HelixCore

P4Python API Scripting Guide

2018.2

December 2018

PERFORCE

www.perforce.com

Contents

How to use this guide

This guide contains details about using the derived API for Python to create scripts that interact with Helix Core Server. You can download the API from the Perforce web site. The derived API depends on the Helix C/C++ API. For details, see the *C/C++ API User Guide*.

This section provides information on typographical conventions, feedback options, and additional documentation.

Syntax conventions

Helix documentation uses the following syntax conventions to describe command line syntax.

Notation	Meaning
`literal`	Must be used in the command exactly as shown.
italics	A parameter for which you must supply specific information. For example, for a *serverid* parameter, supply the ID of the server.
[`-f`]	The enclosed elements are optional. Omit the brackets when you compose the command.
...	■ Repeats as much as needed: • `alias-name[[$(arg1)... [$(argn)]]=transformation` ■ Recursive for all directory levels: • `clone perforce:1666 //depot/main/p4... ~/local-repos/main` • `p4 repos -e //gra.../rep...`
element1 | *element2*	Either *element1* or *element2* is required.

Feedback

How can we improve this manual? Email us at manual@perforce.com.

Other documentation

See https://www.perforce.com/support/self-service-resources/documentation.

P4Python

Introduction

P4Python, the Python interface to the Helix C/C++ API, enables you to write Python code that interacts with a Helix Core Server. P4Python enables your Python scripts to:

- Get Helix Server data and forms in dictionaries and lists.
- Edit Helix Server forms by modifying dictionaries.
- Provide exception-based error handling and optionally ignore warnings.
- Issue multiple commands on a single connection (performs better than spawning single commands and parsing the results).

System Requirements and Release Notes

P4Python is supported on Windows, Linux, Solaris, OS X, and FreeBSD.

For system requirements, see the release notes at https://www.perforce.com/perforce/doc.current/user/p4pythonnotes.txt.

> **Note**
> When passing arguments, make sure to omit the space between the argument and its value, such as in the value pair **-u** and ***username*** in the following example:
>
> **anges = p4.run_changes("-u*username*", "-m1").shift**
>
> If you include a space (**"-u *username*"**), the command fails.

Installing P4Python

> **Important**
> Before installing P4Python, any previously installed versions should be uninstalled.

As of P4Python 2015.1, the recommended mechanism for installing P4Python is via **pip**. For example:

```
$ pip install p4python
```

pip installs binary versions of P4Python where possible, otherwise it attempts to automatically build P4Python from source.

Windows users can download an installer containing pre-built packages for P4Python from the Perforce web site at https://www.perforce.com/downloads/helix-core-api-python.

> **Note**
> When P4Python is built without the `--apidir` option, setup attempts to connect to ftp.perforce.com to download the correct version of the P4API binary. If the P4API download is successful, it is unpacked into a temporary directory.
>
> When P4Python is built and the `--ssl` is provided without a path, setup attempts to determine the correct path of the installed OpenSSL libraries by executing `openssl version`.

Programming with P4Python

P4Python provides an object-oriented interface to Helix Core Server that is intended to be intuitive for Python programmers. Data is loaded and returned in Python arrays and dictionaries. Each `P4` object represents a connection to the Helix Server.

When instantiated, the `P4` instance is set up with the default environment settings just as the command line client `p4`, that is, using environment variables, the registry or user preferences (on Windows and OS X) and, if defined, the `P4CONFIG` file. The settings can be checked and changed before the connection to the server is established with the `P4.connect()` method. After your script connects, it can send multiple commands to the Helix Server with the same `P4` instance. After the script is finished, it should disconnect from the server by calling the `P4.disconnect()` method.

The following example illustrates the basic structure of a P4Python script. The example establishes a connection, issues a command, and tests for errors resulting from the command:

```
from P4 import P4,P4Exception     # Import the module
p4 = P4()                         # Create the P4 instance
p4.port = "1666"
p4.user = "fred"
p4.client = "fred-ws"             # Set some environment variables

try:                              # Catch exceptions with try/except
  p4.connect()                    # Connect to the Perforce server
  info = p4.run( "info" )         # Run "p4 info" (returns a dict)
  for key in info[0]:             # and display all key-value pairs
    print key, "=", info[0][key]
  p4.run( "edit", "file.txt" )    # Run "p4 edit file.txt"
  p4.disconnect()                 # Disconnect from the server
except P4Exception:
```

```
  for e in p4.errors:            # Display errors
      print e
```

This example creates a client workspace from a template and syncs it:

```
from P4 import P4, P4Exception

template = "my-client-template"
client_root = "C:\work\my-root"
p4 = P4()

try:
  p4.connect()
  # Retrieve client spec as a Python dictionary
  client = p4.fetch_client( "-t", template )
  client._root = client_root
  p4.save_client( client )
  p4.run_sync()

except P4Exception:
  # If any errors occur, we'll jump in here. Just log them
  # and raise the exception up to the higher level
```

Note
When extending the P4 class, be sure to match the method signatures used in the default class. P4Python uses both variable length arguments (`*args`) and keyword arguments (`**kwargs`). Review the P4.py in the source bundle for specifics. Example code:

```
class MyP4(P4.P4):
    def run(self, *args, **kargs):
        P4.P4.run(self, *args, **kargs)
```

Submitting a Changelist

This example creates a changelist, modifies it and then submits it:

```
from P4 import P4

p4 = P4()
```

```
p4.connect()
change = p4.fetch_change()

# Files were opened elsewhere and we want to
# submit a subset that we already know about.

myfiles = ['//depot/some/path/file1.c', '//depot/some/path/file1.h']
change._description = "My changelist\nSubmitted from P4Python\n"
change._files = myfiles   # This attribute takes a Python list
p4.run_submit( change )
```

Logging into Helix Server ticket-based authentication

On some servers, users might need to log in to Helix Server before issuing commands. The following example illustrates login using Helix Server tickets:

```
from P4 import P4

p4 = P4()
p4.user = "bruno"
p4.password = "my_password"
p4.connect()
p4.run_login()
opened = p4.run_opened()

...
```

Connecting to Helix Server over SSL

Scripts written with P4Python use any existing `P4TRUST` file present in their operating environment (by default, `.p4trust` in the home directory of the user that runs the script).

If the fingerprint returned by the server fails to match the one installed in the `P4TRUST` file associated with the script's run-time environment, your script will (and should!) fail to connect to the server.

Changing your password

You can use P4Python to change your password, as shown in the following example:

```
from P4 import P4

p4 = P4()
p4.user = "bruno"
p4.password = "MyOldPassword"
p4.connect()

p4.run_password( "MyOldPassword", MyNewPassword" )

# p4.password is automatically updated with the encoded password
```

Timestamp conversion

Timestamp information in P4Python is normally represented as seconds since Epoch (with the exception of **`P4.Revision`**). To convert this data to a more useful format, use the following procedure:

```
import datetime

...

myDate = datetime.datetime.utcfromtimestamp( int( timestampValue ) )
```

Working with comments in specs

As of P4Python 2.3, comments in specs are preserved in the **`parse_<spectype>()`** and **`format_<spectype>()`** methods. This behavior can be circumvented by using **`parse_spec( '<spectype>', spec )`** and **`format_spec( '<spectype>', spec )`** instead of **`parse_<spectype>( spec )`** and **`format_<spectype>( spec )`**. For example:

```
p4 = P4()
p4.connect()

...

# fetch a client spec in raw format, no formatting:
specform = p4.run( 'client', '-o', tagged=False )[0]

# convert the raw document into a spec
client1 = p4.parse_client( specform )
```

```
# comments are preserved in the spec as well
print( client1.comment )

# comments can be updated
client1.comment += "# ... and now for something completely different"

# the comment is prepended to the spec ready to be sent to the user
formatted1 = p4.format_client( client1 )

# or you can strip the comments
client2 = p4.parse_spec( 'client', specform )
formatted2 = p4.format_spec( 'client', specform )
```

P4Python Classes

The `P4` module consists of several public classes:

- P4
- P4.P4Exception
- P4.DepotFile
- P4.Revision
- P4.Integration
- P4.Map
- P4.MergeData
- P4.Message
- P4.OutputHandler
- P4.Progress
- P4.Resolver
- P4.Spec

The following tables provide more details about each public class, including methods and attributes. Attributes are readable and writable unless indicated otherwise. They can be strings, objects, or integers.

You can set attributes in the P4() constructor or by using their setters and getters. For example:

```
import P4
p4 = P4.P4(client="myclient", port="1666")
p4.user = 'me'
```

P4

Helix Server client class. Handles connection and interaction with the Helix Server. There is one instance of each connection.

The following table lists attributes of the class **P4** in P4Python.

Attribute	Description
api_level	API compatibility level. (Lock server output to a specified server level.)
charset	Charset for Unicode servers.
client	**P4CLIENT**, the name of the client workspace to use.
cwd	Current working directory.
disable_tmp_cleanup	Disable cleanup of temporary objects.
encoding	Encoding to use when receiving strings from a non-Unicode server. If unset, use UTF8. Can be set to a legal Python encoding, or to **raw** to receive Python bytes instead of Unicode strings. Requires Python 3.
errors	An array containing the error messages received during execution of the last command.
exception_level	The exception level of the **P4** instance. Values can be: ■ **0** : no exceptions are raised. ■ **1** : only errors are raised as exceptions. ■ **2** : warnings are also raised as exceptions. The default value is 2.
handler	An output handler.
host	**P4HOST**, the name of the host used.
ignore_file	The path of the ignore file, **P4IGNORE**.
input	Input for the next command. Can be a string, a list or a dictionary.
maxlocktime	MaxLockTime used for all following commands
maxresults	MaxResults used for all following commands. This command must be set before the connection is made. If the command is set after the connection is made, the command is ignored.
maxscanrows	MaxScanRows used for all following commands.
messages	An array of **P4.Message** objects, one for each message sent by the server.

Attribute	Description
`p4config_file`	The location of the configuration file used (`P4CONFIG`). This attribute is read-only.
`password`	`P4PASSWD`, the password used.
`port`	`P4PORT`, the port used for the connection.
`prog`	The name of the script.
`progress`	A progress indicator.
`server_case_insensitive`	Detect whether or not the server is case sensitive.
`server_level`	Returns the current Helix Server level.
`server_unicode`	Detect whether or not the server is in Unicode mode.
`streams`	To disable streams support, set the value to `0` or `False`. By default, streams output is enabled for servers at 2011.1 or higher.
`tagged`	To disable tagged output for the following commands, set the value to 0 or False. By default, tagged output is enabled.
`track`	To enable performance tracking for the current connection, set the value to 1 or True. By default, server tracking is disabled.
`track_output`	If performance tracking is enabled, returns an array containing performance tracking information received during execution of the last command.
`ticket_file`	`P4TICKETS`, the ticket file location used.
`user`	`P4USER`, the user under which the connection is run.
`version`	The version of the script.
`warnings`	An array containing the warning messages received during execution of the last command.

The following table lists all public methods of the class `P4`. Many methods are wrappers around `P4.run()`, which sends a command to Helix Server. Such methods are provided for your convenience.

Method	Description
at_exception_level()	In the context of a **with** statement, temporarily set the exception level for the duration of a block.
clone()	Clones from another Perforce service into a local Perforce service, and returns a new **P4** object.
connect()	Connects to the Helix Server.
connected()	Returns **True** if connected and the connection is alive, otherwise **False**.
delete_<*spectype*>()	Deletes the spec <*spectype*>. Equivalent to: `P4.run( "<spectype>", "-d" )`
disconnect()	Disconnects from the Helix Server.
env()	Get the value of a Helix Server environment variable, taking into account **P4CONFIG** files and (on Windows or OS X) the registry or user preferences.
fetch_<*spectype*>()	Fetches the spec <*spectype*>. Equivalent to: `p4.run( "<spectype>", "-o" ).pop( 0 )`
format_<*spectype*>()	Converts the spec <*spectype*> into a string.
identify()	Returns a string identifying the P4Python module.
init()	Initializes a new personal (local) Helix Server, and returns a new **P4** object.
is_ignored()	Determines whether a particular file is ignored via the **P4IGNORE** feature.
iterate_<*spectype*>()	Iterate through specs of form <*spectype*>.
P4()	Returns a new **P4** object.
parse_<*spectype*>()	Parses a string representation of the spec <*spectype*> and returns a dictionary.

Method	Description
run()	Runs a command on the server. Needs to be connected, or an exception is raised.
run_*cmd*()	Runs the command *cmd*. Equivalent to: `P4.run( "command" )`
run_filelog()	This command returns a list of **P4.DepotFile** objects. Specialization for the **P4.run()** method.
run_login()	Logs in using the specified password or ticket.
run_password()	Convenience method: updates the password. Takes two arguments: *oldpassword*, *newpassword*
run_resolve()	Interface to **p4 resolve**.
run_submit()	Convenience method for submitting changelists. When invoked with a change spec, it submits the spec. Equivalent to: `p4.input = myspecp4.run( "submit", "-i" )`
run_tickets()	Interface to **p4 tickets**.
save_<*spectype*>()	Saves the spec <*spectype*>. Equivalent to: `P4.run( "<spectype>", "-i" )`
set_env()	On Windows or OS X, set a variable in the registry or user preferences.
temp_client()	Creates a temporary client.
while_tagged()	In the context of a **with** statement, temporarily toggle tagged behavior for the duration of a block.

P4.P4Exception

Exception class. Instances of this class are raised when errors and/or (depending on the **exception_level** setting) warnings are returned by the server. The exception contains the errors in the form of a string. **P4Exception** is a subclass of the standard Python **Exception** class.

P4.DepotFile

Container class returned by `P4.run_filelog()`. Contains the name of the depot file and a list of `P4.Revision` objects.

Attribute	Description
`depotFile`	Name of the depot file.
`revisions`	List of `P4.Revision` objects

P4.Revision

Container class containing one revision of a `P4.DepotFile` object.

Attribute	Description
`action`	Action that created the revision.
`change`	Changelist number
`client`	Client workspace used to create this revision.
`desc`	Short change list description.
`depotFile`	The name of the file in the depot.
`digest`	MD5 digest of the revision.
`fileSize`	File size of this revision.
`integrations`	List of `P4.Integration` objects.
`rev`	Revision.
`time`	Timestamp (as `datetime.datetime` object)
`type`	File type.
`user`	User that created this revision.

P4.Integration

Container class containing one integration for a `P4.Revision` object.

Attribute	Description
`how`	Integration method (merge/branch/copy/ignored).
`file`	Integrated file.

Attribute	Description
`srev`	Start revision.
`erev`	End revision.

P4.Map

A class that allows users to create and work with Helix Server mappings without requiring a connection to the Helix Server.

Method	Description
`P4.Map()`	Construct a new Map object (class method).
`join()`	Joins two maps to create a third (class method).
`clear()`	Empties a map.
`count()`	Returns the number of entries in a map.
`is_empty()`	Tests whether or not a map object is empty.
`insert()`	Inserts an entry into the map.
`translate()`	Translate a string through a map.
`includes()`	Tests whether a path is mapped.
`reverse()`	Returns a new mapping with the left and right sides reversed.
`lhs()`	Returns the left side as an array.
`rhs()`	Returns the right side as an array.
`as_array()`	Returns the map as an array

P4.MergeData

Class encapsulating the context of an individual merge during execution of a **`p4 resolve`** command. Passed to `P4.run_resolve()`.

Attribute	Description
`your_name`	Returns the name of "your" file in the merge. (file in workspace)
`their_name`	Returns the name of "their" file in the merge. (file in the depot)

Attribute	Description
`base_name`	Returns the name of "base" file in the merge. (file in the depot)
`your_path`	Returns the path of "your" file in the merge. (file in workspace)
`their_path`	Returns the path of "their" file in the merge. (temporary file on workstation into which `their_name` has been loaded)
`base_path`	Returns the path of the base file in the merge. (temporary file on workstation into which `base_name` has been loaded)
`result_path`	Returns the path to the merge result. (temporary file on workstation into which the automatic merge performed by the server has been loaded)
`merge_hint`	Returns hint from server as to how user might best resolve merge.

The **`P4.MergeData`** class also has one method:

`run_merge()`	If the environment variable `P4MERGE` is defined, run it and return a boolean based on the return value of that program.

P4.Message

Class for handling error messages in Helix Server.

Method	Description
`severity`	Returns the severity of the message.
`generic`	Returns the generic class of the error.
`msgid`	Returns the unique ID of the error message.

P4.OutputHandler

Handler class that provides access to streaming output from the server; set **`P4.handler`** to an instance of a subclass of **`P4.OutputHandler`** to enable callbacks:

Method	Description
`outputBinary`	Process binary data.
`outputInfo`	Process tabular data.

Method	Description
`outputMessage`	Process information or errors.
`outputStat`	Process tagged output.
`outputText`	Process text data.

P4.Progress

Handler class that provides access to progress indicators from the server; set `P4.progress` to an instance of a subclass of `P4.Progress` to enable callbacks:

Method	Description
`init()`	Initialize progress indicator as designated type.
`setTotal()`	Total number of units (if known).
`setDescription()`	Description and type of units to be used for progress reporting.
`update()`	If non-zero, user has requested a cancellation of the operation.
`done()`	If non-zero, operation has failed.

P4.Resolver

Class for handling resolves in Helix Server.

Method	Description
`resolve()`	Perform a resolve and return the resolve decision as a string.

P4.Spec

Class allowing access to the fields in a Helix Server specification form.

Attribute	Description
`_fieldname`	Value associated with the field named *fieldname*.
`comments`	Array containing comments in a spec object.
`permitted_fields`	Array containing the names of the fields that are valid for this spec object.

Class P4

Description

Main interface to the Python client API.

This module provides an object-oriented interface to Helix Server, the Perforce version constrol system. Data is returned in Python arrays and dictionaries (hashes) and input can also be supplied in these formats.

Each `P4` object represents a connection to the Helix Server, and multiple commands may be executed (serially) over a single connection (which of itself can result in substantially improved performance if executing long sequences of Helix Server commands).

1. Instantiate your `P4` object.
2. Specify your Helix Server client environment:
   - `client`
   - `host`
   - `password`
   - `port`
   - `user`
3. Set any options to control output or error handling:
   - `exception_level`
4. Connect to the Perforce service.

   The Helix Server protocol is not designed to support multiple concurrent queries over the same connection. Multithreaded applications that use the C++ API or derived APIs (including P4Python) should ensure that a separate connection is used for each thread, or that only one thread may use a shared connection at a time.
5. Run your Helix Server commands.
6. Disconnect from the Perforce service.

Instance Attributes

p4.api_level -> int

Contains the API compatibility level desired. This is useful when writing scripts using Helix Server commands that do not yet support tagged output. In these cases, upgrading to a later server that supports tagged output for the commands in question can break your script. Using this method allows you to lock your script to the output format of an older Helix Server release and facilitate seamless upgrades. Must be called before calling `P4.connect()`.

```
from P4 import P4
p4 = P4()
p4.api_level = 67 # Lock to 2010.1 format
p4.connect()
...
p4.disconnect
```

For more information about the API integer levels, see the Support Knowledgebase article, "Helix Client Protocol Levels".

p4.charset -> string

Contains the character set to use when connecting to a Unicode enabled server. Do not use when working with non-Unicode-enabled servers. By default, the character set is the value of the **`P4CHARSET`** environment variable. If the character set is invalid, this method raises a **`P4Exception`**.

```
from P4 import P4
p4 = P4()
p4.client = "www"
p4.charset = "iso8859-1"
p4.connect()
p4.run_sync()
p4.disconnect()
```

p4.client -> string

Contains the name of your client workspace. By default, this is the value of the **`P4CLIENT`** taken from any **`P4CONFIG`** file present, or from the environment according to the normal Helix Server conventions.

p4.cwd -> string

Contains the current working directly. Can be set prior to executing any Helix Server command. Sometimes necessary if your script executes a **`chdir()`** as part of its processing.

```
from P4 import P4
p4 = P4()
p4.cwd = "/home/bruno"
```

p4.disable_tmp_cleanup -> string

Invoke this prior to connecting if you need to use multiple **`P4`** connections in parallel in a multi-threaded Python application.

```
from P4 import P4
p4 = P4()
p4.disable_tmp_cleanup()
p4.connect()
...
p4.disconnect()
```

p4.encoding -> string

When decoding strings from a non-Unicode server, strings are assumed to be encoded in UTF8. To use another encoding, set **`p4.encoding`** to a legal Python encoding, or **`raw`** to receive Python bytes instead of a Unicode string. Available only when compiled with Python 3.

p4.errors -> list (read-only)

Returns an array containing the error messages received during execution of the last command.

```
from P4 import P4, P4Exceptionp4 = P4()

try:
  p4.connect()
  p4.exception_level = 1
  # ignore "File(s) up-to-date"s
  files = p4.run_sync()

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.exception_level -> int

Configures the events which give rise to exceptions. The following three levels are supported:

- `0` : disables all exception handling and makes the interface completely procedural; you are responsible for checking the **`p4.errors`** and **`p4.warnings`** arrays.
- `1` : causes exceptions to be raised only when errors are encountered.
- `2` : causes exceptions to be raised for both errors and warnings. This is the default.

For example:

```
from P4 import P4
p4 = P4()
p4.exception_level = 1
p4.connect()   # P4Exception on failure
p4.run_sync()  # File(s) up-to-date is a warning - no exception raised
p4.disconnect()
```

p4.handler -> handler

Set the output handler to a subclass of **`P4.OutputHandler`**.

p4.host -> string

Contains the name of the current host. It defaults to the value of **`P4HOST`** taken from any **`P4CONFIG`** file present, or from the environment as per the usual Helix Server convention. Must be called before connecting to the Helix Server.

```
from P4 import P4
p4 = P4()
p4.host = "workstation123.perforce.com"
p4.connect()
...
p4.disconnect()
```

p4.ignore_file -> string

Contains the path of the ignore file. It defaults to the value of **`P4IGNORE`**. Set **`P4.ignore_file`** prior to calling **`P4.is_ignored()`**.

```
from P4 import P4
p4 = P4()
p4.connect()
p4.ignore_file = "/home/bruno/workspace/.ignore"
p4.disconnect()
```

p4.input -> string | dict | list

Contains input for the next command.

Set this attribute prior to running a command that requires input from the user. When the command requests input, the specified data is supplied to the command. Typically, commands of the form **p4 *cmd* -i** are invoked using the **P4.save_<*spectype*>()** methods, which retrieve the value from **p4.input** internally; there is no need to set **p4.input** when using the **P4.save_<*spectype*>()** shortcuts.

You may pass a string, a hash, or (for commands that take multiple inputs from the user) an array of strings or hashes. If you pass an array, note that the first element of the array will be popped each time Helix Server asks the user for input.

For example, the following code supplies a description for the default changelist and then submits it to the depot:

```
from P4 import P4
p4 = P4()
p4.connect()
change = p4.run_change( "-o" )[0]
change[ "Description" ] = "Autosubmitted changelist"
p4.input = change
p4.run_submit( "-i" )
p4.disconnect()
```

p4.maxlocktime -> int

Limit the amount of time (in milliseconds) spent during data scans to prevent the server from locking tables for too long. Commands that take longer than the limit will be aborted. The limit remains in force until you disable it by setting it to zero. See **p4 help maxlocktime** for information on the commands that support this limit.

p4.maxresults -> int

Limit the number of results Helix Server permits for subsequent commands. Commands that produce more than this number of results will be aborted. The limit remains in force until you disable it by setting it to zero. See **p4 help maxresults** for information on the commands that support this limit.

> **Note**
> This command must be set before the connection is made. If the command is set after the connection is made, the command is ignored.

p4.maxscanrows -> int

Limit the number of database records Helix Server scans for subsequent commands. Commands that attempt to scan more than this number of records will be aborted. The limit remains in force until you disable it by setting it to zero. See **`p4 help maxscanrows`** for information on the commands that support this limit.

p4.messages -> list (read-only)

Returns a list of **`P4.Message`** objects, one for each message (info, warning or error) sent by the server.

p4.p4config_file -> string (read-only)

Contains the name of the current **`P4CONFIG`** file, if any. This attribute cannot be set.

p4.password -> string

Contains your Helix Server password or login ticket. If not used, takes the value of **`P4PASSWD`** from any **`P4CONFIG`** file in effect, or from the environment according to the normal Helix Server conventions.

This password is also used if you later call **`p4.run_login()`** to log in using the 2003.2 and later ticket system. After running **`p4.run_login()`**, the attribute contains the ticket allocated by the server.

```
from P4 import P4
p4 = P4()
p4.password = "mypass"
p4.connect()
p4.run_login()
```

p4.port -> string

Contains the host and port of the Helix Server to which you want to connect. It defaults to the value of **`P4PORT`** in any **`P4CONFIG`** file in effect, and then to the value of **`P4PORT`** taken from the environment.

```
from P4 import P4
p4 = P4()
p4.port = "localhost:1666"
p4.connect()
...
```

p4.prog -> string

Contains the name of the program, as reported to Helix Server system administrators running **`p4 monitor show -e`**. The default is **`unnamed p4-python script`**.

```
from P4 import P4
p4 = P4()
p4.prog = "sync-script"
print p4.prog
p4.connect
...
```

p4.progress -> progress

Set the progress indicator to a subclass of **`P4.Progress`**.

p4.server_case_insensitive -> boolean

Detects whether or not the server is case-sensitive.

p4.server_level -> int (read-only)

Returns the current Helix Server level. Each iteration of the Helix Server is given a level number. As part of the initial communication this value is passed between the client application and the Helix Server. This value is used to determine the communication that the Helix Server will understand. All subsequent requests can therefore be tailored to meet the requirements of this server level.

This attribute is 0 before the first command is run, and is set automatically after the first communication with the server.

For more information about the Helix Server version levels, see the Support Knowledgebase article, "Helix Server Version Levels".

p4.server_unicode -> boolean

Detects whether or not the server is in Unicode mode.

p4.streams -> int

If 1 or True, **`p4.streams`** enables support for streams. By default, streams support is enabled at 2011.1 or higher (**`api_level`** >= 70). Raises a **`P4Exception`** if you attempt to enable streams on a pre-2011.1 server. You can enable or disable support for streams both before and after connecting to the server.

```
from P4 import P4
p4 = P4()
p4.streams = False
print p4.streams
```

p4.tagged -> int

If 1 or True, **`p4.tagged`** enables tagged output. By default, tagged output is on.

```
from P4 import P4
p4 = P4()
p4.tagged = False
print p4.tagged
```

p4.ticket_file -> string

Contains the location of the **`P4TICKETS`** file.

p4.track -> boolean

If set to 1 or True, **`p4.track`** indicates that server performance tracking is enabled for this connection. By default, performance tracking is disabled.

p4.track_output -> list (read-only)

If performance tracking is enabled with **`p4.track`**, returns an array containing the performance data received during execution of the last command.

```
from P4 import P4
p4 = P4()
p4.track = 1
p4.run_info()
print p4.track_output
```

p4.user -> string

Contains the Helix Server username. It defaults to the value of **`P4USER`** taken from any **`P4CONFIG`** file present, or from the environment as per the usual Helix Server convention.

```
from P4 import P4
p4 = P4()
p4.user = "bruno"
p4.connect()
...
p4.disconnect()
```

p4.version -> string

Contains the version of the program, as reported to Helix Server system administrators in the server log.

```
from P4 import P4
p4 = P4()
p4.version = "123"
print p4.version
p4.connect()
...
p4.disconnect()
```

p4.warnings -> list (read-only)

Contains the array of warnings that arose during execution of the last command.

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  p4.exception_level = 2 # File(s) up-to-date is a warning
  files = p4.run_sync()

except P4Exception, ex:
  for w in p4.warnings:
      print w

finally:
  p4.disconnect()
```

Class Methods

P4.P4()

Construct a new **P4** object. For example:

```
import P4
p4 = P4.P4()
```

P4.clone(arguments...)

Clone from another Perforce service into a local Helix Server, and returns a new **P4** object.

`P4.clone()` requires specification of the port of the source Perforce service from which files and version history should be cloned from, and either a *remotespec* or a *filespec* that specify which files and history to clone. For example, to clone using a *remotespec*:

```
import P4
p4 = P4.clone( "-p", "port", "-r", "remotespec" )
```

or to clone using a *filespec*:

```
import P4
p4 = P4.clone( "-p", "port", "-f", "filespec" )
```

The cloned instance inherits the case sensitivity and Unicode settings from the source Perforce service.

> **Note**
> All of the additional DVCS commands, such as **`p4 push`** or **`p4 switch`**, are available automatically in the usual fashion. For example: **`p4.run_push()`**. See p4.run_<cmd>() for details.

P4.identify()

Return the version of P4Python that you are using.

```
python -c "from P4 import P4; print P4.identify()"
```

The read-only string attributes **`PATCHLEVEL`** and **`OS`** are also available to test an installation of P4Python without having to parse the output of **`P4.identify()`**.

If applicable, **`P4.identify()`** also reports the version of the OpenSSL library used for building the underlying Helix C/C++ API with which P4Python was built.

P4.init([arguments])

Initializes a new, personal (local) Helix Server, and returns a new **`P4`** object.

Without any arguments, **`P4.init()`** creates a new DVCS server in the current working directory, using the settings for case sensitivity and Unicode support from current environment variables.

`P4.init()` accepts the following keyword arguments:

Keyword	Explanation	Example
`client`	Workspace and server name	**`client="sknop-dvcs"`**
`user`	Helix Server username used for pushing	**`user="sven_erik_knop"`**
`directory`	local path of the root directory for the new server	**`directory="/tmp/test-dvcs"`**

Keyword	Explanation	Example
`casesensitive`	specify case sensitivity	`casesensitive=False`
`unicode`	specify whether Unicode is enabled	`unicode=True`

```
import P4
p4 = P4.init( directory="/Users/sknop/dvcs/" )
p4.connect()
# ...
p4.disconnect()
```

The **`P4`** instance returned by **`P4.init()`** has the port, user, and client workspace already set; all that is required for you is to connect to the server to perform any commands. Connection is not automatic, to give you an opportunity to set any protocol parameters; these parameters can only be set once before a connection is established.

> **Note**
> All of the additional DVCS commands, such as **`p4 push`** or **`p4 switch`**, are available automatically in the usual fashion. For example: **`p4.run_push()`**. See p4.run_<cmd>() for details.

p4.iterate_<spectype>(arguments) -> P4.Spec

The **`iterate_<spectype>()`** methods are shortcut methods that allow you to quickly iterate through clients, labels, branches, etc. Valid <*spectypes*> are **`clients`**, **`labels`**, **`branches`**, **`changes`**, **`streams`**, **`jobs`**, **`users`**, **`groups`**, **`depots`** and **`servers`**. Valid arguments are any arguments that would be valid for the corresponding **`run_<spectype>()`** command.

For example:

```
for client in p4.iterate_clients():
  # do something with the client spec
```

is equivalent to:

```
for c in p4.run_clients():
  client = p4.fetch_client( c['client'] )
```

Instance Methods

p4.at_exception_level()

In the context of a **`with`** statement, temporarily set the exception level for a block. For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.at_exception_level( P4.RAISE_ERRORS ):
  # no exceptions for warnings
  p4.run_sync( "//depot/main/..." )

# exceptions back to normal...
```

p4.connect()

Initializes the Helix Server client and connects to the server.

If the connection is successfully established, returns **`None`**. If the connection fails and **`P4.exception_level`** is 0, returns False, otherwise raises a **`P4Exception`**. If already connected, prints a message.

```
from P4 import P4
p4 = P4()
p4.connect()
...
p4.disconnect()
```

`P4.connect()` returns a context management object that is usable with a **`with`** statement within a block; after the block is finished, the connection is automatically disconnected:

```
import P4
p4 = P4.P4()
with p4.connect():
  # block in context of connection
  ...

# p4 is disconnected outside the block
...
```

p4.connected() -> boolean

Returns **`true`** if connected to the Helix Server and the connection is alive, otherwise **`false`**.

```
from P4 import P4
p4 = P4()
```

```
print p4.connected()
p4.connect()
print p4.connected()
```

p4.delete_<spectype>([options], name) -> list

The **`delete_<spectype>()`** methods are shortcut methods that allow you to delete the definitions of clients, labels, branches, etc. These methods are equivalent to:

```
p4.run( "<spectype>", '-d', [options], "spec name" )
```

The following code uses **`P4.delete_client()`** to delete client workspaces that have not been accessed in more than 365 days:

```
from P4 import P4, P4Exception
from datetime import datetime, timedelta

now = datetime.now()
p4 = P4()

try:
  p4.connect()
  for client in p4.run_clients():
    atime = datetime.utcfromtimestamp( int( client[ "Access" ] ) )
    # If the client has not been accessed for a year, delete it
    if ( atime + timedelta( 365 ) ) < now :
      p4.delete_client( '-f', client[ "client" ] )

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.disconnect()

Disconnect from the Helix Server. Call this method before exiting your script.

```
from P4 import P4
p4 = P4()
```

```
p4.connect()
...
p4.disconnect()
```

p4.env(var)

Get the value of a Helix Server environment variable, taking into account **P4CONFIG** files and (on Windows or OS X) the registry or user preferences.

```
from P4 import P4
p4 = P4()

print p4.env( "P4PORT" )
```

p4.fetch_<spectype>() -> P4.Spec

The **fetch_<*spectype*>()** methods are shortcuts for running **p4.run("<*spectype*>", "-o").pop(0)**. For example:

```
label      = p4.fetch_label( "labelname" )
change     = p4.fetch_change( changeno )
clientspec = p4.fetch_client( "clientname" )
```

are equivalent to:

```
label      = p4.run( "label", "-o", "labelname" )[0]
change     = p4.run( "change", "-o", changeno )[0]
clientspec = p4.run( "client", "-o", "clientname" )[0]
```

p4.format_spec("<spectype>", dict) -> string

Converts the fields in the dict containing the elements of a Helix Server form (spec) into the string representation familiar to users. The first argument is the type of spec to format: for example, client, branch, label, and so on. The second argument is the hash to parse.

There are shortcuts available for this method. You can use **p4.format_<*spectype*>(dict)** instead of **p4.format_spec("<*spectype*>", dict)**, where <*spectype*> is the name of a Helix Server spec, such as client, label, etc.

p4.format_<spectype>(dict) -> string

The **format_<*spectype*>()** methods are shortcut methods that allow you to quickly fetch the definitions of clients, labels, branches, etc. They're equivalent to:

```
p4.format_spec( "<spectype>", dict )
```

p4.is_ignored("<path>") -> boolean

Returns **`true`** if the <*path*> is ignored via the **`P4IGNORE`** feature. The <*path*> can be a local relative or absolute path.

```
from P4 import P4
p4 = P4()

p4.connect()
if ( p4.is_ignored( "/home/bruno/workspace/file.txt" ):
  print "Ignored."
else:
  print "Not ignored."

p4.disconnect()
```

p4.parse_spec("<spectype>", string) -> P4.Spec

Parses a Helix Server form (spec) in text form into a Python dict using the spec definition obtained from the server. The first argument is the type of spec to parse: **`client`**, **`branch`**, **`label`**, and so on. The second argument is the string buffer to parse.

There are shortcuts available for this method. You can use:

```
p4.parse_<spectype>( buf )
```

instead of:

```
p4.parse_spec( "<spectype>", buf )
```

where <*spectype*> is one of **`client`**, **`branch`**, **`label`**, and so on.

p4.parse_<spectype>(string) -> P4.Spec

This is equivalent to:

```
p4.parse_spec( "<spectype>", string )
```

For example, **`parse_job( myJob )`** converts the String representation of a job spec into a Spec object.

To parse a spec, **P4** needs to have the spec available. When not connected to the Helix Server, **P4** assumes the default format for the spec, which is hardcoded. This assumption can fail for jobs if the server's jobspec has been modified. In this case, your script can load a job from the server first with the command **p4.fetch_job('somename')**, and **P4** will cache and use the spec format in subsequent **p4.parse_job()** calls.

p4.run("<cmd>", [arg, ...])

Base interface to all the run methods in this API. Runs the specified Helix Server command with the arguments supplied. Arguments may be in any form as long as they can be converted to strings by **str()**. However, each command's options should be passed as quoted and comma-separated strings, with no leading space. For example:

```
p4.run("print","-o","test-print","-q","//depot/Jam/MAIN/src/expand.c")
```

Failing to pass options in this way can result in confusing error messages.

The **p4.run()** method returns a list of results whether the command succeeds or fails; the list may, however, be empty. Whether the elements of the array are strings or dictionaries depends on:

1. server support for tagged output for the command, and
2. whether tagged output was disabled by calling **p4.tagged = False**.

In the event of errors or warnings, and depending on the exception level in force at the time, **p4.run()** raises a **P4Exception**. If the current exception level is below the threshold for the error/warning, **p4.run()** returns the output as normal and the caller must explicitly review **p4.errors** and **p4.warnings** to check for errors or warnings.

```
from P4 import P4
p4 = P4()
p4.connect()
spec = p4.run( "client", "-o" )[0]
p4.disconnect()
```

Shortcuts are available for **p4.run()**. For example:

```
p4.run_command(args)
```

is equivalent to:

```
p4.run( "command", args )
```

There are also some shortcuts for common commands such as editing Helix Server forms and submitting. For example, this:

```
from P4 import P4
p4 = P4()
p4.connect()
clientspec = p4.run_client( "-o" ).pop( 0 )
```

```
clientspec[ "Description" ] = "Build client"
p4.input = clientspec
p4.run_client( "-i" )
p4.disconnect()
```

…may be shortened to:

```
from P4 import P4
p4 = P4()
p4.connect()
clientspec = p4.fetch_client()
clientspec[ "Description" ] = "Build client"
p4.save_client( clientspec )
p4.disconnect()
```

The following are equivalent:

Shortcut	Equivalent to
`p4.delete_<spectype>()`	**`p4.run( "<spectype>", "-d ")`**
`p4.fetch_<spectype>()`	**`p4.run( "<spectype>", "-o ").shift`**
`p4.save_<spectype>( spec )`	**`p4.input = spec`** **`p4.run( "<spectype>", "-i")`**

As the commands associated with **`p4.fetch_<spectype>()`** typically return only one item, these methods do not return an array, but instead return the first result element.

For convenience in submitting changelists, changes returned by **`p4.fetch_change()`** can be passed to **`p4.run_submit()`**. For example:

```
from P4 import P4
p4 = P4()
p4.connect()

spec = p4.fetch_change()
spec[ "Description" ] = "Automated change"
p4.run_submit( spec )
p4.disconnect()
```

p4.run_<cmd>()

Shorthand for:

```
p4.run( "<cmd>", arguments... )
```

p4.run_filelog(<fileSpec>) -> list

Runs a **`p4 filelog`** on the *fileSpec* provided and returns an array of **`P4.DepotFile`** results (when executed in tagged mode), or an array of strings when executed in nontagged mode. By default, the raw output of **`p4 filelog`** is tagged; this method restructures the output into a more user-friendly (and object-oriented) form.

For example:

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  for r in p4.run_filelog( "index.html" )[0].revisions:
    for i in r.integrations:
      # Do something

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.run_login(<arg>...) -> list

Runs **`p4 login`** using a password or ticket set by the user.

p4.run_password(oldpass, newpass) -> list

A thin wrapper to make it easy to change your password. This method is (literally) equivalent to the following:

```
p4.input( [ oldpass, newpass, newpass ] )
p4.run( "password" )
```

For example:

```
from P4 import P4, P4Exception
p4 = P4()
```

```
p4.password = "myoldpass"

try:
  p4.connect()
  p4.run_password( "myoldpass", "mynewpass" )

except P4Exception:
  for e in p4.errors:
    print e

finally:
  p4.disconnect()
```

p4.run_resolve([<resolver>], [arg...]) -> list

Run a **`p4 resolve`** command. Interactive resolves require the *<resolver>* parameter to be an object of a class derived from **`P4.Resolver`**. In these cases, the *P4.Resolver.resolve()* method is called to handle the resolve. For example:

```
p4.run_resolve ( resolver=MyResolver() )
```

To perform an automated merge that skips whenever conflicts are detected:

```
class MyResolver( P4.Resolver ):
  def resolve( self, mergeData ):
    if not mergeData.merge_hint == "e":
      return mergeData.merge_hint
    else:
      return "s" # skip the resolve, there is a conflict
```

In non-interactive resolves, no **`P4.Resolver`** object is required. For example:

```
p4.run_resolve ( "-at" )
```

p4.run_submit([hash], [arg...]) -> list

Submit a changelist to the server. To submit a changelist, set the fields of the changelist as required and supply any flags:

```
change = p4.fetch_change()
change._description = "Some description"
p4.run_submit( "-r", change )
```

You can also submit a changelist by supplying the arguments as you would on the command line:

```
p4.run_submit( "-d", "Some description", "somedir/..." )
```

p4.run_tickets( ) -> list

`p4.run_tickets()` returns an array of lists of the form (**`p4port`**, **`user`**, **`ticket`**) based on the contents of the local tickets file.

p4.save_<spectype>()>

The **`save_<spectype>()`** methods are shortcut methods that allow you to quickly update the definitions of clients, labels, branches, etc. They are equivalent to:

```
p4.input = dictOrString
p4.run( "<spectype>", "-i" )
```

For example:

```
from P4 import P4, P4Exception
p4 = P4()

try:
  p4.connect()
  client = p4.fetch_client()
  client[ "Owner" ] = p4.user
  p4.save_client( client )

except P4Exception:
  for e in p4.errors:
    print e

finally:
p4.disconnect()
```

p4.set_env(var, value)

On Windows or OS X, set a variable in the registry or user preferences. To unset a variable, pass an empty string as the second argument. On other platforms, an exception is raised.

```
p4.set_env = ( "P4CLIENT", "my_workspace" )
p4.set_env = ( "P4CLIENT", "" )
```

p4.temp_client("<prefix>", "<template>")

Creates a temporary client, using the prefix *<prefix>* and based upon a client template named *<template>*, then switches **`P4.client`** to the new client, and provides a temporary root directory. The prefix makes is easy to exclude the workspace from the spec depot.

This is intended to be used with a **`with`** statement within a block; after the block is finished, the temp client is automatically deleted and the temporary root is removed.

For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.temp_client( "temp", "my_template" ) as t:
  p4.run_sync()
  p4.run_edit( "foo" )
  p4.run_submit( "-dcomment" )
```

p4.while_tagged(boolean)

In the context of a **`with`** statement, enable or disable tagged behavior for the duration of a block. For example:

```
from P4 import P4
p4 = P4()
p4.connect()
with p4.while_tagged( False ):
  # tagged output disabled for this block
  print p4.run_info()

# tagged output back to normal
...
```

Class P4.P4Exception

Description

Instances of this class are raised when **`P4`** encounters an error or a warning from the server. The exception contains the errors in the form of a string. **`P4Exception`** is a shallow subclass of the standard Python Exception class.

Class Attributes

None.

Class Methods

None.

Class P4.DepotFile

Description

Utility class providing easy access to the attributes of a file in a Helix Server depot. Each **`P4.DepotFile`** object contains summary information about the file and a list of revisions (**`P4.Revision`** objects) of that file. Currently, only the **`P4.run_filelog()`** method returns a list of **`P4.DepotFile`** objects.

Instance Attributes

df.depotFile -> string

Returns the name of the depot file to which this object refers.

df.revisions -> list

Returns a list of **`P4.Revision`** objects, one for each revision of the depot file.

Class Methods

None.

Instance Methods

None.

Class P4.Revision

Description

Utility class providing easy access to the revisions of **`P4.DepotFile`** objects. Created by **`P4.run_filelog()`**.

Instance Attributes

rev.action -> string

Returns the name of the action which gave rise to this revision of the file.

rev.change -> int

Returns the change number that gave rise to this revision of the file.

rev.client -> string

Returns the name of the client from which this revision was submitted.

rev.depotFile -> string

Returns the name of the depot file to which this object refers.

rev.desc -> string

Returns the description of the change which created this revision. Note that only the first 31 characters are returned unless you use **`p4 filelog -L`** for the first 250 characters, or **`p4 filelog -l`** for the full text.

rev.digest -> string

Returns the MD5 digest of this revision.

rev.fileSize -> string

Returns this revision's size in bytes.

rev.integrations -> list

Returns the list of **`P4.Integration`** objects for this revision.

rev.rev -> int

Returns the number of this revision of the file.

rev.time -> datetime

Returns the date/time that this revision was created.

rev.type -> string

Returns this revision's Helix Server filetype.

rev.user -> string

Returns the name of the user who created this revision.

Class Methods

None.

Instance Methods

None.

Class P4.Integration

Description

Utility class providing easy access to the details of an integration record. Created by `P4.run_filelog()`.

Instance Attributes

integ.how -> string

Returns the type of the integration record - how that record was created.

integ.file -> string

Returns the path to the file being integrated to/from.

integ.srev -> int

Returns the start revision number used for this integration.

integ.erev -> int

Returns the end revision number used for this integration.

Class Methods

None.

Instance Methods

None.

Class P4.Map

Description

The **`P4.Map`** class allows users to create and work with Helix Server mappings, without requiring a connection to a Helix Server.

Instance Attributes

None.

Class Methods

P4.Map([list]) -> P4.Map

Constructs a new **`P4.Map`** object.

P4.Map.join (map1, map2) -> P4.Map

Join two **`P4.Map`** objects and create a third.

The new map is composed of the left-hand side of the first mapping, as joined to the right-hand side of the second mapping. For example:

```
# Map depot syntax to client syntax
client_map = P4.Map()
client_map.insert( "//depot/main/...", "//client/..." )

# Map client syntax to local syntax
client_root = P4.Map()
client_root.insert( "//client/...", "/home/bruno/workspace/..." )

# Join the previous mappings to map depot syntax to local syntax
local_map = P4.Map.join( client_map, client_root )
local_path = local_map.translate( "//depot/main/www/index.html" )

# local_path is now /home/bruno/workspace/www/index.html
```

Instance Methods

map.clear()

Empty a map.

map.count() -> int

Return the number of entries in a map.

map.is_empty() -> boolean

Test whether a map object is empty.

map.insert(string ...)

Inserts an entry into the map.

May be called with one or two arguments. If called with one argument, the string is assumed to be a string containing either a half-map, or a string containing both halves of the mapping. In this form, mappings with embedded spaces must be quoted. If called with two arguments, each argument is assumed to be half of the mapping, and quotes are optional.

```
# called with two arguments:
map.insert( "//depot/main/...", "//client/..." )

# called with one argument containing both halves of the mapping:
map.insert( "//depot/live/... //client/live/..." )

# called with one argument containing a half-map:
# This call produces the mapping "depot/... depot/..."
map.insert( "depot/..." )
```

map.translate (string, [boolean]) -> string

Translate a string through a map, and return the result. If the optional second argument is `1`, translate forward, and if it is `0`, translate in the reverse direction. By default, translation is in the forward direction.

map.includes(string) -> boolean

Tests whether a path is mapped or not.

```
if map.includes( "//depot/main/..." ):
  ...
```

map.reverse() -> P4.Map

Return a new **`P4.Map`** object with the left and right sides of the mapping swapped. The original object is unchanged.

map.lhs() -> list

Returns the left side of a mapping as an array.

map.rhs() -> list

Returns the right side of a mapping as an array.

map.as_array() -> list

Returns the map as an array.

Class P4.MergeData

Description

Class containing the context for an individual merge during execution of a **`p4 resolve`**.

Instance Attributes

md.your_name -> string

Returns the name of "your" file in the merge. This is typically a path to a file in the workspace.

md.their_name -> string

Returns the name of "their" file in the merge. This is typically a path to a file in the depot.

md.base_name -> string

Returns the name of the "base" file in the merge. This is typically a path to a file in the depot.

md.your_path -> string

Returns the path of "your" file in the merge. This is typically a path to a file in the workspace.

md.their_path -> string

Returns the path of "their" file in the merge. This is typically a path to a temporary file on your local machine in which the contents of **`their_name`** have been loaded.

md.base_path -> string

Returns the path of the base file in the merge. This is typically a path to a temporary file on your local machine in which the contents of `base_name` have been loaded.

md.result_path -> string

Returns the path to the merge result. This is typically a path to a temporary file on your local machine in which the contents of the automatic merge performed by the server have been loaded.

md.merge_hint -> string

Returns the hint from the server as to how it thinks you might best resolve this merge.

Instance Methods

md.run_merge() -> boolean

If the environment variable `P4MERGE` is defined, `md.run_merge()` invokes the specified program and returns a boolean based on the return value of that program.

Class P4.Message

Description

`P4.Message` objects contain error or other diagnostic messages from the Helix Server; they are returned in `P4.messages`.

Script writers can test the severity of the messages in order to determine if the server message consisted of command output (`E_INFO`), warnings, (`E_WARN`), or errors (`E_FAILED` /`E_FATAL`).

Class Methods

None.

Instance Attributes

message.severity -> int

Severity of the message, which is one of the following values:

Value	Meaning
`E_EMPTY`	No error.

Value	Meaning
`E_INFO`	Informational message only.
`E_WARN`	Warning message only.
`E_FAILED`	Command failed.
`E_FATAL`	Severe error; cannot continue.

message.generic -> int

Returns the generic class of the error.

message.msgid -> int

Returns the unique ID of the message.

Class P4.OutputHandler

Description

The `P4.OutputHandler` class is a handler class that provides access to streaming output from the server. After defining the output handler, set `p4.handler` to an instance of a subclass of `P4.OutputHandler`, use `p4.using_handler( MyHandler() )`, or pass the handler as a named parameter for one statement only.

By default, `P4.OutputHandler` returns `REPORT` for all output methods. The different return options are:

Value	Meaning
`REPORT`	Messages added to output (don't handle, don't cancel)
`HANDLED`	Output is handled by class (don't add message to output).
`REPORT|CANCEL`	Operation is marked for cancel, message is added to output.
`HANDLED|CANCEL`	Operation is marked for cancel, message not added to output.

Class Methods

class MyHandler(P4.OutputHandler)

Constructs a new subclass of `P4.OutputHandler`.

Instance Methods

outputBinary -> int

Process binary data.

outputInfo -> int

Process tabular data.

outputMessage -> int

Process informational or error messages.

outputStat -> int

Process tagged data.

outputText -> int

Process text data.

Class P4.Progress

Description

The `P4.Progress` class is a handler class that provides access to progress indicators from the server. After defining the progress class, set `P4.progress` to an instance of a subclass of `P4.Progress`, use `p4.using_progress( MyProgress() )`, or pass the progress indicator as a named parameter for one statement only.

You must implement all five of the following methods: `init()`, `setDescription()`, `update()`, `setTotal()`, and `done()`, even if the implementation consists of trivially returning `0`.

Instance Attributes

None.

Class Methods

class MyProgress(P4.Progress)

Constructs a new subclass of `P4.Progress`.

Instance Methods

progress.init() -> int

Initialize progress indicator.

progress.setDescription(string, int) -> int

Description and type of units to be used for progress reporting.

progress.update() -> int

If non-zero, user has requested a cancellation of the operation.

progress.setTotal(<total>) -> int

Total number of units expected (if known).

progress.done() -> int

If non-zero, operation has failed.

Class P4.Resolver

Description

`P4.Resolver` is a class for handling resolves in Helix Server. It is intended to be subclassed, and for subclasses to override the **`resolve()`** method. When **`P4.run_resolve()`** is called with a **`P4.Resolver`** object, it calls the **`P4.Resolver.resolve()`** method of the object once for each scheduled resolve.

Instance Attributes

None.

Class Methods

None.

Instance Methods

resolver.resolve(self, mergeData) -> string

Returns the resolve decision as a string. The standard Helix Server resolve strings apply:

String	Meaning
`ay`	Accept Yours.
`at`	Accept Theirs.
`am`	Accept Merge result.
`ae`	Accept Edited result.
`s`	Skip this merge.
`q`	Abort the merge.

By default, all automatic merges are accepted, and all merges with conflicts are skipped. The `P4.Resolver.resolve()` method is called with a single parameter, which is a reference to a `P4.MergeData` object.

Class P4.Spec

Description

Utility class providing easy access to the attributes of the fields in a Helix Server form.

Only valid field names may be set in a `P4.Spec` object. Only the field name is validated, not the content. Attributes provide easy access to the fields.

Instance Attributes

spec._<fieldname> -> string

Contains the value associated with the field named <*fieldname*>.

spec.comment -> dict

Contains an array containing the comments associated with the spec object.

Class Methods

P4.Spec.new(dict) -> P4.Spec

Constructs a new `P4.Spec` object given an array of valid fieldnames.

spec.permitted_fields() -> dict

Returns a dictionary containing the names of fields that are valid in this spec object. This does not imply that values for all of these fields are actually set in this object, merely that you may choose to set values for any of these fields if you want to.

Instance Methods

None.

Glossary

A

access level

A permission assigned to a user to control which commands the user can execute. See also the 'protections' entry in this glossary and the 'p4 protect' command in the P4 Command Reference.

admin access

An access level that gives the user permission to privileged commands, usually super privileges.

APC

The Alternative PHP Cache, a free, open, and robust framework for caching and optimizing PHP intermediate code.

archive

1. For replication, versioned files (as opposed to database metadata). 2. For the 'p4 archive' command, a special depot in which to copy the server data (versioned files and metadata).

atomic change transaction

Grouping operations affecting a number of files in a single transaction. If all operations in the transaction succeed, all the files are updated. If any operation in the transaction fails, none of the files are updated.

avatar

A visual representation of a Swarm user or group. Avatars are used in Swarm to show involvement in or ownership of projects, groups, changelists, reviews, comments, etc. See also the "Gravatar" entry in this glossary.

B

base

The file revision, in conjunction with the source revision, used to help determine what integration changes should be applied to the target revision.

binary file type

A Helix Server file type assigned to a non-text file. By default, the contents of each revision are stored in full, and file revision is stored in compressed format.

branch

(noun) A set of related files that exist at a specific location in the Perforce depot as a result of being copied to that location, as opposed to being added to that location. A group of related files is often referred to as a codeline. (verb) To create a codeline by copying another codeline with the 'p4 integrate', 'p4 copy', or 'p4 populate' command.

branch form

The form that appears when you use the 'p4 branch' command to create or modify a branch specification.

branch mapping

Specifies how a branch is to be created or integrated by defining the location, the files, and the exclusions of the original codeline and the target codeline. The branch mapping is used by the integration process to create and update branches.

branch view

A specification of the branching relationship between two codelines in the depot. Each branch view has a unique name and defines how files are mapped from the originating codeline to the target codeline. This is the same as branch mapping.

broker

Helix Broker, a server process that intercepts commands to the Helix Server and is able to run scripts on the commands before sending them to the Helix Server.

C

change review

The process of sending email to users who have registered their interest in changelists that include specified files in the depot.

changelist

A list of files, their version numbers, the changes made to the files, and a description of the changes made. A changelist is the basic unit of versioned work in Helix Server. The changes specified in the changelist are not stored in the depot until the changelist is submitted to the depot. See also atomic change transaction and changelist number.

changelist form

The form that appears when you modify a changelist using the 'p4 change' command.

changelist number

An integer that identifies a changelist. Submitted changelist numbers are ordinal (increasing), but not necessarily consecutive. For example, 103, 105, 108, 109. A pending changelist number might be assigned a different value upon submission.

check in

To submit a file to the Helix Server depot.

check out

To designate one or more files for edit.

checkpoint

A backup copy of the underlying metadata at a particular moment in time. A checkpoint can recreate db.user, db.protect, and other db.* files. See also metadata.

classic depot

A repository of Helix Server files that is not streams-based. The default depot name is depot. See also default depot and stream depot.

client form

The form you use to define a client workspace, such as with the 'p4 client' or 'p4 workspace' commands.

client name

A name that uniquely identifies the current client workspace. Client workspaces, labels, and branch specifications cannot share the same name.

client root

The topmost (root) directory of a client workspace. If two or more client workspaces are located on one machine, they should not share a client root directory.

client side

The right-hand side of a mapping within a client view, specifying where the corresponding depot files are located in the client workspace.

client workspace

Directories on your machine where you work on file revisions that are managed by Helix Server. By default, this name is set to the name of the machine on which your client workspace is located, but it can be overridden. Client workspaces, labels, and branch specifications cannot share the same name.

code review

A process in Helix Swarm by which other developers can see your code, provide feedback, and approve or reject your changes.

codeline

A set of files that evolve collectively. One codeline can be branched from another, allowing each set of files to evolve separately.

comment

Feedback provided in Helix Swarm on a changelist, review, job, or a file within a changelist or review.

commit server

A server that is part of an edge/commit system that processes submitted files (checkins), global workspaces, and promoted shelves.

conflict

1. A situation where two users open the same file for edit. One user submits the file, after which the other user cannot submit unless the file is resolved. 2. A resolve where the same line is changed when merging one file into another. This type of conflict occurs when the comparison of two files to a base yields different results, indicating that the files have been changed in different ways. In this case, the merge cannot be done automatically and must be resolved manually. See file conflict.

copy up

A Helix Server best practice to copy (and not merge) changes from less stable lines to more stable lines. See also merge.

counter

A numeric variable used to track variables such as changelists, checkpoints, and reviews.

CSRF

Cross-Site Request Forgery, a form of web-based attack that exploits the trust that a site has in a user's web browser.

D

default changelist

The changelist used by a file add, edit, or delete, unless a numbered changelist is specified. A default pending changelist is created automatically when a file is opened for edit.

deleted file

In Helix Server, a file with its head revision marked as deleted. Older revisions of the file are still available. in Helix Server, a deleted file is simply another revision of the file.

delta

The differences between two files.

depot

A file repository hosted on the server. A depot is the top-level unit of storage for versioned files (depot files or source files) within a Helix Core Server. It contains all versions of all files ever submitted to the depot. There can be multiple depots on a single installation.

depot root

The topmost (root) directory for a depot.

depot side

The left side of any client view mapping, specifying the location of files in a depot.

depot syntax

Helix Server syntax for specifying the location of files in the depot. Depot syntax begins with: //depot/

diff

(noun) A set of lines that do not match when two files are compared. A conflict is a pair of unequal diffs between each of two files and a base. (verb) To compare the contents of files or file revisions. See also conflict.

donor file

The file from which changes are taken when propagating changes from one file to another.

E

edge server

A replica server that is part of an edge/commit system that is able to process most read/write commands, including 'p4 integrate', and also deliver versioned files (depot files).

exclusionary access

A permission that denies access to the specified files.

exclusionary mapping

A view mapping that excludes specific files or directories.

F

file conflict

In a three-way file merge, a situation in which two revisions of a file differ from each other and from their base file. Also, an attempt to submit a file that is not an edit of the head revision of the file in the depot, which typically occurs when another user opens the file for edit after you have opened the file for edit.

file pattern

Helix Server command line syntax that enables you to specify files using wildcards.

file repository

The master copy of all files, which is shared by all users. In Helix Server, this is called the depot.

file revision

A specific version of a file within the depot. Each revision is assigned a number, in sequence. Any revision can be accessed in the depot by its revision number, preceded by a pound sign (#), for example testfile#3.

file tree

All the subdirectories and files under a given root directory.

file type

An attribute that determines how Helix Server stores and diffs a particular file. Examples of file types are text and binary.

fix

A job that has been closed in a changelist.

form

A screen displayed by certain Helix Server commands. For example, you use the change form to enter comments about a particular changelist to verify the affected files.

forwarding replica

A replica server that can process read-only commands and deliver versioned files (depot files). One or more replicate servers can significantly improve performance by offloading some of the master server load. In many cases, a forwarding replica can become a disaster recovery server.

G

Git Fusion

A Perforce product that integrates Git with Helix, offering enterprise-ready Git repository management, and workflows that allow Git and Helix Server users to collaborate on the same projects using their preferred tools.

graph depot

A depot of type graph that is used to store Git repos in the Helix Server. See also Helix4Git.

Gravatar

gravatar.com is a third party service that you can subscribe to, gravatar enables you to upload an image that you can use in Swarm. When configured, Swarm will attempt to fetch your avatar from gravatar.com and use it within Swarm. If your avatar is not found on gravatar.com, Swarm will use one of its own default avatars to represent your activity. See also the "avatar" entry in this glossary.

group

A feature in Helix Server that makes it easier to manage permissions for multiple users.

H

have list

The list of file revisions currently in the client workspace.

head revision

The most recent revision of a file within the depot. Because file revisions are numbered sequentially, this revision is the highest-numbered revision of that file.

Helix Server

The Helix Server depot and metadata; also, the program that manages the depot and metadata, also called Helix Core Server.

Helix TeamHub

A Perforce management platform for code and artifact repository. TeamHub offers built-in support for Git, SVN, Mercurial, Maven, and more.

Helix4Git

Perforce solution for teams using Git. Helix4Git offers both speed and scalability and supports hybrid environments consisting of Git repositories and 'classic' Helix Server depots.

I

iconv

iconv is a PHP extension that performs character set conversion, and is an interface to the GNU libiconv library.

integrate

To compare two sets of files (for example, two codeline branches) and determine which changes in one set apply to the other, determine if the changes have already been propagated, and propagate any outstanding changes from one set to another.

J

job

A user-defined unit of work tracked by Helix Server. The job template determines what information is tracked. The template can be modified by the Helix Server system administrator. A job describes work to be done, such as a bug fix. Associating a job with a changelist records which changes fixed the bug.

job daemon

A job daemon is a program that checks the Helix Server machine daily to determine if any jobs are open. If so, the daemon sends an email message to interested users, informing them the number of jobs in each category, the severity of each job, and more.

job specification

A form describing the fields and possible values for each job stored in the Helix Server machine.

job view

A syntax used for searching Helix Server jobs.

journal

A file containing a record of every change made to the Helix Server's metadata since the time of the last checkpoint. This file grows as each Helix Server transaction is logged. The file should be automatically truncated and renamed into a numbered journal when a checkpoint is taken.

journal rotation

The process of renaming the current journal to a numbered journal file.

journaling

The process of recording changes made to the Helix Server's metadata.

L

label

A named list of user-specified file revisions.

label view

The view that specifies which filenames in the depot can be stored in a particular label.

lazy copy

A method used by Helix Server to make internal copies of files without duplicating file content in the depot. A lazy copy points to the original versioned file (depot file). Lazy copies minimize the consumption of disk space by storing references to the original file instead of copies of the file.

license file

A file that ensures that the number of Helix Server users on your site does not exceed the number for which you have paid.

list access

A protection level that enables you to run reporting commands but prevents access to the contents of files.

local depot

Any depot located on the currently specified Helix Server.

local syntax

The syntax for specifying a filename that is specific to an operating system.

lock

1. A file lock that prevents other clients from submitting the locked file. Files are unlocked with the 'p4 unlock' command or by submitting the changelist that contains the locked file. 2. A database lock that prevents another process from modifying the database db.* file.

log

Error output from the Helix Server. To specify a log file, set the P4LOG environment variable or use the p4d -L flag when starting the service.

M

mapping

A single line in a view, consisting of a left side and a right side that specify the correspondences between files in the depot and files in a client, label, or branch. See also workspace view, branch view, and label view.

MDS checksum

The method used by Helix Server to verify the integrity of versioned files (depot files).

merge

1. To create new files from existing files, preserving their ancestry (branching). 2. To propagate changes from one set of files to another. 3. The process of combining the contents of two conflicting file revisions into a single file, typically using a merge tool like P4Merge.

merge file

A file generated by the Helix Server from two conflicting file revisions.

metadata

The data stored by the Helix Server that describes the files in the depot, the current state of client workspaces, protections, users, labels, and branches. Metadata includes all the data stored in the Perforce service except for the actual contents of the files.

modification time or modtime

The time a file was last changed.

MPM

Multi-Processing Module, a component of the Apache web server that is responsible for binding to network ports, accepting requests, and dispatch operations to handle the request.

N

nonexistent revision

A completely empty revision of any file. Syncing to a nonexistent revision of a file removes it from your workspace. An empty file revision created by deleting a file and the #none revision specifier are examples of nonexistent file revisions.

numbered changelist

A pending changelist to which Helix Server has assigned a number.

O

opened file

A file that you are changing in your client workspace that is checked out. If the file is not checked out, opening it in the file system does not mean anything to the versioning engineer.

owner

The Helix Server user who created a particular client, branch, or label.

P

p4

1. The Helix Core Server command line program. 2. The command you issue to execute commands from the operating system command line.

p4d

The program that runs the Helix Server; p4d manages depot files and metadata.

P4PHP

The PHP interface to the Helix API, which enables you to write PHP code that interacts with a Helix Server machine.

PECL

PHP Extension Community Library, a library of extensions that can be added to PHP to improve and extend its functionality.

pending changelist

A changelist that has not been submitted.

project

In Helix Swarm, a group of Helix Server users who are working together on a specific codebase, defined by one or more branches of code, along with options for a job filter, automated test integration, and automated deployment.

protections

The permissions stored in the Helix Server's protections table.

proxy server

A Helix Server that stores versioned files. A proxy server does not perform any commands. It serves versioned files to Helix Server clients.

R

RCS format

Revision Control System format. Used for storing revisions of text files in versioned files (depot files). RCS format uses reverse delta encoding for file storage. Helix Server uses RCS format to store text files. See also reverse delta storage.

read access

A protection level that enables you to read the contents of files managed by Helix Server but not make any changes.

remote depot

A depot located on another Helix Server accessed by the current Helix Server.

replica

A Helix Server that contains a full or partial copy of metadata from a master Helix Server. Replica servers are typically updated every second to stay synchronized with the master server.

repo

A graph depot contains one or more repos, and each repo contains files from Git users.

reresolve

The process of resolving a file after the file is resolved and before it is submitted.

resolve

The process you use to manage the differences between two revisions of a file. You can choose to resolve conflicts by selecting the source or target file to be submitted, by merging the contents of conflicting files, or by making additional changes.

reverse delta storage

The method that Helix Server uses to store revisions of text files. Helix Server stores the changes between each revision and its previous revision, plus the full text of the head revision.

revert

To discard the changes you have made to a file in the client workspace before a submit.

review access

A special protections level that includes read and list accesses and grants permission to run the p4 review command.

review daemon

A review daemon is a program that periodically checks the Helix Server machine to determine if any changelists have been submitted. If so, the daemon sends an email message to users who have subscribed to any of the files included in those changelists, informing them of changes in files they are interested in.

revision number

A number indicating which revision of the file is being referred to, typically designated with a pound sign (#).

revision range

A range of revision numbers for a specified file, specified as the low and high end of the range. For example, myfile#5,7 specifies revisions 5 through 7 of myfile.

revision specification

A suffix to a filename that specifies a particular revision of that file. Revision specifiers can be revision numbers, a revision range, change numbers, label names, date/time specifications, or client names.

RPM

RPM Package Manager is a tool, and package format, for managing the installation, updates, and removal of software packages for Linux distributions such as Red Hat Enterprise Linux, the Fedora Project, and the CentOS Project.

S

server data

The combination of server metadata (the Helix Server database) and the depot files (your organization's versioned source code and binary assets).

server root

The topmost directory in which p4d stores its metadata (db.* files) and all versioned files (depot files or source files). To specify the server root, set the P4ROOT environment variable or use the p4d -r flag.

service

In the Helix Core Server, the shared versioning service that responds to requests from Helix Server client applications. The Helix Server (p4d) maintains depot files and metadata describing the files and also tracks the state of client workspaces.

shelve

The process of temporarily storing files in the Helix Server without checking in a changelist.

status

For a changelist, a value that indicates whether the changelist is new, pending, or submitted. For a job, a value that indicates whether the job is open, closed, or suspended. You can customize job

statuses. For the 'p4 status' command, by default the files opened and the files that need to be reconciled.

stream

A branch with additional intelligence that determines what changes should be propagated and in what order they should be propagated.

stream depot

A depot used with streams and stream clients.

submit

To send a pending changelist into the Helix Server depot for processing.

super access

An access level that gives the user permission to run every Helix Server command, including commands that set protections, install triggers, or shut down the service for maintenance.

symlink file type

A Helix Server file type assigned to symbolic links. On platforms that do not support symbolic links, symlink files appear as small text files.

sync

To copy a file revision (or set of file revisions) from the Helix Server depot to a client workspace.

T

target file

The file that receives the changes from the donor file when you integrate changes between two codelines.

text file type

Helix Server file type assigned to a file that contains only ASCII text, including Unicode text. See also binary file type.

theirs

The revision in the depot with which the client file (your file) is merged when you resolve a file conflict. When you are working with branched files, theirs is the donor file.

three-way merge

The process of combining three file revisions. During a three-way merge, you can identify where conflicting changes have occurred and specify how you want to resolve the conflicts.

trigger

A script automatically invoked by Helix Server when various conditions are met. (See Helix Core Server Administrator Guide: Fundamentals on "Using triggers to customize behavior".)

two-way merge

The process of combining two file revisions. In a two-way merge, you can see differences between the files.

typemap

A table in Helix Server in which you assign file types to files.

U

user

The identifier that Helix Server uses to determine who is performing an operation.

V

versioned file

Source files stored in the Helix Server depot, including one or more revisions. Also known as a depot file or source file. Versioned files typically use the naming convention 'filenamev' or '1.changelist.gz'.

view

A description of the relationship between two sets of files. See workspace view, label view, branch view.

W

wildcard

A special character used to match other characters in strings. The following wildcards are available in Helix Server: * matches anything except a slash; ... matches anything including slashes; %%0 through %%9 is used for parameter substitution in views.

workspace

See client workspace.

workspace view

A set of mappings that specifies the correspondence between file locations in the depot and the client workspace.

write access

A protection level that enables you to run commands that alter the contents of files in the depot. Write access includes read and list accesses.

X

XSS

Cross-Site Scripting, a form of web-based attack that injects malicious code into a user's web browser.

Y

yours

The edited version of a file in your client workspace when you resolve a file. Also, the target file when you integrate a branched file.

License Statements

Perforce Software includes software developed by the University of California, Berkeley and its contributors. This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/).

Perforce Software includes software from the Apache ZooKeeper project, developed by the Apache Software Foundation and its contributors. (http://zookeeper.apache.org/)

Perforce Software includes software developed by the OpenLDAP Foundation (http://www.openldap.org/).

Perforce Software includes software developed Computing Services at Carnegie Mellon University: Cyrus SASL (http://www.cmu.edu/computing/).