

---

# PySiLK: SiLK in Python

*Release 1.0.0*

CERT NetSA

March 27, 2008

[silk-help@cert.org](mailto:silk-help@cert.org)

## Abstract

This document describes how to read and write SiLK packed data from within Python.

## Contents

<b>1</b>	<b><a href="#">silk — SiLK record and file support</a></b>	<b>1</b>
1.1	<a href="#">Available Types</a>	2
1.2	<a href="#">PySiLK Example</a>	2
1.3	<a href="#">IPAddr Objects</a>	4
1.4	<a href="#">IPWildcard Objects</a>	4
1.5	<a href="#">IPSet Objects</a>	5
1.6	<a href="#">TCPFlags Objects</a>	6
1.7	<a href="#">RWRec Objects</a>	7
1.8	<a href="#">SilkFile Object</a>	8
1.9	<a href="#">FGlob Objects</a>	9

---

## 1 [silk](#) — SiLK record and file support

The `silk` module supplies objects for interfacing with SiLK records and data files.

The `silk` module exports the following (read-only) constants:

### **sensors**

A tuple of valid sensor names.

### **classes**

A tuple of valid class names.

### **classtypes**

A tuple of valid (class name, type name) tuples.

The `silk` module exports the following functions:

### **ipv6\_enabled()**

Return `True` if SiLK was compiled with IPv6 support, `False` otherwise.

### **initial\_tcpflags\_enabled()**

Return `True` if SiLK was compiled with support for initial TCP flags, `False` otherwise.

**have\_site\_config()**

Returns `True` if the module was able to locate the SiLK configuration file, `False` otherwise.

The value of `$SILK_CONFIG_FILE` should include the name of the configuration file. Otherwise, the module looks for a file named `'silk.conf'` in the following directories: the directory specified in the `$SILK_DATA_ROOTDIR` environment variable; the data root directory that is compiled into SiLK; the directories `'$SILK_PATH/share/silk/'`, and `'$SILK_PATH/share/'`.

## 1.1 Available Types

**class IPAddr**

A representation for IP Addresses.

**class IPWildcard**

A representation of IP wildcard addresses or CIDR blocks.

**class IPSet**

A representation of an IPset.

**class TCPFlags**

A representation of TCP flags.

**class RWRec**

A representation of a SiLK data record.

**class FGlob**

An iterable object that allows retrieval of filenames in a SiLK data store.

## 1.2 PySiLK Example

The following is an example using the PySiLK bindings. The code is meant to show some standard PySiLK techniques, but is not otherwise meant to be useful. Explanations for the code can be found inline in the comments.

```

#!/usr/bin/python2.4

# Import the pysilk bindings
from silk import *

# Import sys for the command line arguments.
import sys

# Main function
def main():

    if len(sys.argv) != 3:
        print ("Usage: %s infile outset" % sys.argv[0])

    # Open an silk file for reading
    infile = SilkFile(sys.argv[1], READ)

    # Create an empty IPset
    destset = IPSet()

    # Loop over the records in the file
    for rec in infile:

        # Do comparisons based on rwrec field value
        if (rec.protocol == 6 and rec.sport in [80, 8080] and
            rec.packets > 3 and rec.bytes > 120):

            # Add the dest IP of the record to the IPset
            destset.add(rec.dip)

    # Save the IPset for future use
    destset.save(sys.argv[2])

    # count the items in the set
    count = 0
    for addr in destset:
        count = count + 1

    print "%d addresses" % count

    # Another way to do the same
    print "%d addresses" % len(destset)

    # Print the ip blocks in the set
    for base_prefix in destset.cidr_iter():
        print "%s/%d" % base_prefix

# Call the main() function when this program is started
if __name__ == '__main__':
    main()

```

## 1.3 IPAddr Objects

An `IPAddr` object represents an IPv4 or IPv6 address.

**class `IPAddr`** (*address*)

The constructor takes either a string *address*, which must be a string representation of either an IPv4 or IPv6 address, or an integer representation of the address. IPv6 addresses are only accepted if `ipv6_enabled()` returns `True`.

Examples:

```
>>> addr1 = IPAddr('192.160.1.1')
>>> addr2 = IPAddr('2001:db8::1428:57ab')
>>> addr3 = IPAddr('::ffff:12.34.56.78')
>>> addr4 = IPAddr(0xffffffff)
>>> addr5 = IPAddr(0xffffffffffffffffffffffffffffffff)
```

Supported operations:

Operation	Result
<code>addr1 &lt; addr2</code>	<code>addr1</code> is considered less than <code>addr2</code> if the 128-bit representation of <code>addr1</code> is less than <code>addr2</code>
<code>int(addr1)</code>	The integer representation of <code>addr</code>

Instance methods:

**`ipv6()`**

Return `True` if the address is an IPv6 address, `False` otherwise.

**`__str__()`**

For an address *addr*, `str(addr)` returns a human-readable representation of that address.

## 1.4 IPWildcard Objects

An `IPWildcard` object represents a range or block of IP addresses. The `IPWildcard` object handles iteration over IP addresses with `for x in wildcard`.

**class `IPWildcard`** (*wildcard*)

The constructor takes a string representation *wildcard* of the wildcard address.

The string *wildcard* can be in CIDR notation, an integer, an integer with a CIDR designation, or an entry in SiLK wildcard notation. In SiLK wildcard notation, a wildcard is represented as a string IP address in canonical form with an `x` representing an entire octet or hexadectet. An IP wildcard string can also have lists or ranges in place of an octet or hexadectet. IPv6 wildcard addresses are only accepted if `ipv6_enabled()` returns `True`.

Examples:

```
>>> a = IPWildcard('1.2.3.0/24')
>>> b = IPWildcard('ff80::/16')
>>> c = IPWildcard('1.2.3.4')
>>> d = IPWildcard('::FFFF:0102:0304')
>>> e = IPWildcard('16909056')
>>> f = IPWildcard('16909056/24')
>>> g = IPWildcard('1.2.3.x')
>>> h = IPWildcard('1:2:3:4:5:6:7.x')
>>> i = IPWildcard('1.2,3.4,5.6,7')
>>> j = IPWildcard('1.2.3.0-255')
>>> k = IPWildcard('::2-4')
>>> l = IPWildcard('1-2:3-4:5-6:7-8:9-a:b-c:d-e:0-ffff')
```

Supported operations:

Operation	Result
<i>addr</i> in <i>wildcard</i>	True if <i>addr</i> is in <i>wildcard</i> , False otherwise
<i>addr</i> not in <i>wildcard</i>	False if <i>addr</i> is in <i>wildcard</i> , True otherwise
<i>string</i> in <i>wildcard</i>	Same as: <code>IPAddr(<i>string</i>) in <i>wildcard</i></code>
<i>string</i> not in <i>wildcard</i>	Same as: <code>IPAddr(<i>string</i>) not in <i>wildcard</i></code>

Instance methods:

`__str__()`

For an IP wildcard *wild*, `str(wild)` returns the string that was used to make the wildcard.

## 1.5 IPSet Objects

An `IPSet` object represents any set of IP addresses, as produced by **rwsetbuild** and related programs. The `IPSet` object handles iteration over IP addresses with `for x in set`, and iteration over CIDR blocks using `for x in set.cidr_iter()`.

**class** `IPSet` ([*iterable*])

The constructor creates an empty `IPSet`. If an iterable is supplied as an argument, each item of the iterable will be added to the `IPSet`. Each item of the iterable should either be an `IPv4 IPAddr` or a string representing a valid IPv4 address.

Other constructors, all class methods:

**load** (*path*)

Creates an `IPSet` from an `IPSet` saved in a file. *path* must be a valid location of an `IPSet`.

Supported operations:

Operation	Equivalent	Result	Notes
<code>len(s)</code> <code>s.cardinality()</code>		cardinality of <code>IPSet s</code> cardinality of <code>IPSet s</code>	(1)
<i>addr</i> in <i>s</i> <i>addr</i> not in <i>s</i>		test <i>addr</i> for membership in <i>s</i> test <i>addr</i> for non-membership in <i>s</i>	(2) (2)
<code>s.issubset(t)</code> <code>s.issuperset(t)</code>	$s \leq t$ $s \geq t$	test whether every element in <i>s</i> is in <i>t</i> test whether every element in <i>t</i> is in <i>s</i>	(3) (3)
<code>s.union(t)</code> <code>s.intersection(t)</code> <code>s.difference(t)</code> <code>s.symmetric_difference(t)</code> <code>s.copy()</code>	$s \mid t$ $s \& t$ $s - t$ $s \wedge t$	new <code>IPSet</code> with elements from both <i>s</i> and <i>t</i> new <code>IPSet</code> with elements common to <i>s</i> and <i>t</i> new <code>IPSet</code> with elements in <i>s</i> but not in <i>t</i> new <code>IPSet</code> with elements in either <i>s</i> or <i>t</i> but not both new set with a copy of <i>s</i>	(3) (3) (3) (3) (3)
<code>s.update(t)</code> <code>s.intersection_update(t)</code> <code>s.difference_update(t)</code> <code>s.symmetric_difference_update(t)</code>	$s \mid= t$ $s \&= t$ $s -= t$ $s \wedge= t$	update <i>s</i> , adding elements from <i>t</i> update <i>s</i> , keeping only elements found in both <i>s</i> and <i>t</i> update <i>s</i> , removing elements found in <i>t</i> update <i>s</i> , keeping elements found in <i>s</i> or <i>t</i> but not in both	(3) (3) (3) (3)
<code>s.add(addr)</code> <code>s.remove(addr)</code> <code>s.discard(addr)</code> <code>s.clear()</code>		add element <i>addr</i> to <code>IPSet s</code> remove <i>addr</i> from <code>IPSet s</code> ; raises <code>KeyError</code> if not present removes <i>addr</i> from <code>IPSet s</code> if present remove all elements from <code>IPSet s</code>	(2)

Notes:

(1) May throw `OverflowError` if there are too many IP addresses in the `IPSet`. Use `s.cardinality()` instead.

- (2) *addr* can be an `IPAddr`, an `IPWildcard`, or the string representation of either. The address or addresses must be an IPv4 addresses.
- (3) With the non-operator version of this method, *t* can be any iterable object of IP addresses or IP address strings. The operator version requires that *t* be an `IPSet`.

Instance methods:

**`cidr_iter()`**

Returns an iterator over CIDR blocks. Each iteration returns a tuple, the first element of which is the first IP address in the block, the second of which is the prefix length of the block. Can be used as `for (addr, prefix) in s.cidr_iter():`.

**`save(filename)`**

Saves the `IPSet` in the file *filename*.

## 1.6 TCPFlags Objects

A `TCPFlags` object represents the eight bits of flags from a TCP session.

**`class TCPFlags(value)`**

The constructor takes either a `TCPFlags` value, a string, or an integer. If a `TCPFlags` value, it returns a copy of that value. If an integer, the integer should represent the 8-bit representation of the flags. If a string, the string should consist of a concatenation of zero or more of the characters 'F', 'S', 'R', 'P', 'A', 'U', 'E', and 'C'—upper or lower-case—representing the FIN, SYN, RST, PSH, ACK, URG, ECE, and CWR flags. Spaces in the string are ignored.

Examples:

```
>>> a = TCPFlags('SA')
>>> b = TCPFlags(5)
```

Instance attributes (read-only):

Attribute	Value
FIN	True if the FIN flag is set, False otherwise
SYN	True if the SYN flag is set, False otherwise
RST	True if the RST flag is set, False otherwise
PSH	True if the PSH flag is set, False otherwise
ACK	True if the ACK flag is set, False otherwise
URG	True if the URG flag is set, False otherwise
ECE	True if the ECE flag is set, False otherwise
CWR	True if the CWR flag is set, False otherwise

Supported operations:

Operation	Result
$\sim f$	The bitwise inversion (not) of <i>f</i>
$f1 \ \& \ f2$	The bitwise intersection (and) of the flags from <i>f1</i> and <i>f2</i>
$f1 \   \ f2$	The bitwise union (or) of the flags from <i>f1</i> and <i>f2</i>
$f1 \ ^ \ f2$	The bitwise exclusive disjunction (xor) of the flags from <i>f1</i> and <i>f2</i>
<code>int(f)</code>	The integer value of the flags <i>f</i>
<i>f</i>	Can be used as a truth value with any flag set == True, False otherwise

Constants:

The following constants are defined:

Constant	Meaning
FIN	A TCPFlags value with only the FIN flags set
SYN	A TCPFlags value with only the SYN flags set
RST	A TCPFlags value with only the RST flags set
PSH	A TCPFlags value with only the PSH flags set
ACK	A TCPFlags value with only the ACK flags set
URG	A TCPFlags value with only the URG flags set
ECE	A TCPFlags value with only the ECE flags set
CWR	A TCPFlags value with only the CWR flags set

Supported methods:

**matches** (*flagmask*)

Given a *flagmask* of the form "*flags/mask*", returns `True` if if the flags of self match *flags* after being masked with *mask*, `False` otherwise.

Given a *flagmask* without the '/', checks for literal equality, as if the mask contained all flags.

**\_\_str\_\_** ()

For an TCPFlags object *f*, `str (f)` returns the a string representation of the flags set in *f*.

## 1.7 RWRec Objects

An `RWRec` object represents a SiLK record.

**class RWRec** ([*rec*],[*field=value*],...)

This constructor creates an empty `RWRec` object. If an `RWRec` *rec* is supplied, it will create a copy of *rec*. The variable *rec* can be a dictionary, such as that supplied by `RWRec.as_dict()`. Initial values for record fields can be included.

Example:

```
>>> recA = RWRec(input=10, output=20)
>>> recB = RWRec(recA, output=30)
>>> (recA.input, recA.output)
(10, 20)
>>> (recB.input, recB.output)
(10, 30)
```

Instance attributes:

Attribute	Value	Type
<code>application</code>	The “service” port set by the collector	integer
<code>bytes</code>	The count of the number of bytes in the flow	integer
<code>classname</code>	The class name of the record (read-only)	string
<code>classtype</code>	A tuple of the class name and type name of the record	(string, string)
<code>dip</code>	The destination IP (can be set as a string)	IPAddr
<code>dport</code>	The destination port	integer
<code>duration</code>	The duration of the flow	<code>datetime.timedelta</code>
<code>etime</code>	The end time of the flow	<code>datetime.datetime</code>
<code>initflags</code>	The TCP flags of the first packet of the flow (may be None)	TCPFlags
<code>icmpcode</code>	The ICMP code (only valid if <code>protocol</code> is 1)	integer
<code>icmptype</code>	The ICMP type value (only valid if <code>protocol</code> is 1)	integer
<code>input</code>	The router’s incoming SNMP interface	integer
<code>nhip</code>	The router’s next-hop IP (can be set as a string)	IPAddr
<code>output</code>	The router’s outgoing SNMP interface	integer
<code>packets</code>	The packet count for the flow	integer
<code>protocol</code>	The IP protocol	integer
<code>restflags</code>	The union of the flags of all but the first packet of the flow (may be None)	TCPFlags
<code>sensor</code>	The sensor ID	string
<code>sip</code>	The source IP (can be set as a string)	IPAddr
<code>sport</code>	The source port	integer
<code>stime</code>	The start time of the flow	<code>datetime.datetime</code>
<code>tcpflags</code>	The union of the TCP flags of all packets in the flow	TCPFlags
<code>timeout_killed</code>	Whether the flow ended early due to timeout by the collector (may be None)	boolean
<code>timeout_started</code>	Whether the flow is a continuation from a timed-out flow (may be None)	boolean
<code>typename</code>	The type name of the record (read-only)	string

Supported methods:

**`is_web()`**  
 True if the record can be represented as a web record, False otherwise.

**`as_dict()`**  
 Returns a dictionary representing the contents of the record.

**`__str__()`**  
 For an record *rec*, `str(rec)` returns the string representation of `rec.as_dict()`.

Supported operations:

Operation	Result
<code>rec1 == rec2</code>	True if <i>rec1</i> is structurally equivalent to <i>rec2</i>
<code>rec1 != rec2</code>	True if <i>rec1</i> is not structurally equivalent to <i>rec2</i>

## 1.8 SilkFile Object

An `SilkFile` object represents a channel for writing to or reading from SiLK flow files. A SiLK file open for reading can be iterated over using `for rec in file`.

**class `SilkFile`** (*filename*, *mode*, *compression*=`DEFAULT`, *notes*=[], *invocations*=[])  
 The constructor takes a filename, a mode, and a set of optional keyword parameters. The *filename* should be the path to the file to open. The *mode* should be one of the following constant values:



Mode	Meaning
READ	Open file for reading
WRITE	Open file for writing
APPEND	Open file for appending

The *compression* parameter can be one of the following constants:

Constant	Meaning
DEFAULT	Default compression scheme compiled into SiLK
NO_COMPRESSION	No compression
ZLIB	Use zlib block compression
LZO1X	Use lzo1x block compression

If *notes* or *invocations* are set, they should be list of strings. These add annotation and invocation headers to the file.

Examples:

```
>>> myinputfile = SilkFile('/path/to/file', READ)
>>> myoutputfile = SilkFile('/path/to/file', WRITE, compression=LZO1X,
                           notes=['My output file',
                                'another annotation'])
```

Instance methods:

**read()**

Returns an `RWRec` representing the next record in the `SilkFile`. If there are no records left in the file, returns `None`.

**write(rec)**

Writes the `RWRec` *rec* to the file. Returns `None`.

**next()**

A `SilkFile` object is its own iterator, for example `iter(f)` returns *f*. When the `SilkFile` is used as an iterator, the `next()` method is called repeatedly. This method returns the next record, or raises `StopIteration` when EOF is hit.

**notes()**

Returns the list of annotation headers for the file as a list of strings.

**invocations()**

Returns the list of invocation headers for the file as a list of strings.

**close()**

Closes the file. Returns `None`.

## 1.9 FGlob Objects

An `FGlob` object is an iterable object which iterates over filenames from a SiLK data store. It does this internally by calling the `rwfglob` program. The `FGlob` object assumes that the `rwfglob` program is in the `PATH`, and will throw an exception when used if not.

```
class FGlob (classname=None, type=None, sensors=None, start_date=None, end_date=None,
             data_rootdir=None, site_config_file=None)
```

Arguments are:

*classname*, if given, should be a string representing the class name. If not given, defaults based on the site configuration file.

*type*, if given, can be either a string representing a type name or comma-separated list of type names, or can be a list of strings representing type names. If not given, defaults based on the site configuration file.

*sensors*, if given, should be either a string representing a comma-separated list of sensor names or IDs, and integer representing a sensor ID, or a list of strings or integers representing sensor names or IDs. If not given, defaults to all sensors.

*start\_date*, if given, should be either a string in the format YYYY/MM/DD[:HH], a date object, a datetime object (which will be used to the precision of one hour), or a time object (which is used for the given hour on the current date). If not given, defaults to start of current day.

*end\_date*, if given, should be either a string in the format YYYY/MM/DD[:HH], a date object, a datetime object (which will be used to the precision of one hour), or a time object (which is used for the given hour on the current date). If not given, defaults to *start\_date*. *end\_date* cannot be used without a *start\_date*.

*data\_rootdir*, if given, should be a string representing the directory in which to find the packed SiLK data files. If not given, defaults to \$SILK\_DATA\_ROOTDIR or the compiled-in default.

*site\_config\_file*, if given, should be a string representing the path of the site configuration file. If not given, defaults to \$SILK\_CONFIG\_FILE or '\$SILK\_DATA\_ROOTDIR/silk.conf'.

At least one of *classname*, *type*, *sensors*, *start\_date* must be specified.

An `FGlob` object can be used as a standard iterator. For example:

```
for filename in FGlob(classname="all", start_date="2005/09/22"):
    for rec in SilkFile(filename):
        ...
```