Streams

Alexander Evgin

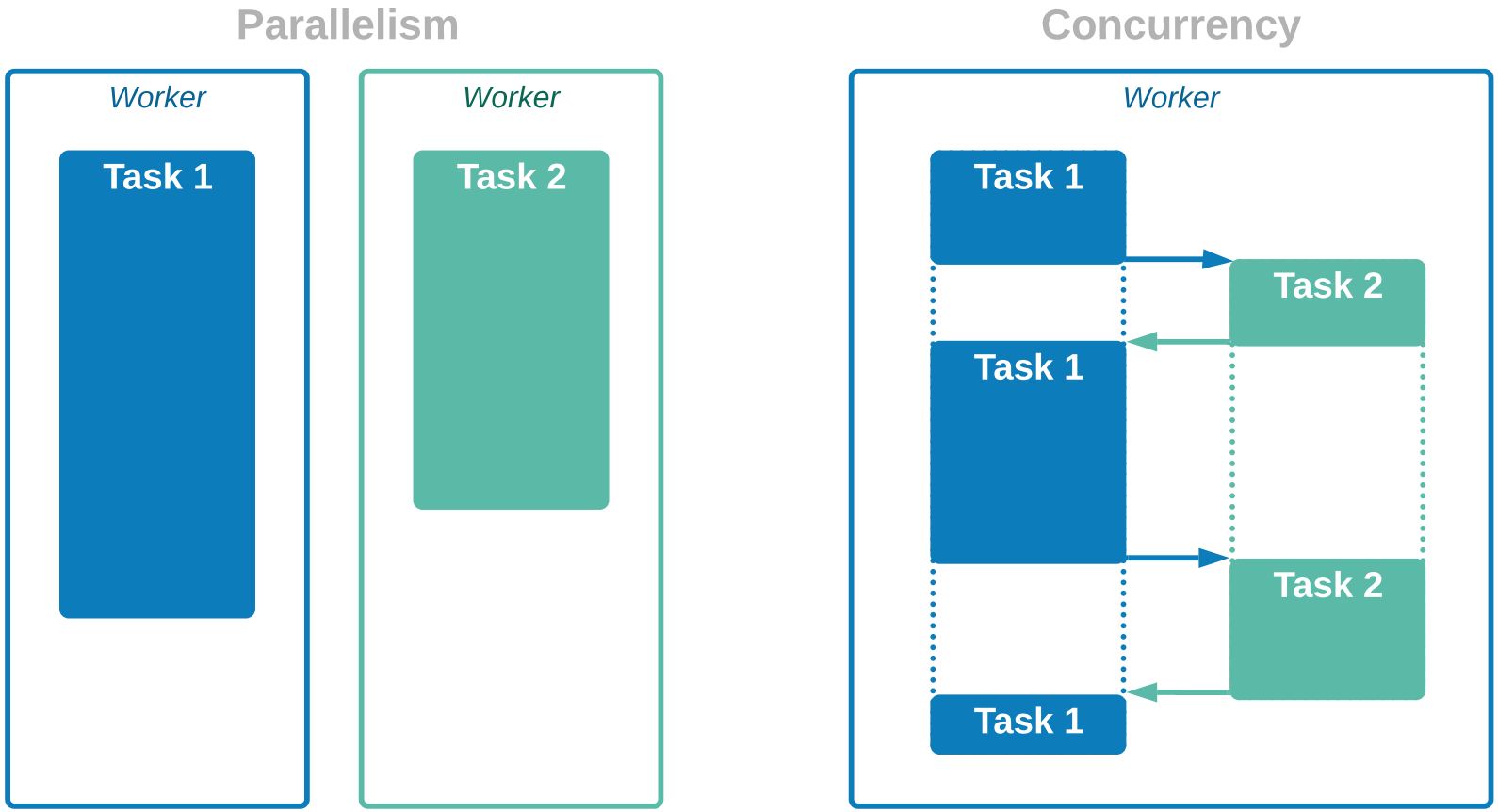
17 апреля 2020 г.

Outline

Recap

- Process (process environment)
- Thread (inside process, share its memory)
- IPC — Inter-process communication (shared memory, pipe, socket, ***message queue***).
- Mutex — примитив синхронизации потоков (`acquire`, `release`).
- GIL — глобальный мьютекс интерпретатора, который гарантирует, что в каждый момент времени только один поток имеет доступ к внутреннему состоянию интерпретатора.
- Асинхронные методы — кооперативная многозадачность.

Recap


Parallelism
Worker
Task 1
Worker
Task 2
Concurrency
Worker
Task 1
Task 2
Task 1
Task 2
Task 1

Command-line arguments

Command-line arguments

```
~$ python hello.py
Hello, world

~$ python hello.py Alice
Hello, Alice
```

Command-line arguments

hello.py:

```
1  import sys
2  assert sys.argv[0] == "hello.py"
3
4  if len(sys.argv) > 1:
5      print('Hello,', sys.argv[1])
6  else:
7      print('Hello, world')
```

Command-line arguments

Usage:

```
python [options] [-c cmd | -m mod | file | - ] [args]
```

Command-line arguments

Usage:

```
python [options] [-c cmd | -m mod | file | - ] [args]
```

```
~$ python server.py Pretty-Server \
                    -p 8000 \
                    --bind 127.0.0.1 \
                    -v
```

```
~$ python -c "import sys; print(sys.argv[1])" \
      Hello, world!
```

argparse

main.py:

```python
import argparse

parser = argparse.ArgumentParser()
parser.add_argument('-f', '--foo', help='foo help')
parser.add_argument('bar', help='bar help')

args = parser.parse_args()
print(args)
```

```
~$ python main.py -f FOO BAR
Namespace(bar='BAR', foo='FOO')
```

argparse

```
~$ python main.py -h
usage: main.py [-h] [-f FOO] bar

positional arguments:
  bar                   bar help

optional arguments:
  -h, --help            show this help message and exit
  -f FOO, --foo FOO     foo help
```

argparse: actions

- "store_true"

```
parser.add_argument('--foo', action='store_true')
```

- "append"

```
parser.add_argument('--bar', action='append')
```

- "count"

```
parser.add_argument('-v', action='count')
```

```
~$ python main.py --foo --bar 1 --bar 2 -vvv
Namespace(bar=['1', '2'], foo=True, v=3)
```

argparse: options

- "nargs"

```
parser.add_argument('--foo', nargs=2)
```

- "default"

```
parser.add_argument('--bar', default=42)
```

- "type"

```
parser.add_argument('baz', type=int)
```

- "required"

```
parser.add_argument('--qux', required=True)
```

```
~$ python main.py --foo 1 2 --qux XYZ 999
Namespace(bar=42, baz=999, foo=['1', '2'], qux='XYZ')
```

Streams

Streams

Stream (стрим, поток) — абстрактная модель данных, представляющая собой последовательность атомарных объектов (байтов, чисел, символов и т.п.), элементы которой становятся доступны в течение времени.

Операции:

- ***read*** from stream
- ***write*** to stream

(Замечание: названия методов конкретных потоков могут отличаться)

В Python потоки часто называют `file object` (или `file-like object`). Иногда реализация потока чтения представляет собой ***итератор***.

Streams

Примеры:

- Файл в файловой системе

```
f = open('file.txt')  # f - file object
```

- Сокет

```
request = sock.recv(1024)
sock.sendall(response)
```

- Стандартные потоки ввода-вывода

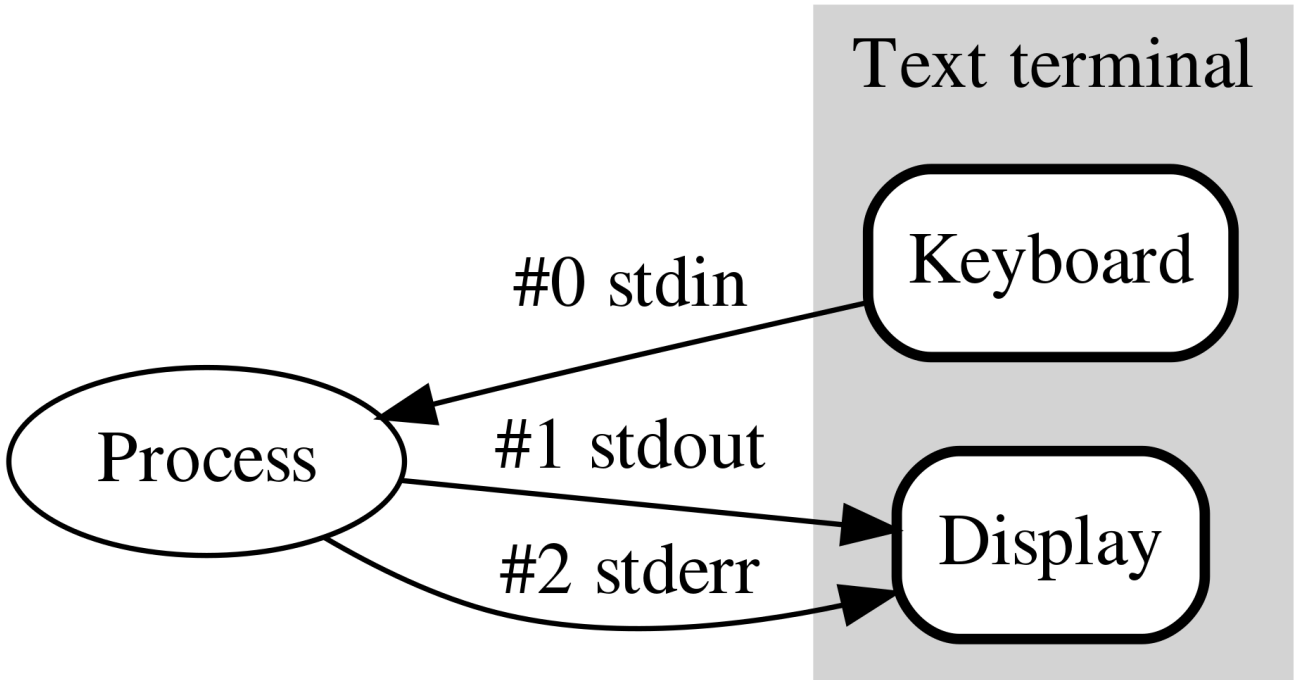
Streams

- **In-memory data**
  - Доступна целиком в любое время
  - Необходима память под весь объем данных

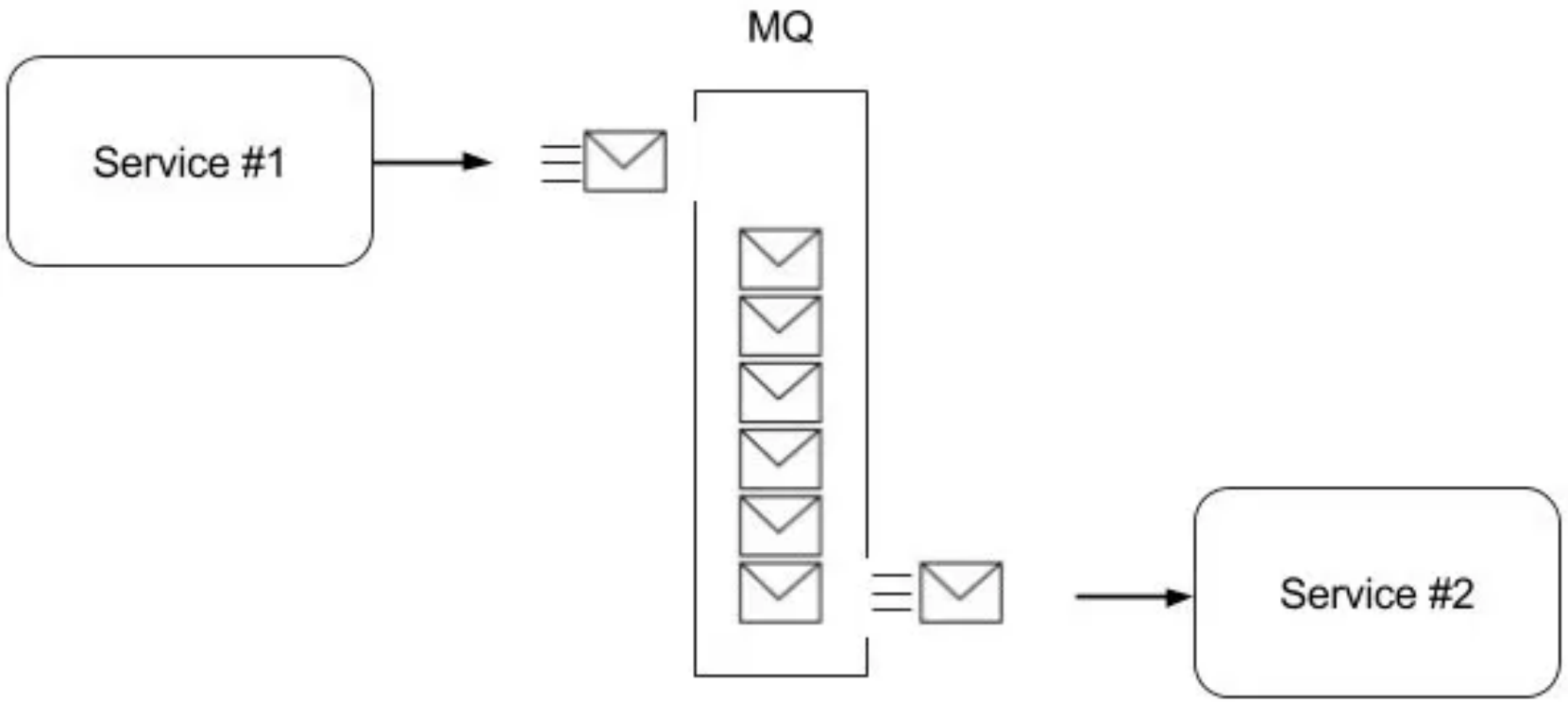
- **Stream data**
  - Каждый элемент доступен только после того, как был “прочитан” из потока
  - Необходима память только под один элемент (в оптимальном случае)

Standard streams

- `stdin`
- `stdout`
- `stderr`

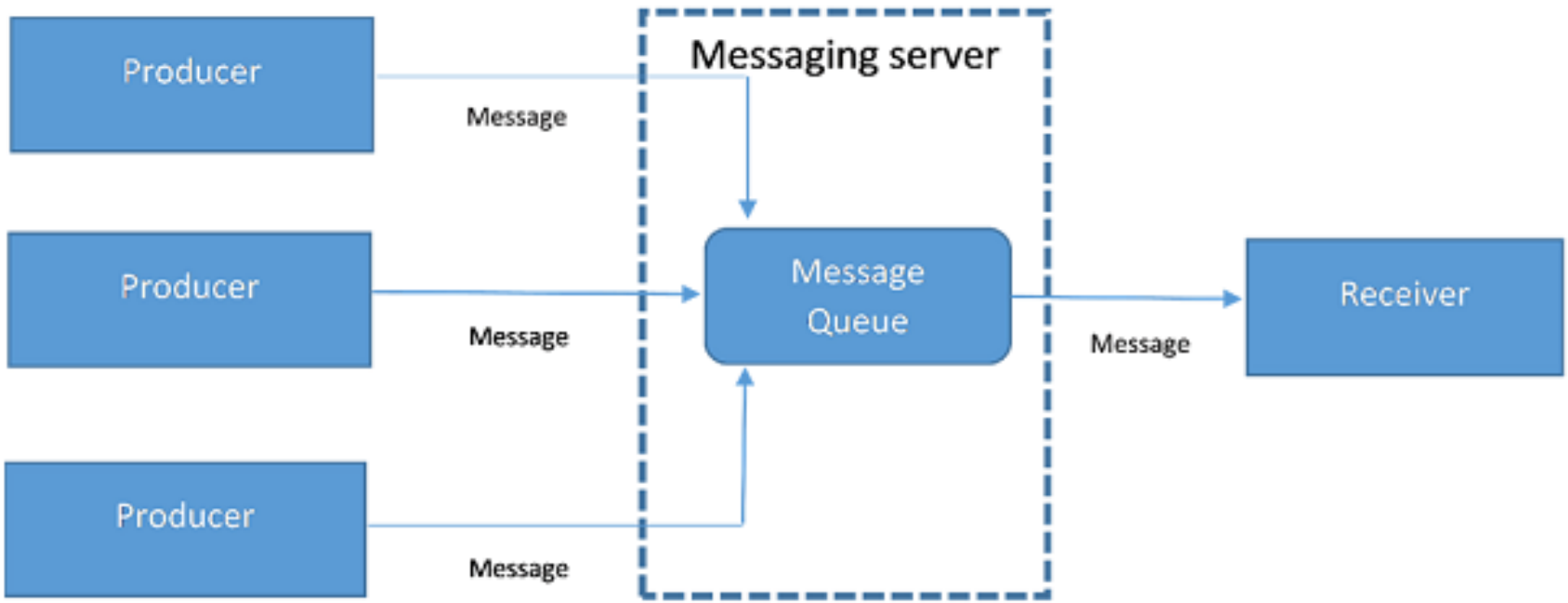


Message queue



Message queue with broker

- Clients subscribe to broker channels
- Producers publish to channel



io — BytesIO, StringIO

```
import io

f = io.StringIO('some initial text data')
assert f.read()[:4] == 'some'

f = io.BytesIO(b'some initial binary data: \x00\x01')
assert f.read()[:4] == b'some'



f = io.BytesIO()
f.write(b'some new data here')
f.getvalue()  # f.getbuffer()
```

io

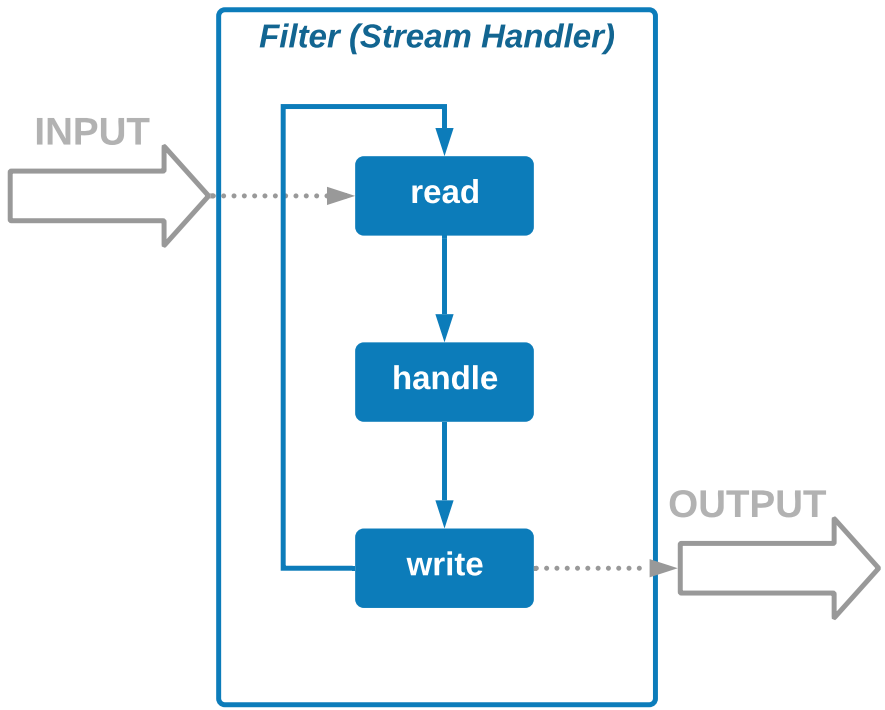
```
f = io.BytesIO()
f.write(b'foobar')
f.read()  # ?
```

io

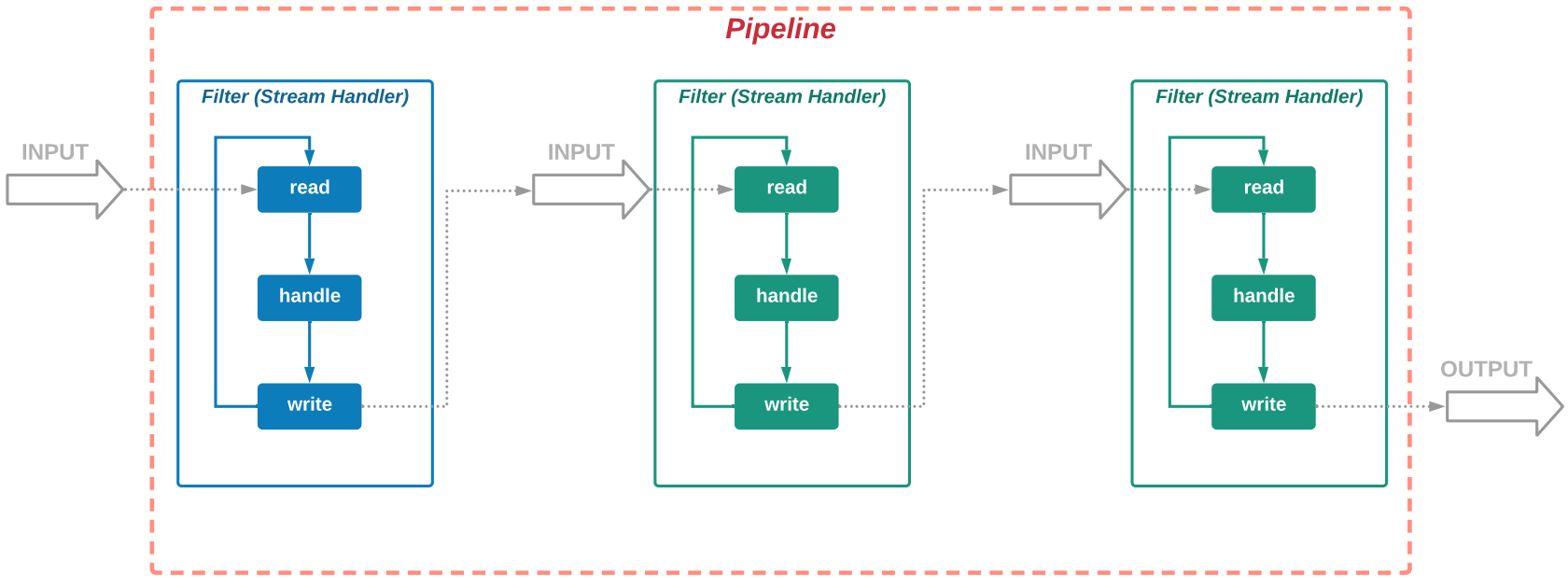
```
f = io.BytesIO()
f.write(b'foobar')
f.seek(0)
f.read()  # b'foobar'
```

Filter (stream handler)

- read
- handle
- write



Pipeline


Pipeline
Filter (Stream Handler)
Filter (Stream Handler)
Filter (Stream Handler)
INPUT
INPUT
INPUT
read
handle
write
read
handle
write
read
handle
write
OUTPUT

Logging

Why do we need logging?

- Debug code (“see” what is going on)
- Manage working systems (track events while software runs)
- Collect statistics

Логгер — черный ящик самолета вашего кода.

print

```
data = load()
print('Loaded {} bytes'.format(len(data))
process(data)
```

- `print` aimed to deliver content to stdout
- Not integrable
- Hard to configure

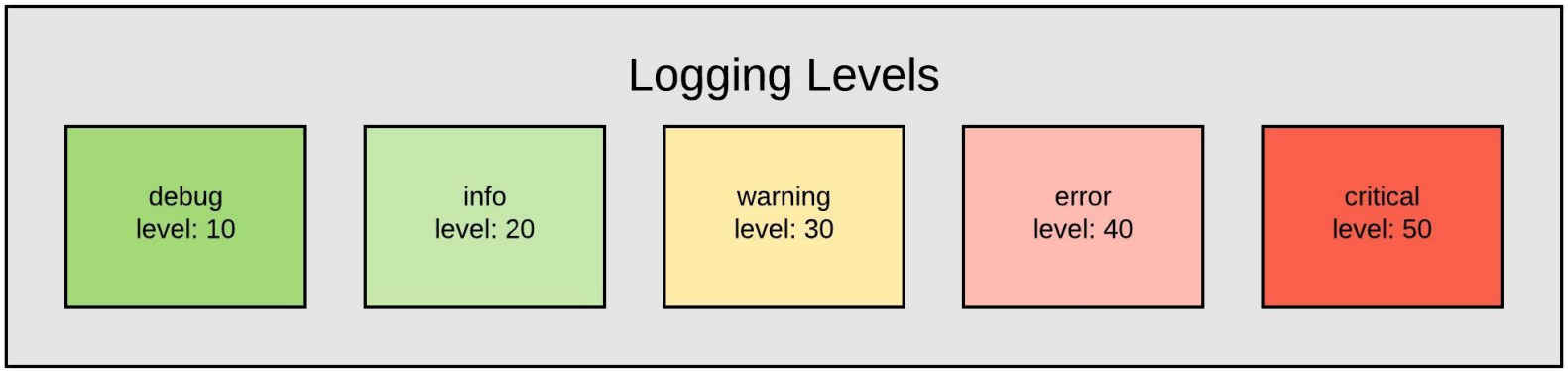
logging

Simple example:

```
1  import logging
2
3  logging.warning('Watch out!')
4  logging.info('I told you so')
```

```
WARNING:root:Watch out!
```

Logging levels



Basic examples

```
1  import logging
2  LOG_MESSAGE = '[%(asctime)s] %(name)s - %(levelname)s - %(message)s'
3  logging.basicConfig(filename='example.log',
4                      level=logging.DEBUG,
5                      format=LOG_MESSAGE)
6  logger = logging.getLogger('MyLogger')
7
8  logger.info('This message should appear in file')
```

example.log:

```
[2020-04-17 00:09:26,214] MyLogger - INFO - This
message should appear in file
```