreationTime = TimeStamp.FromUnixTime (0x4e5f4990)
>>> f.CreationTime
9/1/2011 9:00:00 AM (UTC+0)
>>> ## Add some metadata to f
>>> f.MetaData.Add ("Key1", "Val1")
Key1: Val1
>>> f.MetaData.Add ("Key2", "Val2", "Group")
Key2 (Group): Val2
>>> ## Get the original memory range and create a subrange of it as f's Data
>>> mem = ds.MemoryRanges [0]
>>> f.Data = mem.GetSubRange (0, 0x1000)
>>> ## See what f contains
>>> f
File '/MyFile' (4096b)
>>> ## Create a directory
>>> dir1 = Directory ("MyDirLev1")
>>> ## Create another directory
>>> dir2 = Directory ("MyDirLev2")
>>> ## add dir1 as child of fs and dir as child of dir1
>>> fs.Children.Add (dir1)
>>> dir1.Children.Add (dir2)
>>> ## Add the file under dir2
>>> dir2.Children.Add (f)
>>> ## Create another file under dir2 and mark as deleted
>>> f2 = File ("MyDeletedFile")
>>> f2.Deleted = DeletedState.Deleted
>>> dir2.Children.Add (f2)
>>> ## Finally, add fs to the filesystems collection in DataStore
>>> fs
```

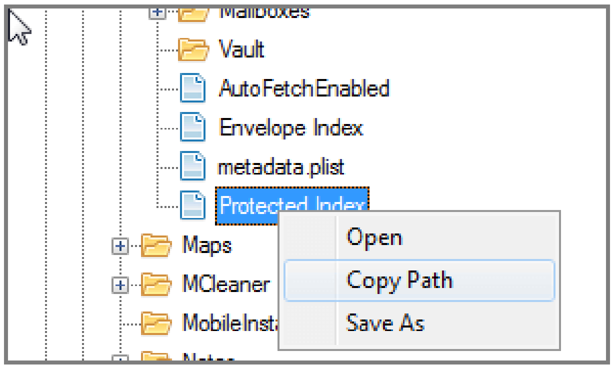
```
FileSystem 'MyFileSystem' (4 nodes) [MyDirLev1]
>>> ds.FileSystems.Add (fs)
```

This results in the following filesystem structure in the **DataStore** tree.



6.4. Getting a Node's full path from the DataStore tree

When a specific file requires further inspection using the python shell, its full path in the filesystem can be retrieved from the **DataStore** tree in Inseyets.PA by right-clicking on the **Node** and selecting **Copy Path**. This copies the full path of the **Node** to the clipboard.



For tagged files this can be done directly from the tree part of tagged files.

For each **Node** in the **FileSystem**, the path is accessible via the **AbsolutePath** property.

```
>>> f.AbsolutePath
'/Data/mobile/Library/com.apple.itunesstored/itunesstored2.'
```

6.5. Accessing Nodes in a FileSystem

There are several ways to get specific Nodes from a **FileSystem**. These include accessing a single Node, or retrieving a collection of **Nodes** that match a search result.

- Using direct indices and the **.Children** property:

```
>>> ## Start with a complex filesystem
>>> fs
FileSystem 'Data (Apple : HFS [+])' (15127 nodes) [Data]
>>> ## Get nodes from it
>>> fs.Children [0]
Directory '/Data' (28 children) [root, db, empty, vm, MobileDevice, HFS+
Private Data, .HFS+ Private Directory Data
, lib, tmp, folders, Managed Preferences, cache, lock, audit, Keychains,
stash, local, ea, logs, backups, keybags, wireless, log, mobile, spool, msgs,
preferences, run]
>>> fs.Children[0].Children[1]
Directory '/Data/db' (4 children) [dhcpclient, launchd.db, PanicReporter,
timezone]
>>> fs.Children[0].Children[1].Children[2]
Directory '/Data/db/PanicReporter' (1 child) [current.panic]
>>> fs.Children[0].Children[1].Children[2].Children[0]
Symbolic link current.panic -> /Library/Logs/CrashReporter/Panics/2011-02-12-
200628.panic.plist
```

- Using the **Node**'s full path in direct access:

```
>>> ## We want to get the following file
>>> ## /Data/root/Library/AddressBook/AddressBook.sqlitedb
>>> fs ['/Data/root/Library/AddressBook/AddressBook.sqlitedb']
File '/Data/root/Library/AddressBook/AddressBook.sqlitedb' (212992b)
>>> ## but if Node does not exist we get an exception
>>> fs ['/Data/root/Library/AddressBook/AddressBook.sqlitedb1234']
Traceback (most recent call last):
 File "<string>", line 1, in <module>
Exception: File /Data/root/Library/AddressBook/AddressBook.sqlitedb1234 not
found.
```

» Using the **Node**'s full path and **GetByPath** function:

```
>>> ## Same as before only using GetByPath
>>> fs.GetByPath ('/Data/root/Library/AddressBook/AddressBook.sqlitedb')
File '/Data/root/Library/AddressBook/AddressBook.sqlitedb' (212992b)
>>> ## Here, if the Node does not exist we get None as a result
>>> res = fs.GetByPath
('/Data/root/Library/AddressBook/AddressBook.sqlitedb1234')
>>> res == None
True
```

» Using the **Node**'s full path and **TryGetByPath** function:

```
>>> ## TryGetByPath output is a tuple with indication of success
>>> ## Success will output True and the Node
>>> fs.TryGetByPath ('/Data/root/Library/AddressBook/AddressBook.sqlitedb')
(True, File '/Data/root/Library/AddressBook/AddressBook.sqlitedb' (212992b))
>>> ## Failure will output False and None
>>> fs.TryGetByPath
('/Data/root/Library/AddressBook/AddressBook.sqlitedb1234')
(False, None)
```

» Using the **GetAllNodes** function, which returns a collection with all the filesystem **Node** objects:

```
>>> ## GetAllNodes will return a collection with all the nodes in the
filesystem
>>> fs.GetAllNodes ()
<Data.Files.NodeCollection+<GetAllNodesInternal>d__0 object at
0x000000000000002F [Data.Files.NodeCollection+<GetAllNodesInternal>d__0]>
>>> ## print nodes absolute path
>>> for Node in _:
 print Node.AbsolutePath
/Data
/Data/root
/Data/root/Library
/Data/root/Library/Lockdown
/Data/root/Library/Lockdown/pair_records
/Data/root/Library/Lockdown/pair_records/30132796-125209691221841904.plist
/Data/root/Library/Lockdown/pair_records/30132832513909728326172028.plist
/Data/root/Library/Lockdown/pair_records/30118105-180496331224419632.plist
/Data/root/Library/Lockdown/pair_records/3012518617774694460012408.plist
/Data/root/Library/Lockdown/pair_records/9016926980658937761372207.plist
/Data/root/Library/Lockdown/pair_records/30120955601226777262221504.plist
/Data/root/Library/Lockdown/device_public_key.pem
```

6.5.1. FileSystem search

The **FileSystem** object enables a **Search** function. It receives either a file name, full path or a regular expression pattern and returns a collection of nodes:

```
>>> ## Search all nodes AddressBook.sqlite and print the AbsolutePath of the
results
>>> for f in fs.Search ('AddressBook\.sqlite'):

            print f.AbsolutePath
/Data/root/Library/AddressBook/AddressBook.sqlitedb
/Data/mobile/Library/AddressBook/AddressBook.sqlitedb
>>> ## Search all nodes with AddressBook in their name
>>> for f in fs.Search ('AddressBook'):

            print f.AbsolutePath
/Data/root/Library/AddressBook
/Data/root/Library/AddressBook/AddressBook.sqlitedb
/Data/mobile/Library/AddressBook
/Data/mobile/Library/AddressBook/AddressBookImages.sqlitedb
/Data/mobile/Library/AddressBook/AddressBook.sqlitedb
/Data/mobile/Library/Caches/com.apple.IconsCache/com.apple.MobileAddressBook_
defaultRole-defaultIcon-0
>>> ## Search all quicktime movies inside the DCIM directory using regexp
pattern
>>> for f in fs.Search ('/.*?/mobile/Media/DCIM/.*?/.*?.MOV'):

            print f.AbsolutePath
/Data/mobile/Media/DCIM/102APPLE/IMG_2634.MOV
/Data/mobile/Media/DCIM/102APPLE/IMG_2575.MOV
/Data/mobile/Media/DCIM/103APPLE/IMG_3032.MOV
/Data/mobile/Media/DCIM/103APPLE/IMG_3148.MOV
/Data/mobile/Media/DCIM/100APPLE/IMG_0040.MOV
```

7. SQLiteParser

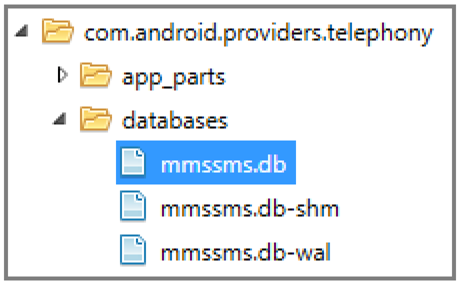
The SQLiteParser library is a set of classes that aid in reading database SQLite files. It allows us to read from both the main db file and the attached WAL file. The parser can read intact and deleted records.

To start using the SQLiteParser library, we must import it.

```
>>> import SQLiteParser
```

7.1. Database class

To start working with a SQLite database, we must initialize a Database class. We use a given node that we know is a SQLite database file.



```
>>> node = ds.FileSystems[0][PATH_OF_FILE]
>>> db = SQLiteParser.Database.FromNode(node)
>>> db
<SQLiteParser.Database object at 0x000000000000004D [[SQliteDb Tables(28):
[spam_filter,spam_addr,spam_part,spam_rate,spam_drm,spam_sms,android_
metadata,pdu,sqlite_sequence,addr,part,rate,drm,sms,raw,attachments,sr_
pending,cmas,wpm,canonical_addresses,threads,pending_
msgs,mychannels,words,words_content,words_segments,words_segdir,spam_pdu]]]>
```

If the database has an adjacent WAL file, the parser reads it along with the main sqlite db.

Let us look at some of the properties of a database:

```
>>> db.DBNode ##gives us the node from which the Database is created
File '/Root/data/com.android.providers.telephony/databases/mmssms.db'
(249856b)
>>> db.DBWalNode ##gives us the WAL node of the same db
File '/Root/data/com.android.providers.telephony/databases/mmssms.db-wal'
(4120032b)
>>> db.Tables ## gives us an array of all the tables names (not including
sqlite_master)
Array[str](('spam_filter', 'spam_addr', 'spam_part', 'spam_rate', 'spam_drm',
'spam_sms', 'android_metadata', 'pdu', 'sqlite_sequence', 'addr', 'part',
'rate', 'drm', 'sms', 'raw', 'attachments', 'sr_pending', 'cmas', 'wpm',
'canonical_addresses', 'threads', 'pending_msgs', 'mychannels', 'words',
'words_content', 'words_segments', 'words_segdir', 'spam_pdu'))
```

7.1.1. Records and reading intact records

To read all the intact records from a certain table, we can run over all the records in a certain table:

```
>>> for record in db["addr"]:
print record
{_id: 20, msg_id: 4, contact_id: , address: +972544358345, type: 151, charset:
106}
{_id: 19, msg_id: 4, contact_id: , address: insert-address-token, type: 137,
charset: 106}
{_id: 16, msg_id: 3, contact_id: , address: +972544358345, type: 151, charset:
106}
{_id: 15, msg_id: 3, contact_id: , address: insert-address-token, type: 137,
charset: 106}
{_id: 12, msg_id: 1, contact_id: , address: +972542547036, type: 151, charset:
106}
{_id: 11, msg_id: 1, contact_id: , address: insert-address-token, type: 137,
charset: 106}
```

Now for each record, we can look at the information it stores:

```
>>> record
{_id: 20, msg_id: 4, contact_id: , address: +972544358345, type: 151, charset:
106}
>>> record["address"]
Field(+972544358345)
>>> record["address"].Value
'+972544358345'
>>> record["address"].Source
List[Chunk]([Chunk (MemoryRange (249856 bytes in 2 chunks)[0x6F5B:0x6F68]
(0xDb)])
>>> record.Deleted
Data.Files.DeletedState.Intact
```

7.1.2. TableSignature and reading deleted records

Each table has a default set of signatures defined in the sqlite_master table, but that set of signatures is too general (mostly it only states if a column is Int / Text / Blob).

We must define a more definitive set of signatures to minimize the extraction of false positives. Each content size has a serial type according to the following table.

Serial Type	Content Size	Meaning
0	0	Null.
1	1	8-bit twos-complement integer.
2	2	Big-endian 16-bit twos-complement integer.
3	3	Big-endian 24-bit twos-complement integer.
4	4	Big-endian 32-bit twos-complement integer.
5	6	Big-endian 48-bit twos-complement integer.
6	8	Big-endian 64-bit twos-complement integer.
7	8	Big-endian IEEE 754-2008 64-bit floating point number.
8	0	Integer constant 0. Only available for schema format 4 and higher.
9	0	Integer constant 1. Only available for schema format 4 and higher.
10, 11		Not used. Reserved for expansion.
N≥12 and even	(N-12)/2	A BLOB that is (N-12)/2 bytes in length.
N≥13 and odd	(N-13)/2	A string in the database encoding and (N-13)/2 bytes in length. The nul terminator is omitted.

And we use the enum listed in the following table.

Serial Type	Content Size
0	SQLiteParser.Tools.SignatureType.Null
1	SQLiteParser.Tools.SignatureType.Byte
2	SQLiteParser.Tools.SignatureType.Short
3	SQLiteParser.Tools.SignatureType.Int24
4	SQLiteParser.Tools.SignatureType.Int
5	SQLiteParser.Tools.SignatureType.Int48
6	SQLiteParser.Tools.SignatureType.Long
7	SQLiteParser.Tools.SignatureType.Float
8	SQLiteParser.Tools.SignatureType.Const0
9	SQLiteParser.Tools.SignatureType.Const1
N≥12 and even	SQLiteParser.Tools.SignatureType.Text
N≥13 and odd	SQLiteParser.Tools.SignatureType.Blob

If we do not know the signature type of our columns, we can use the following code:

```
>>> signatures = SQLiteParser.Tools.GetColumnStatistics(db, "addr")
>>> for name in signatures.Keys:

                name, signatures[name]
('_id', List[SignatureType]([SQLiteParser.Tools+SignatureType.Null]))
('msg_id', List[SignatureType]([SQLiteParser.Tools+SignatureType.Byte,
SQLiteParser.Tools+SignatureType.Const1]))
('contact_id', List[SignatureType]([SQLiteParser.Tools+SignatureType.Null]))
('address', List[SignatureType]([SQLiteParser.Tools+SignatureType.Text]))
('type', List[SignatureType]([SQLiteParser.Tools+SignatureType.Short]))
('charset', List[SignatureType]([SQLiteParser.Tools+SignatureType.Byte]))
```

We now review how to create a table signature so that we can read deleted records:

```
>>> ts = SQLiteParser.TableSignature("addr")
>>> SQLiteParser.Tools.AddSignatureToTable(ts, "type",
SQLiteParser.Tools.SignatureType.Short)
>>> SQLiteParser.Tools.AddSignatureToTable(ts, "address",
SQLiteParser.FieldType.Text, SQLiteParser.FieldConstraints.NotNull)
```

We create the TableSignature object and then start adding the relevant signatures to it. There are 2 ways of adding signatures: we can add a list of the SignatureTypes, or we can tell what our field type is and decide that it is never null.

Important! When defining the TableSignature, restricting the signatures too much might cause us to lose deleted records. We recommend that you start with an empty TableSignature and then add signatures to reduce our false positives.

After we have a TableSignature, we can read the deleted records:

```
>>> for record in db.ReadTableRecords(ts, True): ##the True parameter decides
if we extract deleted

            Record
```

7.2. DBNull

When a specific column in a record has no value, it receives a value of DBNull. It is therefore important to check if a column's value is DBNull before trying to use it:

```
>>> from System.Convert import IsDBNull
>>> IsDBNull(record["type"].Value)
False
```

7.3. Code samples – BeeTalk and Mozilla Firefox

Code samples are available in the Inseyets.PA installation directory under PythonSamples.

8. Content models

To insert decoded content into the Analyzed Data part in the **DataStore** tree, we introduce the abstract **Model** class and its children. Each class that inherits from **Model** has the common property **Deleted**, which – much like for filesystem **Node** objects – is set with values from the **DeletedState** enum previously mentioned in the **FileSystem** section.

We distinguish between two types of model classes: model classes inserted in the analyzed data section of the **DataStore** tree (Contacts, SMS, Emails) and other model classes that are dependent upon other models (e.g. phone number, email address, street address, etc.).

Not all properties need to be set when a model object is used. For example, one can generate an SMS model which has only body and sender number information (that is, the to and timestamp information are not set).

8.1. List of dependent and independent Models

The independent **Models** that are added to the **DataStore** tree are listed in the following table.

Contact	UserAccount	VoiceMail
SMS	CalendarEntry	Password
Email	Journey	InstalledApplication
MMS	Cookie	ApplicationUsage
Note	VisitedPage	DictionaryWord
Chat	WebBookmark	SharedFile
Location	BluetoothDevice	Map
SearchedItem	WirelessNetwork	Notification
InstantMessage	CarvedString	PoweringEvent

The dependent **Models** (added as properties to other models) are listed in the following table.

PhoneNumber	Attachment	UserID
StreetAddress	WebAddress	Party
ContactPhoto	Organization	Coordinate
EmailAddress		

8.2. Adding independent models into the DataStore

After generating any model from the independent list above, adding it to the **DataStore** is very simple:

```
>>> ## Create a contact and update the name only
>>> cont = Contact ()
>>> ## Adding a Name Field (Fields are explained in the next section)
>>> cont.Name.Value = "John Doe"
>>> ## Add contact to the DataStore
>>> ds.Models.Add (cont)
>>> ## If we have a list of contacts we can use AddRange
>>> contacts = []
>>> contacts.append (cont)
>>> ds.Models.AddRange (contacts)
```

8.3. Field, MultiField, SingleModelField, MultiModelField

Apart from **DeletedState**, all model properties are constructed in a **Field** object, which is a generic type for various value options. This method enables the decoding of content while maintaining Inseyets.PA highlights to the location from which the value was extracted. For example, a Unicode string as hex may appear in data 6100620063006400650 where the decoded value is the string abcde. The **Field** class allows us to keep information about both value (decoded string) and source (original buffer). A field has the properties listed in the following table.

Property	Type	Description
Value	<>	Generic type determined for each Field type property.
Source	MemoryRange	Source from which the value was decoded.

Some model properties are defined as **MultiField**, as they receive more than one value (for example – contact notes). Updating a **MultiField** object is done using the **Add** function. This also applies to the **SingleModelField** and **MultiModelField**, where the type is itself a model.

For each model described, the initialized value property type is set using the notation Field<>, MultiField <>, SingleModelField<> or MultiModelField<> where the type of the value is stated inside the brackets.

8.4. DataField

DataField is used for a model property that has the value **MemoryRange**. It has only a **Source** property and no **Value**.

8.5. NodeField

The NodeField contains information about a filesystem node. It is used by models that have filedata property, for example map model stores its image inside the NodeField. Its properties: are listed in the following table.

Property	Type	Description
Value	Node	

8.6. ContactEntry

A **ContactEntry** can be a phone number, an email or any other method of communication (apart from addresses) detailed for a contact. Its properties are listed in the following table.

Property	Type	Description
Value	Field<string>	Entry value (phone number or email string).
Category	Field<string>	Entry category (work, home, etc.).
Domain	Field<string>	Entry domain (phone number, email, web address, etc.).

Since **ContactEntry** has inherited classes for specific types, the Domain property is rarely used (since it is defined with the constructor of each specific entry).

The Category property receives any string value, although some values add an icon next to the entry in the contacts table: General, Work, Home, Car, Mobile, Fax, Pager, Other, Unknown.

The objects that inherit **ContactEntry** are:

- PhoneNumber – used for phone numbers.
- EmailAddress – used for email addresses.
- WebAddress – used for a contact's webpage addresses, etc.
- UserID – used for a contact's various user IDs (e.g. Skype, BlackBerry PIN, etc.).

Since creating a **ContactEntry** is the same for all the above, an example is provided under the discussion of the Contact (on page 45)model.

8.7. StreetAddress

For a contact’s street addresses we apply this model. It has many properties but it is not mandatory to set all of them with values.

Property	Type	Description
Street1	Field<string>	Street information.
Street2	Field<string>	Additional street information.
HouseNumber	Field<int>	
City	Field<string>	
State	Field<string>	
Country	Field<string>	
PostalCode	Field<string>	Address Postal Code or ZIP.
POBox	Field<string>	
Neighborhood	Field<string>	
Category	Field<string>	Same values as ContactEntry categories.

8.8. Organization

The Organization model is used to describe a contact’s organization and position details.

Property	Type	Description
Name	Field<string>	Oragnization name.
Position	Field<string>	Contact’s position in the organization.

8.9. ContactPhoto

The ContactPhoto model can contain a profile or address-book photo.

Property	Type	Description
Name	Field<string>	Filename (if exists).
PhotoNode	NodeField	Photo MemoryRange.

Data property has a special type called **DataField**. This field only receives **MemoryRange** as input in the **Source** property (the attachment data). It has no **Value** property.

8.10. Contact

After detailing the submodels that construct the contact, the following table lists its properties.

Property	Type	Description
Name	Field<string>	Contact Name.
Photos	MultiModelField<ContactPhoto>	Contact Photos.
Entries	MultiModelField<ContactEntry>	Contact entries collection.
Addresses	MultiModelField<StreetAddress>	Addresses collection.
Organizations	MultiModelField<Organization>	Organizations collection.
Notes	MultiField<string>	Contact Notes.
Source	Field<string>	Contact Source.
Type	Field<ContactType>	The type of contact (follower, following).
Group	Field<string>	
TimeContacted	Field<TimeStamp>	
TimeCreated	Field<TimeStamp>	
TimeModified	Field<TimeStamp>	
TimesContacted	Field<int>	

The **Source** property is usually left blank for contacts that come from the phone’s address book and is used for other contacts found (e.g. Skype or Facebook).

8.11. Contact creation flow

Let us demonstrate the creation of a contact using a simple contact record example. Assume in a filesystem each contact is stored in a different file, where the fields (values and keys) are separated by a pipe (|) delimiter.

```
>>> ## Get the file
>>> f = ds.FileSystems[0]['/contact_donald.bit']
>>> ## Read the content and display it
>>> buf = f.Data.read ()
>>> buf
'Name|Donald Duck|Mobile|+12125555555|Work|+1222666666|Address|123 Magic
Kingdom|Orlando|FL|US|Organization|Disney|'
```

We will construct a simple parser that splits the string into fields and then parses each key and the content thereafter according to the key itself – namely **Name**, **Mobile**, **Work** and **Organization** have a single value field, and **Address** has four value fields.

For each field, we keep an offset variable in the buffer that allows us the extraction of the correct subrange for the **Source** property. Here is the function that parses and creates a new contact for this example:

```
def parse_contact (buf):
    ## Split the buffer into fields
    vals = buf.split ('|')
    ## Init a new Physical Analyzer Contact object
    cont = Contact ()
    ## Mark as intact
    cont.Deleted = DeletedState.Intact
    ind = 0
    off = 0
    while ind < len (vals) :
        ## Extract key, update offset in buffer and index in vals
```

```
        key = vals [ind]
        off += len (key) +1
```

```
ind += 1
## Break the loop if the key is empty
if key == "":
        break
elif key == "Name":
        ## Extract contact name and update to PA contact
        val = vals [ind]
        cont.Name.Value = val
        cont.Name.Source = f.Data.GetSubRange (off, len (val))
        off += len (val) + 1
        ind += 1
elif key == "Mobile" or key == "Work":
        ## Extract a phone number. Init a new PhoneNumber object.
        ## Number Category is the key here - Work or Mobile
        ## A PhoneNumber is a ContactEntry and will be inserted into
        ## The Entries property of the contact.
        ## Same process will be applied to WebAddress, EmailAddress,
        ## And UserID
        val = vals [ind]
        ph = PhoneNumber ()
        ph.Deleted = cont.Deleted
        ph.Category.Value = key
        ph.Value.Value = val
        ph.Value.Source = f.Data.GetSubRange (off, len (val))
```

```
        ## Insert the phone number entry into Entries property
        cont.Entries.Add (ph)
        ind += 1
        off += len (val) + 1
elif key == "Address" :
        ## Extract a street address - Street, City, State, Country
        ## Keep the offsets updated for correct memory ranges creation
```

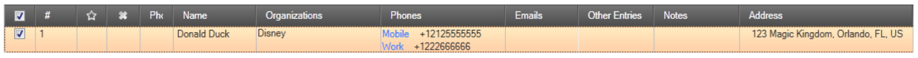
```
            addr = StreetAddress ()
            addr.Deleted = cont.Deleted
            addr.Street1.Value = vals [ind]
            addr.Street1.Source = f.Data.GetSubRange (off, len (vals [ind]))
            off += len (vals [ind]) + 1
            addr.City.Value = vals [ind+1]
            addr.City.Source = f.Data.GetSubRange (off, len (vals [ind + 1]))
            off += len (vals [ind + 1]) + 1
            addr.State.Value = vals [ind + 2]
            addr.State.Source = f.Data.GetSubRange (off, len (vals [ind +2]))
            off += len (vals [ind + 2]) + 1
            addr.Country.Value = vals [ind + 3]
            addr.Country.Source = f.Data.GetSubRange (off, len (vals[ind+3]))
            off += len (vals [ind + 3]) + 1
            ind += 4
            ## Update the contact addresses property
            cont.Addresses.Add (addr)
        elif key == "Organization":
            ## Extract an organization name
```

```
            ## Init an Organization object
            org = Organization ()
            org.Deleted = cont.Deleted
            org.Name.Value = vals [ind]
            org.Name.Source = f.Data.GetSubRange (off, len (vals [ind]))
            ind += 1
            off += len (vals [ind]) + 1
            ## Update the organizations property.
            cont.Organizations.Add (org)
    ## Return the contact
    return cont
```

When using this function, we get the following result in the shell:

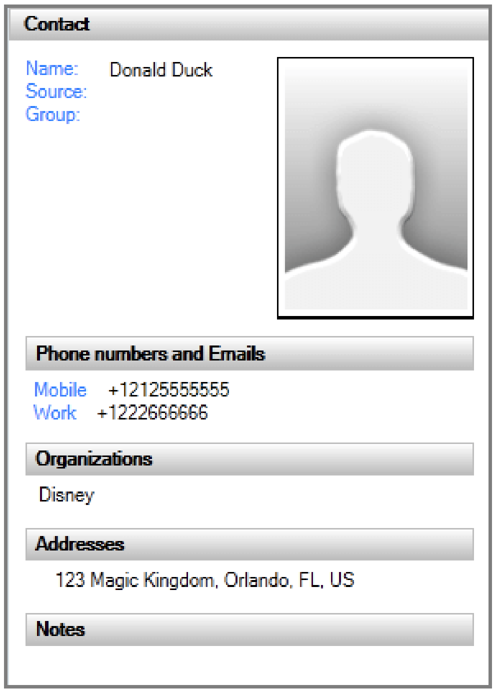
```
>>> ## Run the function
>>> buf
'Name|Donald Duck|Mobile|+12125555555|Work|+1222666666|Address|123 Magic
Kingdom|Orlando|FL|US|Organization|Disney|'
>>> parse_contact (buf)
Contact {Name: Field(Donald Duck), Entries: Field(Phone (Mobile)
'+12125555555', Phone (Work) '+1222666666'), Addresses: Field(StreetAddress
{Street1: Field(123 Magic Kingdom), City: Field(Orlando), State: Field(FL),
Country: Field(US)}), Organizations: Field(Disney)}
>>> ## Add contact to the DataStore
>>> ds.Models.Add (_)
```

After adding it to the **DataStore**, the following entry is displayed in the contacts table:



☑	#	☆	✖	Phc	Name	Organizations	Phones	Emails	Other Entries	Notes	Address
☑	1				Donald Duck	Disney	Mobile +12125555555 Work +1222666666				123 Magic Kingdom, Orlando, FL, US

Or in the detailed right-pane:



Contact

Name: Donald Duck
Source:
Group:

Phone numbers and Emails

Mobile +12125555555
Work +1222666666

Organizations

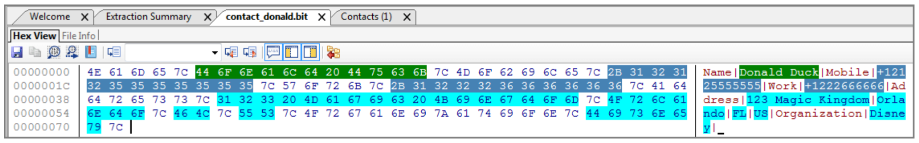
Disney

Addresses

123 Magic Kingdom, Orlando, FL, US

Notes

And since we provided **Source** properties, we can see Inseyets.PA highlights in the **Hex View** tab of the file:



```
00000000  4E 61 6D 65 7C 44 6F 6E 61 6C 64 20 44 75 63 6B 7C 4D 6F 62 69 6C 65 7C 2B 31 32 31  Name|Donald Duck|Mobile|+121
0000001C  32 35 35 35 35 35 35 35 7C 57 6F 72 6B 7C 2B 31 32 32 32 36 36 36 36 36 36 7C 41 64  25555555|Work|+1222666666|Ad
00000038  64 72 65 73 73 7C 31 32 33 20 4D 61 67 69 63 20 4B 69 6E 67 64 6F 6D 7C 4F 72 6C 61  dress|123 Magic Kingdom|Orla
00000054  6E 64 6F 7C 46 4C 7C 55 53 7C 4F 72 67 61 6E 69 7A 61 74 69 6F 6E 7C 44 69 73 6E 65  ndo|FL|US|Organization|Disne
00000070  79 7C                                                                                y|_
```

8.12. UserAccount

The UserAccount model represents a specific user account associated with an app or a service.

Property	Type	Description
Name	Field<string>	
Username	Field<string>	
Password	Field<string>	
ServiceType	Field<string>	The app or service from which the account was extracted.
ServerAddress	Field<string>	
Photos	MultiModelField<ContactPhoto>	UserAccount Photos.
Entries	MultiModelField<ContactEntry>	UserAccount entries collection.
Notes	MultiField<string>	
Addresses	MultiModelField<StreetAddress>	Addresses collection.
Organizations	MultiModelField<Organization>	Organizations collection.
TimeCreated	Field<TimeStamp>	

8.13. Party

The Party model is a container which represents one participant, party or attendee in several different models. The Party model has the properties listed in the following table.

Property	Type	Description
Role	Field<PartyRole>	Enum: General, To, From.
Identifier	Field<string>	Party identifier (e.g. number, email address, etc.).
Name	Field<string>	Party name.
Status	Field<PartyStatus>	The status of the message for that party.
DateDelivered	Field<TimeStamp>	The time the party received the message.
DateRead	Field<TimeStamp>	The time the party read the message.
DatePlayed	Field<TimeStamp>	The time the party played the message.
IPAddress	Field<string>	IP address associated with the party.

PartyRole is an enum containing the following values: General, To, From

PartyStatus is an enum containing the following values: Unknown, Sent, Unsent

There are several ways of initializing a **Party** object, as shown in the following example:

```
>>> ## Assume we have an identifier with value, or value and source
>>> (val,src)
('+12125555555', MemoryRange (12 bytes in 1 chunks))
>>> ## Init a Party object and add val/src to Identifier
>>> pa = Party()
>>> pa.Identifier.Value = val
>>> pa.Identifier.Source = src
>>> ## Update role
>>> pa.Role.Value = PartyRole.From
>>>
>>> ## Init with role using MakeFrom, MakeTo or MakeGeneral
>>> pa = Party.MakeFrom(val,src)
>>> pa
From: +12125555555
>>> ## Add party name
>>> pa.Name.Value = 'Donald Duck'
>>> pa
From: +12125555555 Donald Duck
```

8.14. SMS

The SMS model has the properties listed in the following table.

Property	Type	Description
TimeStamp	Field<TimeStamp>	
Status	Field<MessageStatus>	Displays the status value saved in the the MessageStatus Enum described below.
Parties	MultiModelField<Party>	SMS parties.
Folder	Field<string>	SMS Folder (e.g. Inbox, Draft, Sent).
SMSC	Field<string>	SMSC Number.
Body	Field<string>	
Source	Field<string>	SMS Source (default SMS option in the phone is left here empty).
AllTimeStamps	MultiModelField<TimeStampEntry>	A set of TimeStamps associated with the SMS, e.g., device time and network time.

An **SMS** is categorized in the **DataStore** according to the Folder property.

The MessageStatus enum contains the following values: Default, Sent, Unsent, Read, Unread.

8.15. Call

The Call model is used to describe a call-log entry and has the properties listed in the following table.

Property	Type	Description
Type	Field<CallType>	
Parties	MultiModelField<Party>	Call parties
TimeStamp	Field<TimeStamp>	
Duration	Field<TimeSpan>	Standard C# TimeSpan class
Source	Field<string>	Call Source
NetworkName	Field<string>	
NetworkCode	Field<string>	
VideoCall	bool	
CountryCode	Field<string>	

Calls are categorized in the DataStore according to the Source property.

The CallType enum contains the following values: Unknown, Incoming, Outgoing, Missed, SentMessages, ReceivedMessages, Hotlist, Conference and Rejected.

8.15.1. Use of C# TimeSpan class

Call duration information is usually stored in seconds. You can set the call duration.

Other methods for initializing the TImeSpan class can be found on Microsoft's MSDN.

```
>>> ## Import the TimeSpan class
>>> from System import TimeSpan
>>> ## Set the call duration with 100 seconds
>>> call = Call ()
>>> call.Duration.Value = TimeSpan (0,0,100)
>>> call
Call {Type: Field(Unknown), Duration: Field(00:01:40)}
```

8.16. Attachment

Before describing several models that may contain attachment files, let us review the properties of the Attachment model, which itself is a dependent model.

Property	Type	Description
FileName	Field<string>	
ContentType	Field<string>	
Charset	Field<string>	
Data	DataField	
MetaData	MultiField<string>	
URL	Field<string>	A URL string associated with the attachment.

The Data property has a special type called DataField, which receives only a MemoryRange as input in the Source property (the attachment data). It has no Value property.

```
>>> ## Generating an attachment
>>> att = Attachment ()
>>> att.Filename.Value = "MyAttachment"
>>> ## Assume we have a memory range which is the attachment itself called mem
>>> mem
MemoryNode 'Image' (50b in 1 chunks), 0 children
>>> att.Data.Source = mem
>>> att
Attachment {Filename: Field(MyAttachment), Data: Field(50 Bytes)}
```

8.17. MailMessage, Email, MMS

As emails and MMS messages obtain many common properties, they both inherit from the MailMessage class. The MailMessage properties are listed in the following table.

Property	Type	Description
Folder	Field<string>	
Status	Field<MessageStatus>	Status (the same values as SMS Status).
From	SingleModelField<Party>	
To	MultiModelField<Party>	
Cc	MultiModelField<Party>	
Bcc	MultiModelField<Party>	
Subject	Field<string>	
Body	Field<string>	
TimeStamp	Field<TimeStamp>	
Priority	Field<MailPriority>	Priority (Low, High, Normal).
Attachments	MultiModelField<Attachment>	
Source	Field<string>	

The MessageStatus enum values are described in the SMS model. The MailPriority values are: Normal, Low, High.

To, Cc, Bcc and Attachments may be set with several values using the **Add** function:

```
>>> email = Email ()
>>> email.To.Add ('mickey@disney.com')
>>> ## Return to our previously constructed attachment model
>>> att
Attachment {Filename: Field(MyAttachment), Data: Field(50 Bytes)}
>>> email.Attachments.Add (att)
```

The Email class contains an additional property.

Property	Type	Description
Account	Field<string>	Email account this mail was extracted from.

The Account and Folder properties are used for categorizing emails in the **DataStore** tree.

The MMS model is categorized by the Folder property.

8.18. Coordinate

The Coordinate model is used to describe a single coordinate for several location-based models (instant message, location, GPS fixes etc.). It has the properties listed in the following table.

Property	Type	Description
Longitude	Field<double>	
Latitude	Field<double>	
Elevation	Field<double>	
Map	Field<string>	Free text map to which the coordinate relates.
Comment	Field<string>	
PositionAddress	Field<string>	

Apart from the standard method of updating properties, it also has two useful constructors if the Source property of each field is not available:

```
>>> ## Generate a coordinate with either (lat, long) or (lat, long, elevation)
>>> Coordinate (28.419317, -81.581208)
(28.41932, -81.58121)
>>> Coordinate (28.419317, -81.581208, 30)
(28.41932, -81.58121)
>>> _.Elevation
Field(30)
```

8.19. InstantMessage

The InstantMessage model is used for entering data regarding a single chat message event (or file sending).

Property	Type	Description
From	SingleModelField<Party>	
To	MultiModelField<Party>	
Subject	Field<string>	
Body	Field<string>	
SourceApplication	Field<string>	
TimeStamp	Field<TimeStamp>	
DateRead	Field<TimeStamp>	
DateDelivered	Field<TimeStamp>	
Attachments	MultyModelField<Attachment>	List of attachments sent with the message.
Position	SingleModelField<Coordinate>	Single location sent with the message.
Status	Field<MessageStatus>	
SharedContacts	MultiModelField<Contact>	Collection of Contacts sent in the message.
Label	<string>	
PositionAddress	<string>	

8.20. Chat

The Chat model has the properties listed in the following table.

Property	Type	Description
Messages	MultiModelField<InstantMessage>	
Id	Field<string>	A unique ID for this chat instance.
StartTime	Field<TimeStamp>	
LastActivity	Field<TimeStamp>	
Participants	MultiModelField<Party>	
Source	Field<string>	

The Chat is categorized according to the **Source** property. This is usually set with the application which was used for the chat (e.g. Skype, WhatsApp, etc.).

```
>>> ## Generate a chat
>>> chat = Chat()
>>> chat.Id.Value = 'MyChat'
>>> chat.Source.Value = 'MyApp'
>>> ## Init an instant message
>>> im = InstantMessage()
>>> im.From.Value = Party.MakeFrom ('mickey',None)
>>> im.To.Add(Party.MakeTo ('donald', None))
>>> im.TimeStamp.Value = TimeStamp.FromUnixTime (1314867600)
>>> ## Add this im to the chat
>>> chat.Messages.Add(im)
```

8.21. Notification

The Notification model is used to represent incoming notifications from various apps, such as incoming push notifications.

Property	Type	Description
Participants	MultiModelField<Party>	A notification can include multiple parties, such as multiple tags in a photo.
To	SingleModelField<Party>	Recipient of the notification (single user).
Subject	Field<string>	
Body	Field<string>	
Source	Field<string>	Source application.
TimeStamp	Field<TimeStamp>	
DateRead	Field<TimeStamp>	
Attachments	MultyModelField<Attachment>	List of attachments sent with the message.
Position	SingleModelField<Coordinate>	Single location sent with the message.
Status	Field<MessageStatus>	
PositionAddress	Field<string>	Free text of the address.
Urls	MultiModelField<WebAddress>	All the internet links associated with the notification.
Type	Field<NotificationType>	
NotificationId	Field<string> NotificationId	Application specific ID of the notification.
PositionAddress	MultiField<string>	

8.22. Note

The Note model is used for storing user notes. Its properties are listed in the following table.

Property	Type	Description
Title	Field<string>	
Body	Field<string>	
Summary	Field<string>	
Creation	Field<TimeStamp>	
Modification	Field<TimeStamp>	
Source	Field<string>	
Position	SingleModelField<Coordinate>	
Address	SingleModelField<StreetAddress>	
Attachments	MultiModelField<Attachment>	List of notes attachments.
Folder	Field<string>	
PositionAddress	Field<string>	

8.23. CalendarEntry

The CalendarEntry model has the properties listed in the following table.

Property	Type	Description
Category	Field<string>	Event category (Task, Birthday etc.).
Subject	Field<string>	
Location	Field<string>	
Details	Field<string>	
Attendees	MultiModelField<Party>	
StartDate	Field<TimeStamp>	
EndDate	Field<TimeStamp>	
Reminder	Field<TimeStamp>	
Priority	Field<EventPriority>	Event priority enum
Status	Field<EventStatus>	Event status enum
Class	Field<EventClass>	Event class enum (Normal, Personal etc.).
RepeatRule	Field<RepeatRule>	Repeat rule enum (daily, weekly etc.).
RepeatUntil	Field<TimeStamp>	
RepeatDay	Field<RepeatDay>	Repeat day enum (days of the week).
RepeatInterval	Field<uint>	Interval number (e.g. every 2 weeks).
Source	Field<string>	

The Calendar enums need all to be specially imported as in the code below

EventPriority values: Unknown, Low, Normal, High.

EventStatus values: Unknown, Accepted, NeedsAction, Sent, Tentative, Confirmed, Declined, Completed, Delegated, InProgress, WaitingOnInfo.

EventClass values: Normal, Personal, Private, Confidential.

RepeatRule values: None, Daily, Weekly, Monthly, Yearly.

RepeatDay values: None, Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, First, Second, Third, Fourth, Last.

These values are set as flags and can be combined as in the following example:

```
>>> ## Import calendar enums and init a CalendarEntry
>>> from Data.Models.Calendar import RepeatDay, RepeatRule, EventType,
EventStatus,
EventPriority, EventClass
>>> cal = CalendarEntry ()
>>> ## Update repeat day with two values
>>> cal.RepeatDay.Value = RepeatDay.First | RepeatDay.Sunday
>>> cal
CalendarEntry {RepeatDay: Field(Sunday, First), Priority: Field(Unknown),
Status: Field(Unknown), Class: Field(Normal), RepeatRule: Field(None),
RepeatInterval: Field(0)}
```

8.24. Location

The Location model stores a single coordinate and any additional information related to it. Its properties are listed in the following table.

Property	Type	Description
Position	SingleModelField<Coordinate>	
Address	SingleModelField<StreetAddress>	
TimeStamp	Field<TimeStamp>	
Name	Field<string>	
Description	Field<string>	
Type	Field<string>	
Precision	Field<string>	
Map	Field<string>	
Category	Field<string>	
Confidence	Field<string>	
Origin	Field<LocationOrigin>	Represents the location's origin (device or external).
PositionAddress	Field<string>	

The Location is categorized by the **Category** property in the **DataStore** tree.

The LocationOrigin enum contains the following values: Unknown, Device, Exernal.

8.25. Journey

A Journey is constructed of several Locations with an initialized RoadPosition property that describes a locations-recorded journey (used for many GPS devices decoding). Its properties are listed in the following table.

Property	Type	Description
Name	Field<string>	
WayPoints	MultiModelField<Location>	Journey locations
Source	Field<string>	
StartTime	Field<TimeStamp>	
EndTime	Field<TimeStamp>	
FromPoint	SingleModelField<Location>	
ToPoint	SingleModelField<Location>	

8.26. Cookie

The Cookie model has the properties listed in the following table.

Property	Type	Description
Name	Field<string>	
Value	Field<string>	
Domain	Field<string>	
Path	Field<string>	
Expiry	Field<TimeStamp>	
CreationTime	Field<TimeStamp>	
LastAccessTime	Field<TimeStamp>	

8.27. VisitedPage

The VisitedPage model constructs the table entries for the Web History segment in the **DataStore** tree. Its properties are listed in the following table.

Property	Type	Description
Title	Field<string>	
Url	Field<string>	
LastVisited	Field<TimeStamp>	
VisitCount	Field<int>	
Source	Field<String>	

8.28. WebBookmark

The WebBookmark model is used for web bookmarks saved by the user. Its properties are listed in the following table.

Property	Type	Description
Title	Field<string>	
Url	Field<string>	
LastVisited	Field<TimeStamp>	
Source	Field<String>	
VisitCount	Field<int>	
TimeStamp	Field<TimeStamp>	
Path	Field<string>	
Position	SingleModelField<Coordinate>	
PositionAddress	Field<string>	

The model is categorized by the Path property.

8.29. BluetoothDevice

The BluetoothDevice model is used for Bluetooth devices that were connected to the device. Its properties are listed in the following table.

Property	Type	Description
Name	Field<string>	
MACAddress	Field<string>	
Info	Field<string>	Free text info regarding the device.
LastConnected	Field<TimeStamp>	The last date the devices were connected via this Bluetooth connection.

```
>>> ## Add a bt device to DataStore
>>> bt = BluetoothDevice ()
>>> bt.Name.Value = 'MyComp'
>>> bt.MACAddress.Value = '00:01:02:03:04:05'
>>> ds.Models.Add (bt)
>>> bt
BluetoothDevice {Name: Field(MyComp), MACAddress: Field(00:01:02:03:04:05)}
```

8.30. WirelessNetwork

The WirelessNetwork model is used for wireless network connections. Its properties are listed in the following table.

Property	Type	Description
BSSId	Field<string>	Network MAC address.
SSId	Field<string>	Network name.
SecurityMode	Field<string>	
LastConnection	Field<TimeStamp>	
LastAutoConnection	Field<TimeStamp>	

8.31. VoiceMail

The VoiceModel is stored for VM notifications or saved copies of the voicemail on the device. Its properties are listed in the following table.

Property	Type	Description
From	SingleModelField<Party>	Phone number and Caller name.
Name	Field<string>	File name.
TimeStamp	Field<string>	
Duration	Field<TimeStamp>	
Recording	NodeField	To store the message audio file if it is saved on the device.
Message	DataField	The actual voice mail message as a DataField.

8.32. Password

Property	Type	Description
AccessGroup	Field<string>	
Account	Field<string>	
Data	Field<string>	The password itself
GenericAttribute	Field<string>	
Label	Field<string>	
Server	Field<string>	
Service	Field<string>	
Type	Field<PasswordType>	

The Password model is used for presenting stored passwords. It has several properties to contain many possible password containers.

PasswordType is an enum with the following values: Default, Key, Secret and Token.

8.33. InstalledApplication

The InstalledApplication stores records on applications (mainly smartphone apps) the user installed. Its properties are listed in the following table.

Property	Type	Description
Name	Field<string>	
Version	Field<string>	
Description	Field<string>	
Identifier	Field<string>	
PurchaseDate	Field<TimeStamp>	
Copyright	Field<string>	
DeletedDate	Field<TimeStamp>	
AppGUID	Field<string>	The GUID identifier of the application.
Permissions	MultiField<PermissionCategory>	The app permissions.
DecodingStatus	Field<DecodingStatus>	
Users	MultiModelField<User>	

PermissionsCategory is an enum with the following values: Accounts, AppInfo, Audio, Bluetooth, Bookmarks, Calendar, Camera, Contacts, CostMoney, DeviceAlarms, Display, Locations, Messages, Microphone, Network, PersonalInfo, PhoneCalls, Photos, Reminders, SocialInfo, Storage, UserDictionary and Voicemail.

8.34. ApplicationUsage

The ApplicationUsage model is used for keeping records on various applications usage (mainly found in smartphones). Its properties are listed in the following table.

Property	Type	Description
Name	Field<string>	
LaunchCount	Field<int>	Number of times it was launched.
ActivationCount	Field<int>	Number of times the application was switched from background to active mode.
ActiveTime	Field< TimeSpan>	Time the application was active.
BackgroundTime	Field<TimeSpan>	Time the application was in background mode.
Date	Field<TimeStamp>	Last updating date.
LastLaunch	Filed<TimeStamp>	Last launch time.
LastUsageDuration	Field<TimeSpan>	Duration of last usage.

8.35. DictionaryWord

The DictionaryWord saves information on user automatically generated dictionary (to prevent from autocorrect algorithms to change specific words). Its properties are listed in the following table.

Property	Type	Description
Word	Field<string>	
Locale	Field<string>	Dictionary locale (e.g. en-us).
Frequency	Field<string>	Number of appearances of the word.
Source	Field<string>	

8.36. SharedFile

The SharedFile model saves information about a file shared between several people (like *Share File* in BlackBerry). Its properties are listed in the following table.

Property	Type	Description
Caption	Field<string>	The file caption.
Source	Field<string>	
Type	Field<FileType>	
TimeStamp	Field<TimeStamp>	
Content	NodeField	
Owner	SingleModelField<Party>	
Responders	MultiModelField<Party>	
Comments	MultiModelField<InstantMessage>	List of comments about the file.

8.37. Map

The map model saves a single image of a map and the metadata that was used when the map was viewed. Its properties are listed in the following table.

Property	Type	Description
Source	Field<string>	
ZoomLevel	Field<int>	
Tiles	MultiField<string>	List of coordinates.
MapData	NodeField	The image of the map.

8.38. SearchedItem

The SearchedItem model stores a single searched item. Its properties are listed in the following table.

Property	Type	Description
TimeStamp	Field<TimeStamp>	
Value	Field<string>	
Source	Field<string>	
SearchResults	Field<string>	
Position	SingleModelField<Coordinate>	
PositionAddress	Field<string>	

8.39. CarvedString

The CravedString model contains a string found in the dump, usually using carving ability

Property	Type	Description
Value	Field<string>	
Source	Field<string>	
Metadata	Field<string>	Additional data on the string.

8.40. PoweringEvent

The PoweringEvent stores information about the power status of device components

Property	Type	Description
Element	Field<PowerElementType>	The component
TimeStamp	Field<TimeStamp>	
Event	Field<Event>	The event itself

The PowerElementType enum with the following value: Device.

The Event enum with the following values: On, Off and Reset.

8.41. MobileCard

The MobileCard model stores information about the various cards on the mobile device.

Property	Type	Description
Name	Field<string>	
Description	Field<string>	
Organization	Field<Organization>	
PurchaseTime	Field<TimeStamp>	
ModifyTime	Field<TimeStamp>	
ActivationTime	Field<TimeStamp>	
ExpirationTime	Field<TimeStamp>	
Position	Field<Coordinate>	
PositionAddress	Field<string>	
Type	Field<MobileCardType>	
Details	MultiField<string>	
Barcode	Field<string>	
Source	Field<string>	

MobileCardType enum with the following values: Unknown, Coupon, StoreCard, EventTicket, BoardingPass, Generic

9. Other useful functions

9.1. SMS PDU Parser

Use the following sequence to automatically parse a PDU formatted SMS into SMS model:

```
>>> ## Assume we have a memory range which is a PDU formatted SMS
>>> mem
MemoryNode 'Image' (38b in 1 chunks), 0 children
>>> ## Import relevant lib
>>> from PhoneUtils.GSM import PDUParser
>>> ## Try SMS parsing
>>> ## The function requires SMSC existence indication
>>> hasMMC = True
>>> PDUParser.TryParsePDU(mem, hasMMC).SMS
SMS {TimeStamp: Field(8/26/2002 7:37:41 PM(UTC+0)), Status: Field(Default),
Parties: Field(From: +31641600986), SMSC: Field(+31624000000), Body: Field(How
are you?)}
```

9.2. Independent scripts

To use all relevant classes and methods in an independent script, add one of the following lines at the start of the script.

For Decoding Scope, add the following line:

```
from physical import *
```

This guarantees that the **ds** variable is initialized when the script runs in Inseyets.PA. To run a script, choose **Python** > **Run script > Decoding Scope** and then the script file.

For Applicative Scope, add the following line:

```
from PAphysical import *
```

This guarantees that the **ds** variable is initialized when the script runs in Inseyets.PA. To run a script, choose **Python** > **Run script > Applicative Scope** and then the script file.