Overview

Python is a powerful, object-oriented open-source scripting language that is in use all over the world. In **Iguana** and **Chameleon**, you can write Python scripts that allow you to manipulate HL7 message data. The following pages provide a brief summary of the features of Python.

Basic Concepts

Data Types

Numbers can be integers or floating point values:

```
      42    3.14159
```

Strings can be enclosed in single or double quotes, and can contain any printable character:

```
      "test"      'Hello, world!'
```

The following **escape sequences** can be used in strings:

Escape Sequence	Meaning
\	Backslash
\'	Single quote (useful in strings enclosed in single quotes)
\"	Double quote (useful in strings enclosed in double quotes)
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab

To create a raw string, in which backslashes are not interpreted as escape sequences, specify **r** before the opening single quote or double quote that encloses the string:

```
      rawstr = r"This is a \raw \string \that \contains four backslashes"
```

Variables

A **variable** can be any combination of letters, digits and underscore characters. The first character cannot be a digit. Variables in Python are **case sensitive:** `variable` and `VARIABLE` are not the same.

```
      x     _LabName    RESULT2     VaRiAbLe
```

Assignment

Use an **assignment** to store a value in a variable:

```
      patientid = 42113
      patientstatus = "Admitted"
```

The None Object

Python defines a special object, called **None**, that can be used to specify the empty value:

```
value = None
```

By default, the Python **None** object **is disabled in VMD files**. See **Disabling/Enabling the Python None Object** in the manual for more details.

String and Number Conversion

Use **int**, **float** and **str** to convert numbers to strings and vice versa:

```
integertemp = int("37")
floattemp = float("98.6")
stringtemp = str(98.6)
```

Displaying Values

print displays values on the screen or in a log file:

```
print 'The patient ID is', patientid
```

You can use **%s** with **print** to display the values of variables as part of a string:

```
print 'The patient IDs are %s and %s' % (patientid1, patientid2)
```

Comments

Everything after the **#** character is treated as a comment and ignored:

```
# this is a comment
temperature = 98.6   # this is also a comment
```

Multi-Line Statements

Use \ to continue a statement on more than one line:

```
floattemp =\
      float("98.6")
```

Arithmetic

Python supports the standard arithmetic operations on integers and floating point numbers:

```
y = x + 1    # addition              y = x - 1    # subtraction
y = x * 1.8  # multiplication        y = x / 1.8  # division
y = 33 % 4   # remainder from division, or modulo; y is 1 in this example
y = 2 ** 5   # exponentiation, or x to the power y; 32 in this example
```

Operations are normally performed in this order: ******, then *****, **/** and **%**, then **+** and **-**.
Use parentheses to specify an order of operation.

You can use the **+** and ***** operators with strings:

```
patientid = "4" + "2" + 2 * "1" + "3"  # patientid is assigned '42113'
```

Conditional Statements and Loops

Conditional Statements: if, elif and else

Use **if**, **elif** and **else** to define code to be executed if a specified condition is true:

```
if patientid == 42113:
      print "The patient ID is 42113"
elif patientid == 42007:
      print "The patient ID is 42007"
else:
      print "The patient ID is some other number"
```

Python uses indenting to determine which statements are contained inside a conditional statement. Avoid mixing spaces and tabs when indenting.

The condition in a conditional statement must be terminated with a **:** (colon) character.

Loops: while and for

Use **while** to define code to be executed while a specified condition is true:

```
# display the numbers from 1 to 10
x = 1
while x <= 10:
      print x
      x = x + 1
```

Use **for** to loop through a range of numbers or a list:

```
# display the numbers from 1 to 10
for x in range(1, 11):
      print x
```

Controlling Loops: break and continue

Use **break** to exit from the middle of a loop, or **continue** to start another iteration of a loop:

```
# print 1 to 5
x = 1
while x <= 10:
      print x
      if x == 5:
            break
      x = x + 1
```

```
# print 1 to 10, skipping 5
x = 1
while x <= 10:
      if x == 5:
            x = x + 1
            continue
      print x
      x = x + 1
```

Comparison Operators

Symbol	Meaning	Symbol	Meaning
==	Equal to	**<=**	Less than or equal to
!= or **<>**	Not equal to	**>**	Greater than
<	Less than	**>=**	Greater than or equal to

Boolean Operators

Use **and** and **or** to specify multiple conditions for a conditional statement, or **not** to negate a condition:

```
if not (patientid == 42113 and hospitalid == 2000) or labid == 5555:
      print "True!"
```

Using None in Comparisons

If your VMD file has **None** defined, you can use it in conditional expressions:

```
if value == None:
      # Value is empty.
```

Lists

A **list** is an ordered collection of values. Lists are enclosed in brackets ([]):

```
patientids = [42446, 42113, 42007]
segmentlist = ["MSH", "EVN", "PID", "NK1", "PV1"]
```

Lists can contain numbers, strings, or other lists.

Assignment From Lists

You can assign a list to a variable or to multiple variables at once:

```
patientinfo = ['JohnDoe', 42446, 'Admitted', 1000]
(patientname, patientid, patientstatus) = ['JohnDoe', 42446, 'Admitted']
```

You can also assign a single element of a list to a variable:

```
patientid = patientinfo[1] # assigns the second element of patientinfo to patientid
```

List Editing Functions

Function	Description	Example
append	Add a value to the end of a list	x = [1, 2, 3] x.append(4) # x is now [1, 2, 3, 4]
del	Delete a value from a list	x = [1, 2, 3, 4] del x[1] # x is now [1, 3, 4]

index	Return the index of an item in a list	x = [1, 2, 3, 5, 2, 4]
		y = x.index(3) # y is now 2

len	Return the number of values in a list	x = [1, 2, 3, 4]
		y = len(x) # y is now 4

pop	Remove an item from a list and return it	x = [1, 2, 3, 4]
		y = x.pop(1) # x is now [1, 3, 4]; y is now 2
		z = x.pop() # x is now [1, 3]; z is now 4

remove	Remove a specified element from a list	x = [1, 2, 3, 4] x.remove(2) # x is now [1, 3, 4]
reverse	Reverse the order of a list	x = [1, 2, 3, 4] x.reverse() # x is now [4, 3, 2, 1]

sort	Sort a list in numeric or alphabetic order	x = [3, 1, 4, 2]
		x.sort() # x is now [1, 2, 3, 4]
		y = ['c', 'd', 'b', 'a']
		y.sort() # y is now ['a', 'b', 'c', 'd']

You can also use **+** to join two lists:

```
x = [1, 2] + [3, 4]   # x now contains [1, 2, 3, 4]
```

Lists and Conditional Statements

You can use lists with the **for** and **if** statements:

```
primes = [2, 3, 5, 7, 11,     segments = ["MSH", "EVN", "PID",
      13, 17, 19, 23, 29]           "NK1", "PV1"]
for x in primes:              if x in segments:
      print x                       print x, "is a segment in the list"
```

Dictionaries

A **dictionary** is a collection of key-value pairs. In a dictionary definition, a key and its value are separated by a colon:

```
pidlist = {"Smith,Mary":"P12345", "Doe,John":"P12346", "Jones,Charlie":"P12347"}
```

Accessing Dictionaries

To access a value, supply its key:

```
patientid = pidlist["Doe,John"]
```

To add a new element to a dictionary, assign a value to a new key:

```
pidlist["Baxter,Ted"] = "P12350"
```

To update an element of a dictionary, assign a new value to its key:

```
# update the patient ID for Charlie Jones
pidlist["Jones,Charlie"] = "P55555"
```

To delete an element from a dictionary, use **del**:

```
del(pidlist["Doe,John"])
```

Use **has_key** to check whether a key is defined in a dictionary:

```
if not pidlist.has_key["Roe,Jane"]:
      print "Jane Roe's patient ID is not known"
```

Dictionaries and Loops

To use a dictionary in a loop, use the **keys** function. This processes each element of the dictionary in turn:

```
for name in pidlist.keys():
      patientid = pidlist[name]
      print name, "has Patient ID", patientID
```

Note that **keys** does not process elements in any particular order. To process keys in alphabetical order, use **sort**:

```
sortedkeys = pidlist.keys()
sortedkeys.sort()
for name in sortedkeys:
      patientid = pidlist[name]
      print name, "has Patient ID", patientid
```

Mapping

You can use dictionaries to map one set of values to another:

```
mapping = {
      'PatientID_internal':'PatientID_external',
      'DoctorID_internal':'DoctorID_external',
      'FacilityID_internal':'FacilityID_external'
}
```

This is more convenient than using a chain of **if** and **elif** statements.

Functions

Creating a Function

To create a function, use the **def** statement:

```
      def print_HL7_field_delimiter():
            print "|"
```

The statements contained in the function definition must be indented.

To call a function, specify its name followed by parentheses:

```
      print_HL7_field_delimiter()
```

You must define a function before you can use it.

Function Parameters

You can use parameters to pass values to a function:

```
      def print_delimiter(text):
            print text

      print_delimiter("|")
```

You can specify a default value for a parameter, to be used if the function call does not provide one:

```
      def print_multiple_delimiters(text, count=1):
            print text * count

      print_multiple_delimiters("|")          # prints |
      print_multiple_delimiters("|", 3)       # prints |||
```

Return Values

Use **return** to specify a return value from a function:

```
      def FtoC(degf):
            degc = (degf - 32) / 1.8
            return degc

      tempf = 98.6
      tempc = FtoC(tempf)
```

A function can return more than one value:

```
      def FtoC_andK(degf):
            degc = (degf - 32) / 1.8
            degk = degc + 273.15
            return degc, degk
```

Local and Global Variables

Variables created (assigned to) inside functions are **local** variables (unless the **global** statement is used to indicate a global variable). A local variable cannot be accessed outside the function in which it was created:

```
      def FtoC(degf):
            degc = (degf - 32) / 1.8
            return degc       # degc is a local variable
```

Variables created outside functions are **global variables**, and can be accessed anywhere.

Working With Strings

String Indexing and Slices

You can use an index or a slice to copy part of a string to a variable:

Copy Operation	Syntax	Example	In Example, substring of "XYZ Hospital and Treatment Center" which is assigned to x
Copy a single character	[*num*]	`x = loc[1]`	**x** is assigned the second character of the string, which is **"Y"**
Copy a single character, indexing from end of string	[-*num*]	`x = loc[-2]`	**x** is assigned the second-last character of the string, which is **"e"**
Copy a slice	[*num1*:*num2*]	`x = loc[1:3]`	**x** is assigned the second and third characters, which are **"YZ"**
Copy a slice, starting from the beginning of the string	[:*num*]	`x = loc[:3]`	**x** is assigned the first three characters, which are **"XYZ"**
Copy all but the first *num* characters of a string	[*num*:]	`x = loc[17:]`	**x** is assigned the last characters of the string, which are **"Treatment Center"**
Copy a slice, starting from the end of the string	[-*num*:]	`x = loc[-3:]`	**x** is assigned the last three characters, which are **"ter"**
Copy all but the last *num* characters of a string	[:-*num*]	`x = loc[:-2]`	**x** is assigned **"XYZ Hospital and Treatment Cent"**
Copy a slice, indexing from the end of the string	[-*num1*:-num2]	`x = loc[-4:-2]`	**x** is assigned the third-last and fourth-last characters, which are **"nt"**

You can also use slices with lists:

```
        segmentlist = ["MSH", "EVN", "PID", "NK1", "PV1"]
        x = segmentlist[1:3]     # x is now ['EVN', 'PID']
```

String Capitalization Functions

Function	Description	Example

capitalize	Convert the first character to upper case, and the rest to lower case	`x = "abc"`
		`x = x.capitalize() # x is now "Abc"`

lower	Convert all characters to lower case	`x = "ABC"`
		`x = x.lower() # x is now "abc"`

swapcase	Convert upper case characters to lower case, and lower to upper	`x = "Abc"`
		`x = x.swapcase() # x is now "aBC"`

title	Convert the first character of every word to upper case, and the rest to lower case	`x = "ABC DEF"`
		`x = x.title() # x is now "Abc Def"`

upper	Convert all characters to upper case	`x = "abc"`
		`x = x.upper() # x is now "ABC"`

Editing Functions

Function	Description	Example

strip([*chars*]**)**	Remove all leading and trailing occurrences of the characters in *chars* – remove spaces and tabs if *chars* is not specified	`location = "*=*Treatment*=Center=*="`
		`newloc = location.strip("*=")`
		`# newloc is "Treatment*=Center"`

lstrip([*chars*]**)**	Same as **strip**, except that it affects leading characters only	`location = "*=*Treatment*=Center=*="`
		`newloc = location.lstrip("*=")`
		`# newloc is "Treatment*=Center=*="`

rstrip([*chars*]**)**	Same as **strip**, except that it affects trailing characters only	`location = "*=*Treatment*=Center=*="`
		`newloc = location.rstrip("*=")`
		`# newloc is "*=*Treatment*=Center"`

replace(*src*, *dst* [,*max*]**)**	Replace all occurrences of the substring *src* with *dst* – *max*, if specified, is the maximum number of replacements	`location = "Treatment Center"`
		`newloc = location.replace("e","E",2)`
		`# newloc is "TrEatmEnt Center"`

zfill(*len***)**	Pad a string with leading zeroes to make it length *len*	`patientid = "42113"`
		`patientid = patientid.zfill(10)`
		`# patientid is now "0000042113"`

Chameleon also defines built-in functions that handle character stripping:

Function	Description	Example
strip_chars(*char*, *string***)**	Strip all occurrences of *char* from *string*	`value = strip_chars('_', value)`
strip_leading_chars(*char*, *string***)**	Strip all leading *char* characters from *string*	`value = strip_leading_chars('0', value)`
strip_trailing_chars(*char*, *string***)**	Strip all trailing *char* characters from *string*	`value = strip_trailing_char('0', value)`
strip_non_numeric_chars (*string***)**	Remove all non-numeric characters from *string*	`value = strip_non_numeric_chars(value)`

Splitting and Searching Functions

Function	Description	Example

split(*delim* [,*max*]**)**	Split a string into a list, breaking at every occurrence of *delim* – *max* is optional and represents a maximum number of breaks	`location = "XYZ,Treatment,Center"`
		`wordlist = location.split(",")`
		`# wordlist contains ['XYZ',`
		`# 'Treatment', 'Center']`

find(*str* [,*start* [,*end*]]**)**	Return the index of a character or substring *str* in a string (or -1 if not found) – *start* and *end* are optional, and represent the start and end indexes of the search	`location = "XYZ Treatment Center"`
		`x = location.find("Treat")`
		`# x is now 4`

delim.**join(***list***)**	Create a string from *list*, using *delim* to separate each pair of elements in the list	`wordlist = ['XYZ','Treatment','Center']`
		`location = ",".join(wordlist)`

String Comparison Functions

Function	Description
isalnum	Return **True** if all characters in the string are alphanumeric
isalpha	Return **True** if the string consists of letters
isdigit	Return **True** if the string consists of digits
islower	Return **True** if the string consists of non-capitalized letters
isspace	Return **True** if the string consists of whitespace (spaces or tabs)
istitle	Return **True** if the string is in title format (for example, "This Is A Title")
isupper	Return **True** if the string consists of capitalized letters
startswith(*prefix***)**	Return **True** if the string starts with the substring *prefix*
endswith(*suffix***)**	Return **True** if the string ends with the substring *suffix*

Because string comparison functions return **True** or **False**, they are ideal for use in conditional statements:

```
location = "XYZ Hospital and Treatment Center"
if (location.startswith("XYZ")):
      print "The location starts with 'XYZ'"
```

Pattern Matching in Strings

In Python, the **re** module allows you to use regular expressions to search in a string for a substring matching a specified pattern. Functions provided in this module include:

Function	Description
re.search(*pattern*,*str*[,*flag*]**)**	Search for a substring matching pattern in string *str*; *flag* is optional, and controls the behavior of the search. Returns the search object used by **search.start** and **search.group**
search.start	If a pattern match is found by **re.search**, return the index of the start of the matched substring
search.group	If a pattern match is found by **re.search**, return the matched substring
re.sub(*pattern*, *repl*, *str*[, *count*]**)**	Find occurrences of pattern in *str* and replace them with *repl*. *count*, if provided, specifies the maximum number of replacements

Special Characters in Pattern Matching

The pattern parameter for **re.search** can contain any or all of the following special characters:

Character	Meaning
*****	Zero or more occurrences of the preceding character
+	One or more occurrences of the preceding character
?	Zero or one occurrences of the preceding character
.	Any character
[_chars_**]**	Any character inside the brackets
[_char1_**-**_char2_**]**	Any character in the range between _char1_ and _char2_
[^_chars_**]**	Any character not inside the brackets
{_num_**}**	Exactly _num_ occurrences of the preceding character
{_num1_**,**_num2_**}**	Between _num1_ and _num2_ occurrences of the preceding character
|	Matches either of two alternatives (for example, **abc|def**)
^	Matches the start of the string only
$	Matches the end of the string only
^(?!_str_**)**	Matches anything other than _str_
^(?!_str1_**|**_str2_**)**	Matches anything other than _str1_ and _str2_
\	If followed by any of the above characters, indicates that the following character is not to be treated as a special character
\s	Matches any whitespace character (including space, tab, newline and carriage return)
\d	Matches any digit (0 through 9)
\w	Matches any digit, any alphabetic character, or underscore

Here is an example that uses a regular expression to perform a search:

```
      import re

      pattern = '(iss)+'
      search = re.search(pattern,'Mississippi')
      if search:
            match = search.group()
            index = search.start()
            print "Matched", match, "at index", index
```

To specify that case is to be ignored when searching, specify the **IGNORECASE** flag as a parameter for **search**:

```
      import re

      substring = 'xyz'
      # the following search is successful
      search = re.search(substring,'XYZ HOSPITAL',re.IGNORECASE)
```

Error Detection

Python allows you to define **exception handlers** that catch and handle runtime errors generated by your program. To define an exception handler, use the **try** and **except** statements:

```
try:
      cost = totalcost / days
except ZeroDivisionError:
      print "Division by zero error"
```

The error name in the except statement always matches the error name that appears in a runtime error message. You can provide multiple except statements in an exception handler.

For a complete list of the runtime errors defined in Python, see the **Built-in Exceptions** section of the online Python documentation.

Modules

A **module** is a file containing a collection of functions and variables. This collection can be referenced by other Python programs. For example, here is a module that handles temperature conversion:

```
def FtoC(degf):
      degc = (degf - 32) / 1.8
      return degc

def CtoF(degc):
      degf = degc * 1.8 + 32
      return degf
```

All files that define modules must have a suffix of **.py**. The name of a file always matches the name of its module: if a file is named **temperature.py**, the module it contains is named **temperature**.

Importing Modules

To use a module, import it into your code using the **import** statement:

```
import temperature

degf = temperature.CtoF(37.0)
```

When you use **import**, you must specify the module name to access the module's functions and variables. If you do not want to specify the module name when calling a function, use **from** to import the function:

```
from temperature import CtoF

degf = CtoF(37.0)
```

You can use **from** to import every function and variable in a module:

```
from temperature import *
```

Using Built-In Modules

Python provides built-in modules that perform a variety of common tasks. To use a built-in module, ensure that the module is in the Python engine's search path, and use **import** or **from** to import the module into your code.

For a complete list of the Python modules supported in Chameleon, see the Supported Python Libraries section of the manual: *http://www.interfaceware.com/manual/python_libraries.html*.

Using Python Scripts in Chameleon

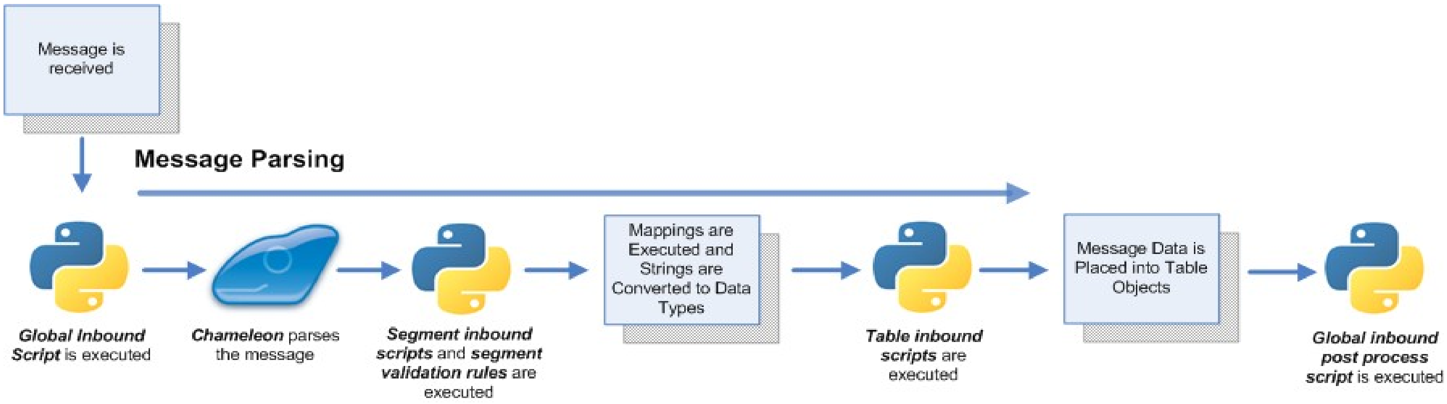
In **Chameleon**, you can use Python scripts to massage data when parsing incoming HL7 messages, generating outgoing HL7 messages, or transforming one HL7 format into another.

Using Python When Parsing Messages

When parsing HL7 messages in **Chameleon**, you can create:

- A **Global Inbound Script**, to be executed before parsing begins;
- **Segment Inbound Scripts**, which manipulate segment field data;
- **Table Inbound Scripts**, which manipulate table column data;
- A **Global Inbound Post Process Script**, to be executed after the message data is placed into table objects.

This diagram shows the order in which these scripts are executed:


Message is received
Message Parsing
Global Inbound Script is executed
Chameleon parses the message
Segment inbound scripts and *segment validation rules* are executed
Mappings are Executed and Strings are Converted to Data Types
Table inbound scripts are executed
Message Data is Placed into Table Objects
Global inbound post process script is executed

The following global variables are defined in message parsing scripts:

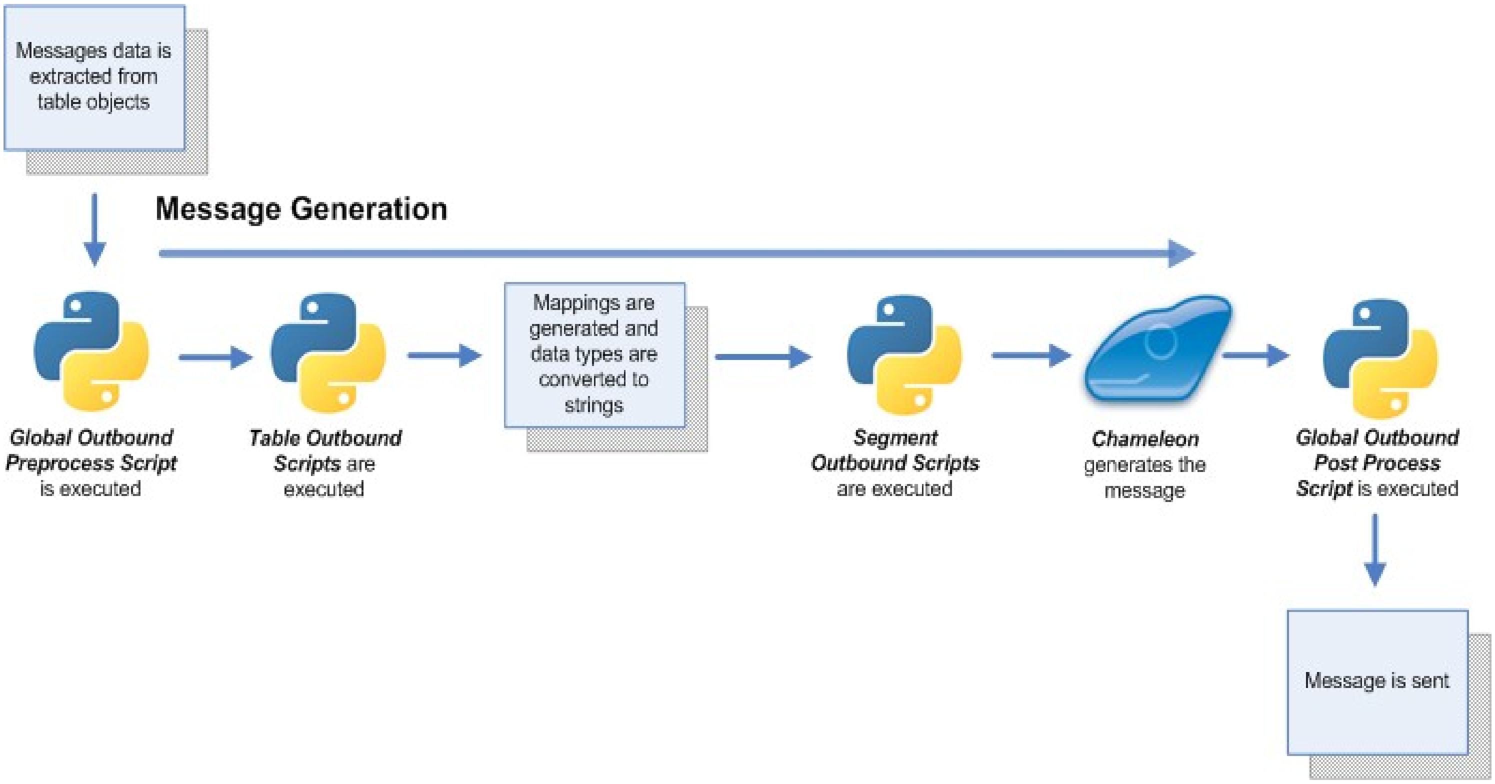
Script	Variable	Contents
All scripts	**environment**	Enables database connection and date/time formatting
Global Inbound and Global Inbound Post Process only	**value**	The entire message string
Segment Inbound only	**value**	The first subfield of the current field
	field	All subfields of the current field
Table Inbound only	**value**	The table column data
	table	Contains a method that removes the current row of the table

Using Python When Generating Messages

When generating HL7 messages in **Chameleon**, you can create:

- A **Global Outbound Preprocess Script**, which is executed before message generation to define variables and functions;
- **Table Outbound Scripts**, which manipulate table column data;
- **Segment Outbound Scripts**, which manipulate segment field data;
- A **Global Outbound Post Process Script**, which is executed after the message string has been generated.

This diagram shows the order in which these scripts are executed:


Messages data is extracted from table objects
Message Generation
Global Outbound Preprocess Script is executed
Table Outbound Scripts are executed
Mappings are generated and data types are converted to strings
Segment Outbound Scripts are executed
Chameleon generates the message
Global Outbound Post Process Script is executed
Message is sent

The following global variables are defined in message generation scripts:

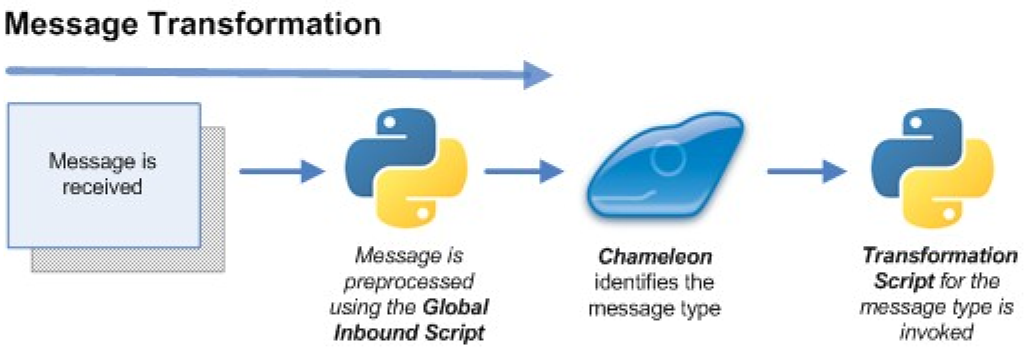
Script	Variable	Contents
All scripts	**environment**	Enables database connection and date/time formatting
	guid	Contains a method that creates a unique global ID for the HL7 message
Table Outbound only	**value**	The table column data
Segment Outbound only	**value**	The first subfield of the current field
	field	All subfields of the current field
Global Outbound Post Process only	**value**	The entire message string

Using Python When Transforming Messages

When using **Chameleon** to transform HL7 messages, you can create:

- A **Global Inbound Script**, which preprocesses the message before transformation begins;
- A **Transformation Script**, which performs the actual transformation.

This diagram shows the order in which these scripts are executed:


Message Transformation
Message is received
Message is preprocessed using the Global Inbound Script
Chameleon identifies the message type
Transformation Script for the message type is invoked

The following global variables are defined in message transformation scripts:

Variable	Contents
environment	Enables iteration over all message segments, database connection, and date/time formatting
value	The HL7 message string

Delimiter Functions

The following functions specify or set HL7 delimiters in **Chameleon**. In these functions, *environment* is the predefined **Chameleon** environment variable.

Function	Description
separator_char(*environment*,*num***)**	Returns the delimiter specified in the Options Window. *num* corresponds to: 0 - Segment delimiter 1 - Composite delimiter 2 - Sub-composite delimiter 3 - Sub-sub-composite delimiter
set_separator_char(*environment*, *num*, *newValue***)**	Sets the delimiter specified by *num* to *newValue*. The values of *num* are the same as in **separator_char**
escape_char(*environment***)**	Returns the escape delimiter specified in the Options window.
set_escape_char(*environment*, *newValue***)**	Sets the escape delimiter to *newValue*
repeat_char(*environment***)**	Returns the repeat delimiter specified in the Options window.
set_repeat_char(*environment*, *newValue***)**	Sets the repeat delimiter to *newValue*

Additional Resources

We hope that you have found this Quick Reference Guide useful. For more information on Python, refer to the following resources:

- The **Using Python Scripting** section of the **iNTERFACEWARE** products manual: *http://www.interfaceware.com/manual/python.html*
- The documentation provided for the version of Python supported by **Chameleon**: *http://www.python.org/doc/2.2.3/*
- The Python Tutorial page: *http://www.python.org/doc/2.2.3/tut/tut.html*

www.interfaceware.com
sales@interfaceware.com
☎ 1-888-824-6785