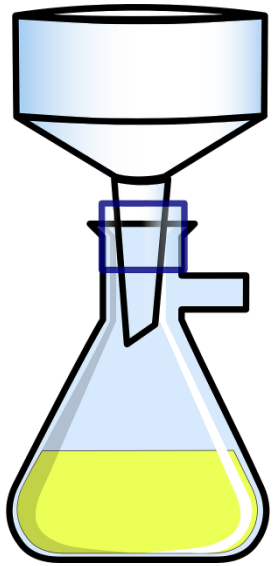
Interacting with the Linux System

Dr. Chris Brown

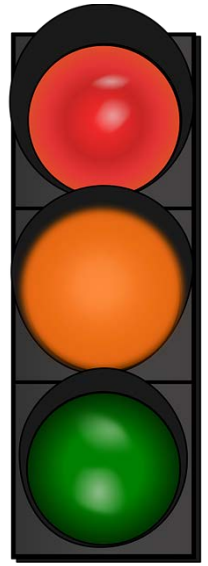
Interacting with the Linux System

Slicing

The command line and the environment



Files, Streams and Filters



Signals

Slicing

Slicing

Slicing extracts part of a sequence

- Creates a new sequence
- Does not modify the original

Any sequence
(string, list, tuple)

```
x[4:15:2]
```

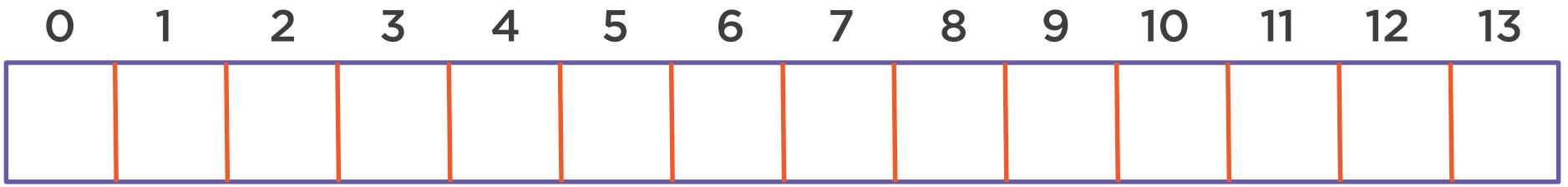
Start position
(if omitted, start
at beginning)

End position (non-inclusive)
(if omitted, continue to end)

Optional
Increment

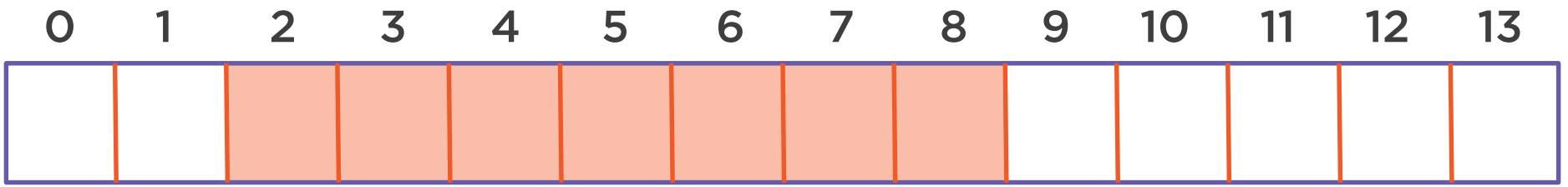
Negative values mean:
"count backwards from end"

Slicing

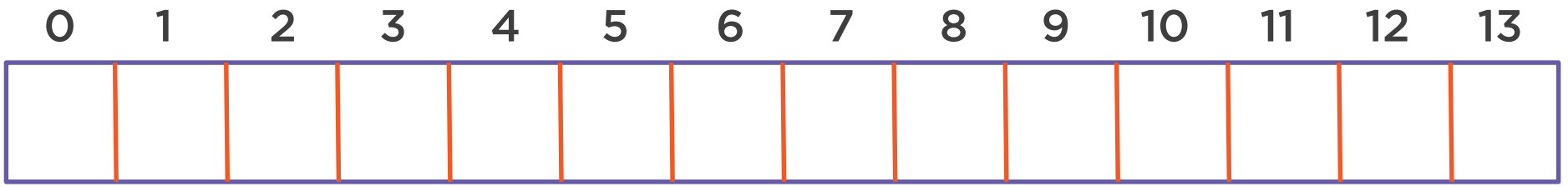


`data`

`data[2:9]`

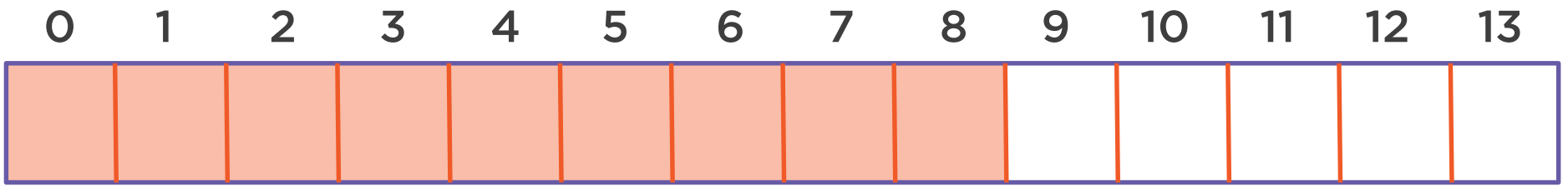


Slicing

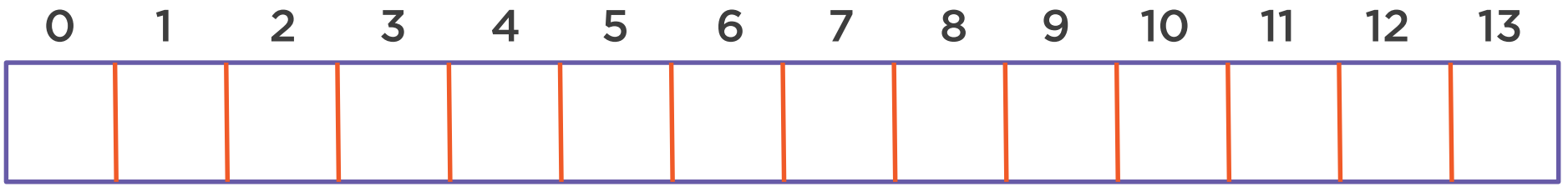


`data`

`data[:9]`

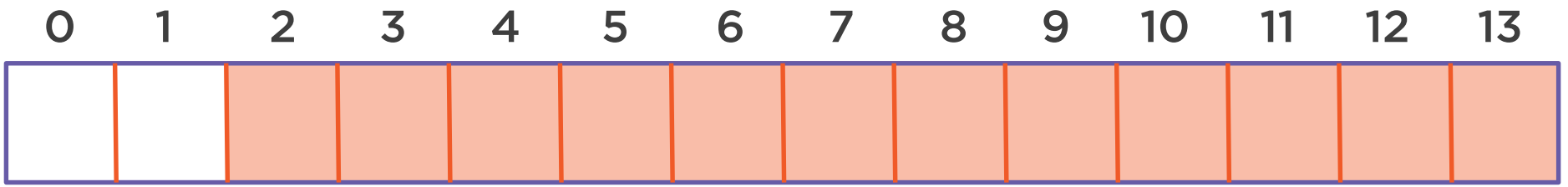


Slicing

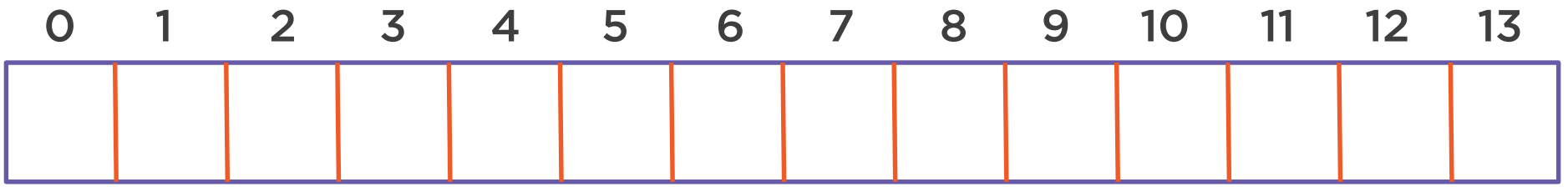


`data`

`data[2:]`

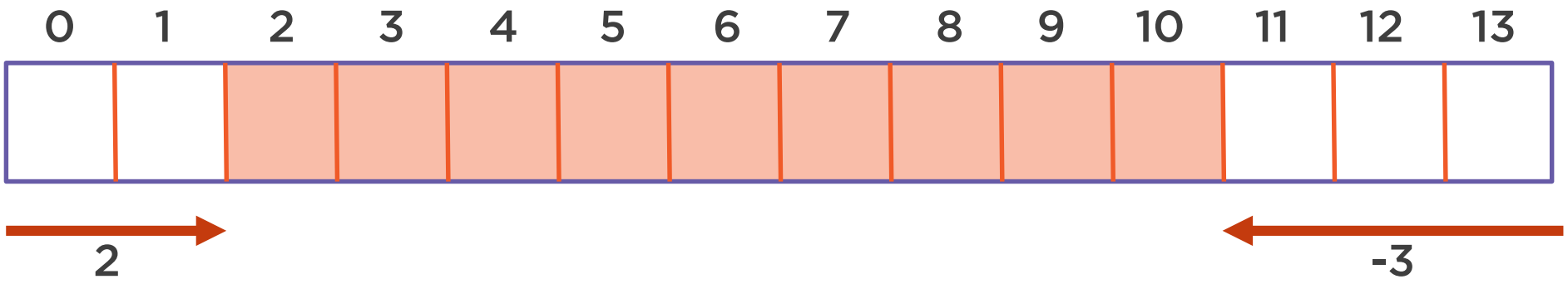


Slicing

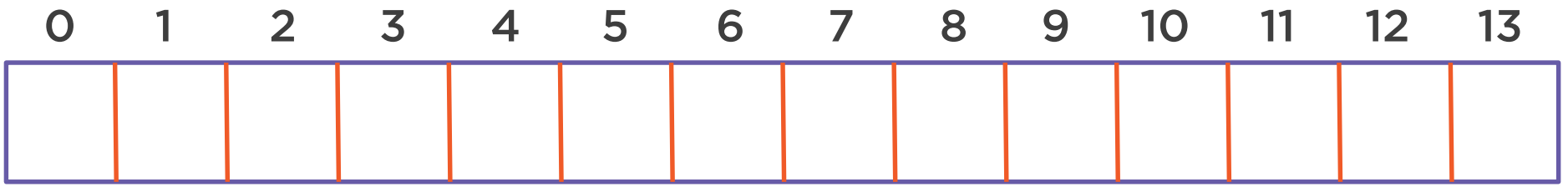


`data`

`data[2:-3]`

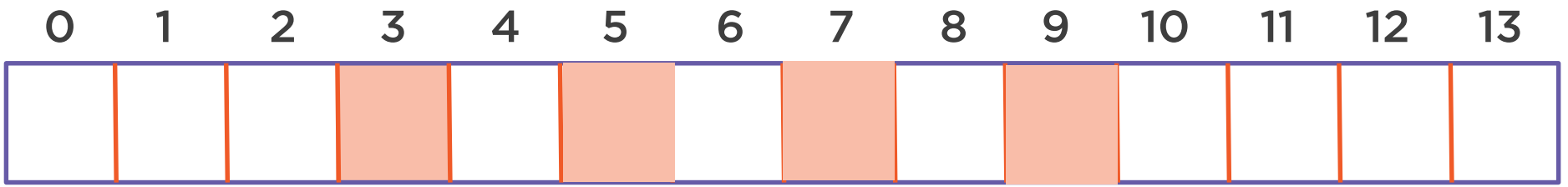


Slicing



`data`

`data[3:10:2]`



Slicing – Useful Special Cases

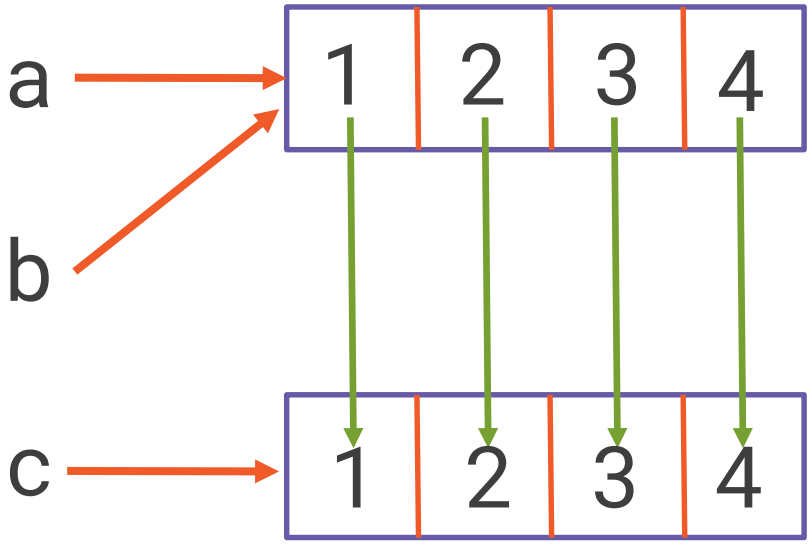
`data[1:]`	Remove the first element
`data[:-1]`	Remove the last element
`data[-1:]`	Get the last element
`data[::-1]`	Reverse the list
`data[:]`	Make a "deep" copy

All slicing operations create a new sequence.

They do *not* modify the original

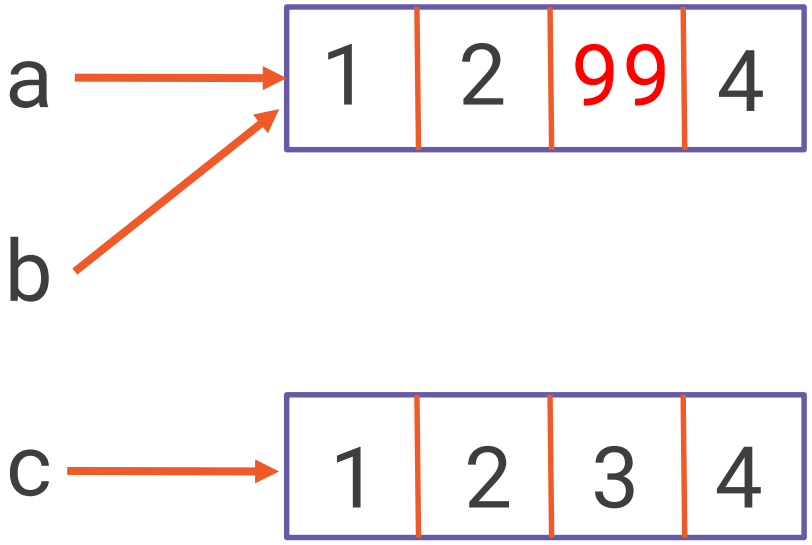
Shallow vs Deep Copies

```
a = [1, 2, 3, 4]

b = a

c = a[:]

a[2] = 99
```



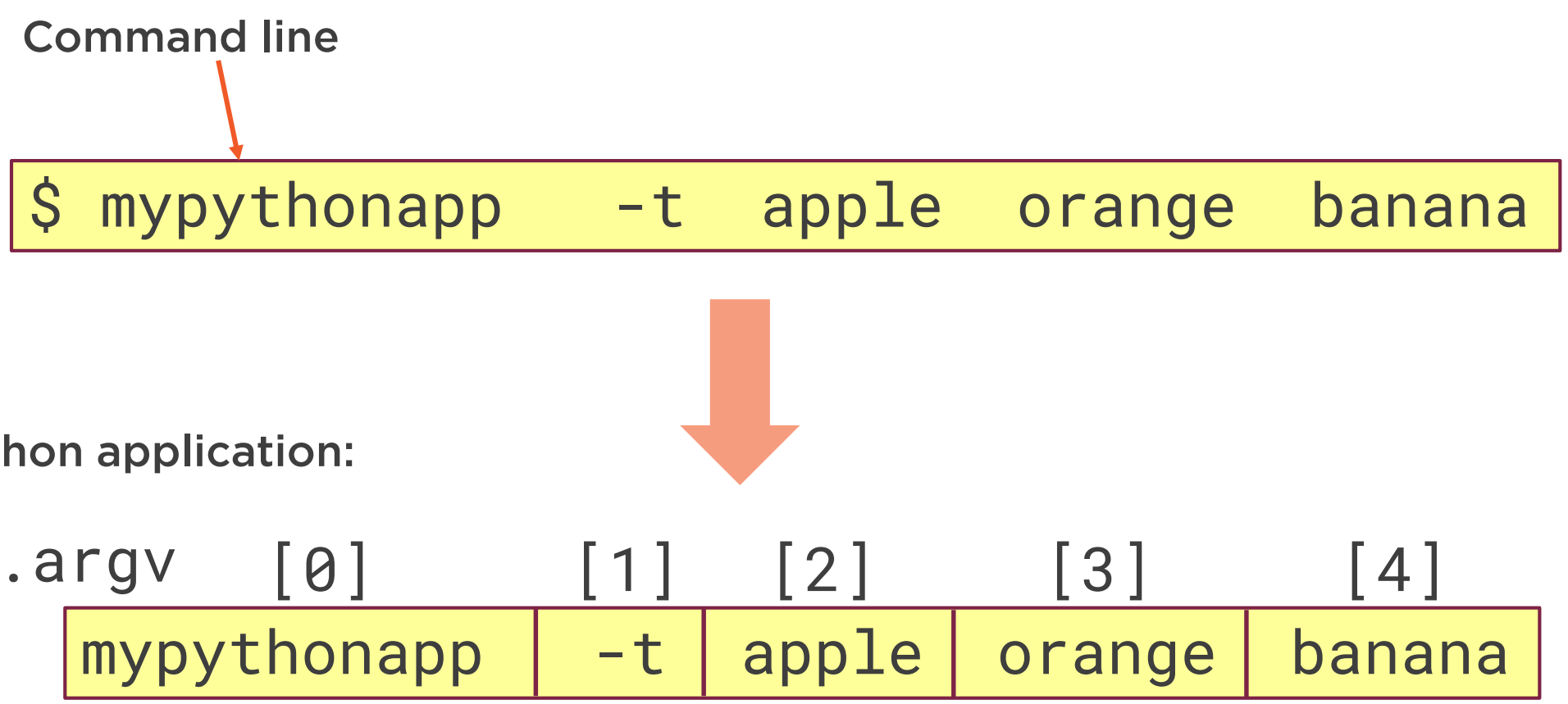
Shallow vs Deep Copies

```
a = [1, 2, 3, 4]

b = a

c = a[:]

a[2] = 99
```



The Command Line and the Environment

Accessing Command Line Arguments



Echoing Command Line Arguments

```
#!/usr/bin/python3
import sys

for arg in sys.argv[1:]:
    print(arg, end=' ')
print()
```

Do not echo the program name

Terminate with a space not a newline

A More Complex Example

In a later lesson we will write a program that monitors a specified list of disk partitions and generates a report for those that are fuller than a specified threshold.

We want it to accept command line arguments and options like this:

```
$ check-partitions --help
Usage: optparse-demo.py [options] partition ...

Options:
  -h, --help            show this help message and exit
  -t THRESHOLD, --threshold=THRESHOLD
                        Set threshold (%)
  -s, --single          just check once, don't loop
  -m MAILBOX, --mailbox=MAILBOX
                        mail report to this mailbox
```

Using the `optparse` Module

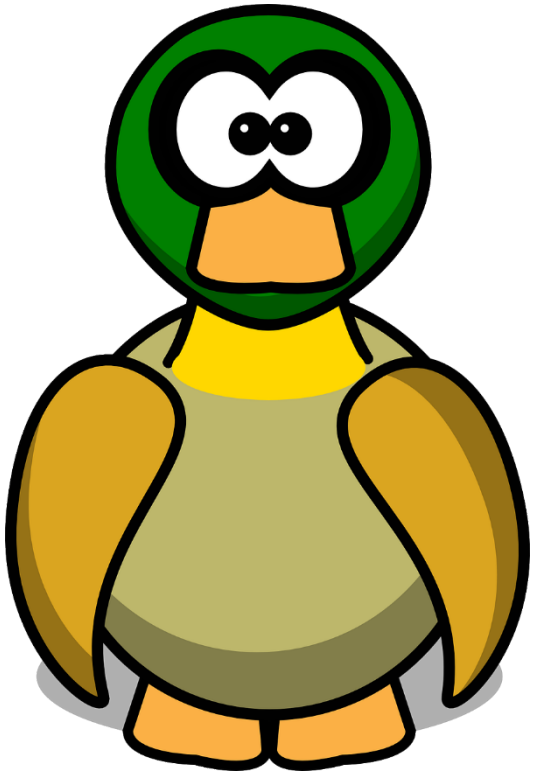
Import just one name
into the current namespace

```
from optparse import OptionParser

parser = OptionParser()
parser.add_option("-t", "--threshold",
                  dest="threshold",
                  type="int",
                  default=90,
                  help="Set threshold (%)")
# Add more options ...

(options, args) = parser.parse_args()
```

Create the parser

Python lets you
break lines within a
comma-separated
argument list

Files, Streams and Filters



A number of classes in Python offer "file-like" behaviour

- Can be read and written like a file

An example of "duck typing"

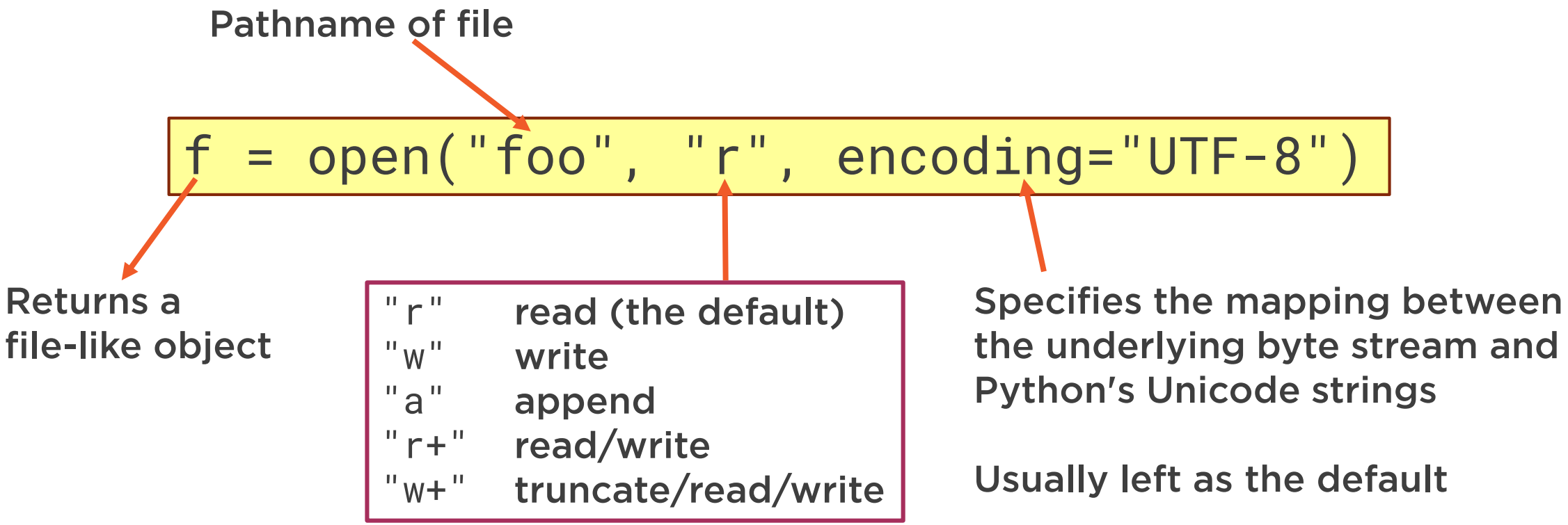
- If it walks like a duck, swims like a duck, and quacks like a duck, then it's a duck

File-Like Objects

File-like objects support the following methods:

Method	Description
`read(n)`	Read up to n bytes (default: whole file)
`readline(n)`	Read up to and including the next newline
`readlines(n)`	Read and return a list of lines (up to n bytes)
`write(s)`	Write a string to the file
`writelines(lines)`	Write a list of lines (does not add newlines)
`flush()`	Flushes the output buffers
`close()`	Flush and close the stream
`truncate(size)`	Truncate to size bytes

Opening a Text File



stdin, stdout and stderr

The sys module provides three "file-like objects" corresponding to a program's three standard streams:

Stream Manipulation Example

```
import sys

print("this is written to stdout")

print("this is written to stderr", file = sys.stderr)

f = open("out1", "w")
print("this is written to out1", file = f)
f.close()

with open("out2", "w") as f:
    print("this is written to out2", file = f)

old_stdout = sys.stdout
with open("out3", "w") as f:
    sys.stdout = f
    print("this is written to out3")
sys.stdout = old_stdout
print("stdout is restored")
```

Standard Filter Behaviour

A filter reads a single input stream, transforms it, and writes a single output stream

With no file names on its command line, a filter reads `stdin`:

With one or more file names, the filter reads those files in turn:

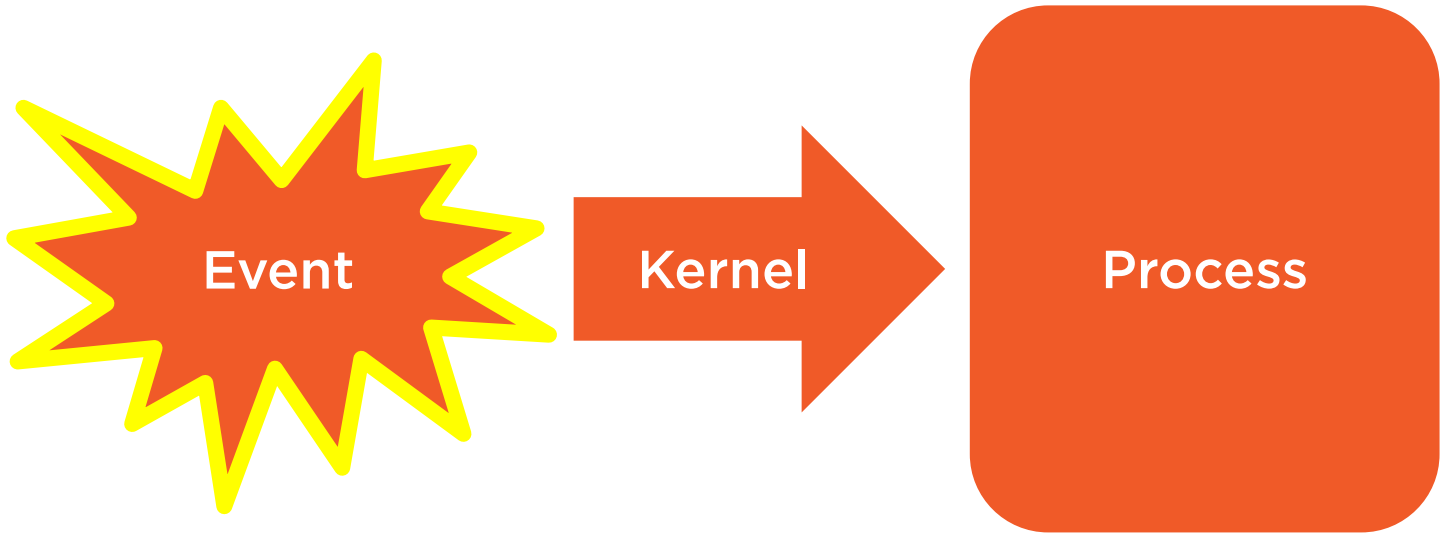
Simple Filter Template

```
def process_file(f):
    for line in f:
        pass  # Not implemented yet

# Start here

if (len(sys.argv)) == 1:
    process_file(sys.stdin)
else:
    for path in sys.argv[1:]:
        try:
            file = open(path, "r")
        except Exception as e:
            print("%s" % e, file=sys.stderr)
            continue
        process_file(file)
        close(file)
```

Signals and What to Do with Them

What are Signals?

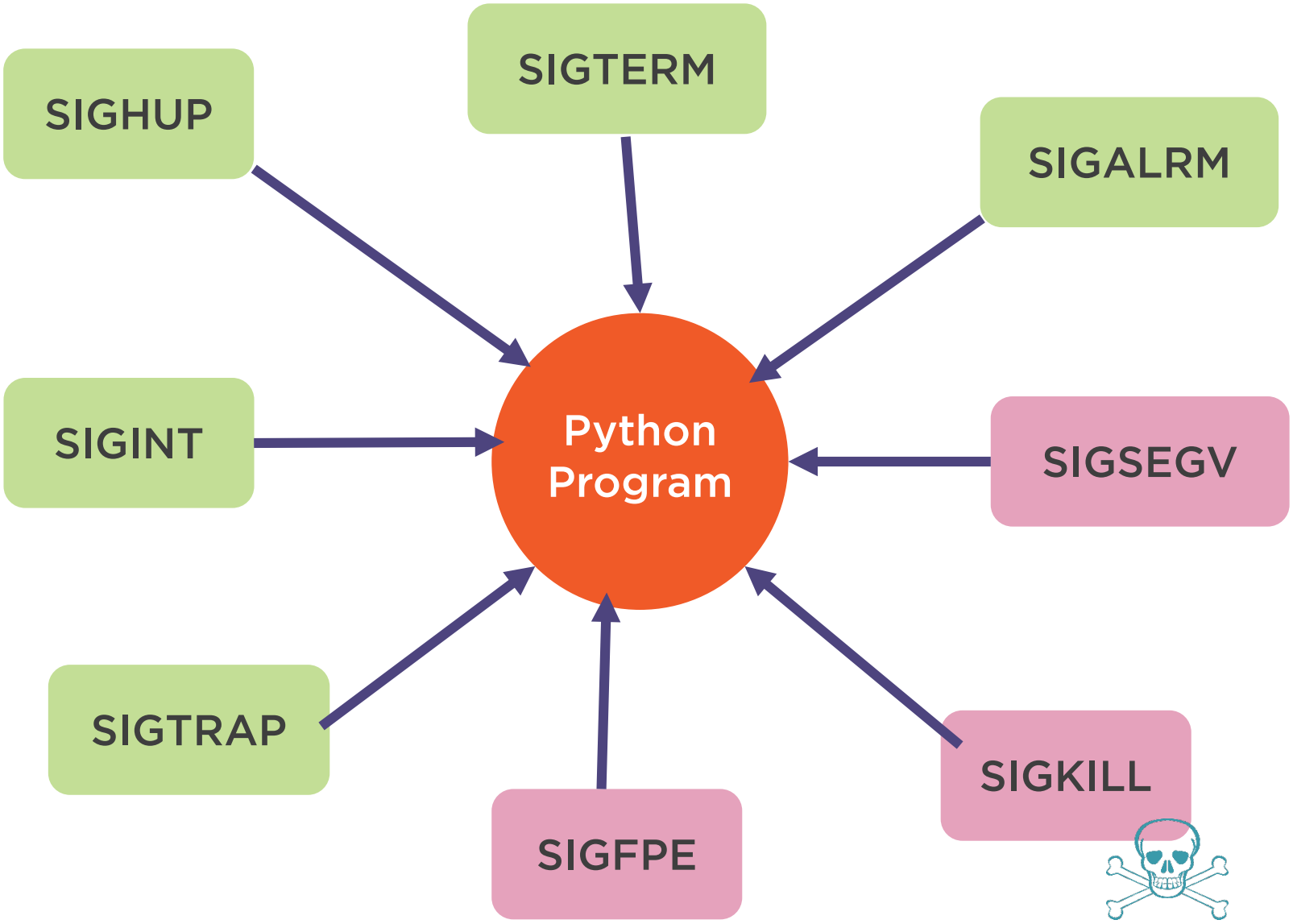
A signal is an event (usually asynchronous) delivered to a process by the kernel



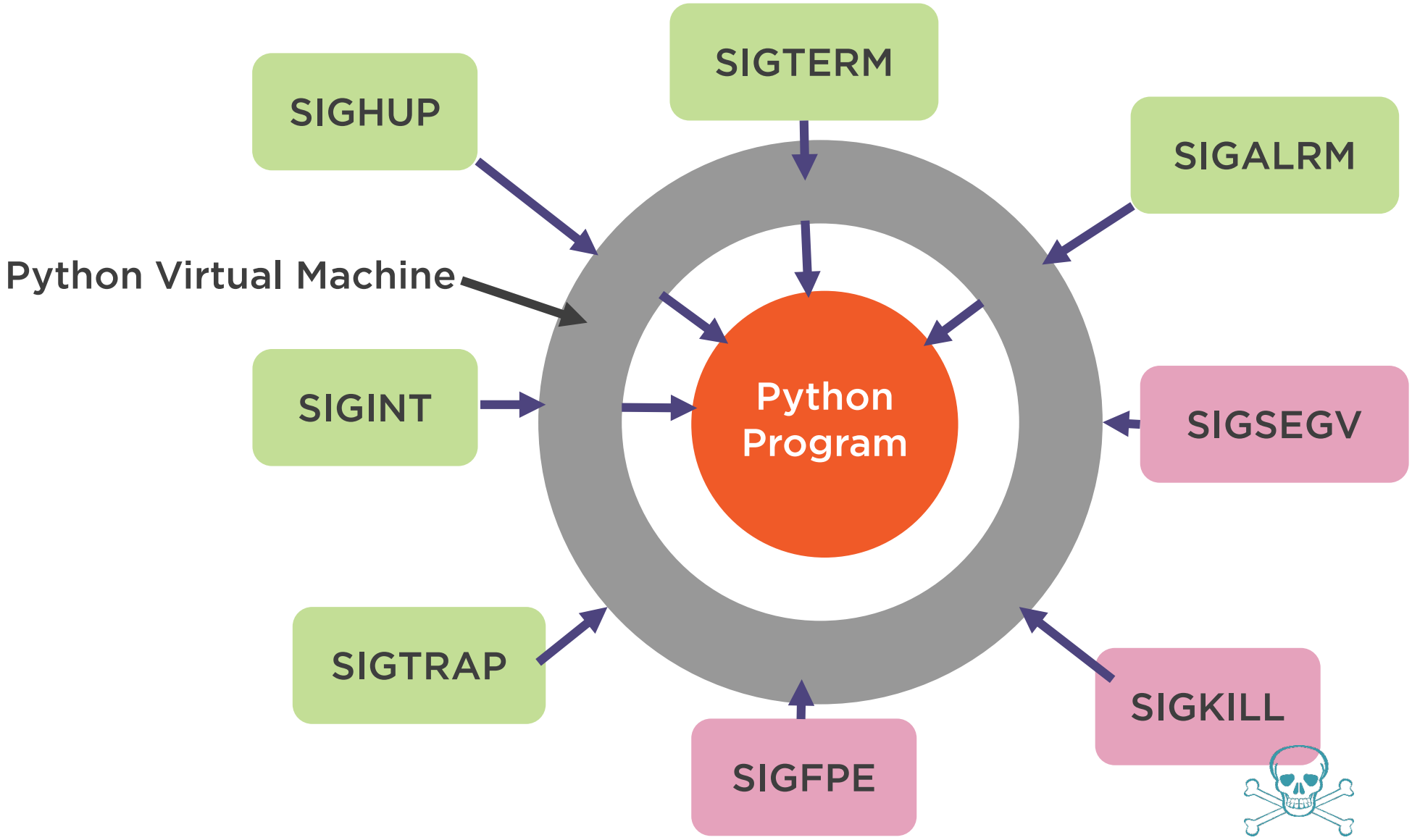
Signal Types

Signal Name	Number	Default Action	Description
SIGHUP	1	Term	Some daemons interpret this as "re-read your configuration file"
SIGINT	2	Term	The signal sent by ^C on terminal
SIGTRAP	5	Core	Trace/breakpoint trap
SIGFPE	8	Core	Arithmetic error, e.g. divide by zero
SIGKILL	9	Term	Lethal signal, cannot be caught
SIGUSR1	10	Term	For user-defined purposes
SIGSEGV	11	Core	Invalid memory reference
SIGALRM	14	Term	Expiry of alarm clock timer
SIGTERM	15	Term	Polite "please terminate" signal

Signal Delivery



Signal Delivery



Signal Demonstrations

Ignoring Signals

Turn debugging on/off

Print current status

Implement a timeout

Summary

- Slicing
  - Strings, lists, tuples
- Parsing command-line arguments
  - optargs
- Accessing environment variables
- Files, streams and filters
  - "File-like" objects
  - stdin, stdout, stderr
- Signals
  - How to ignore them
  - How to catch them

In the Next Lesson

Combining Python with other tools

Running external commands

Creating and using pipes

Handling common file formats

Sending mail