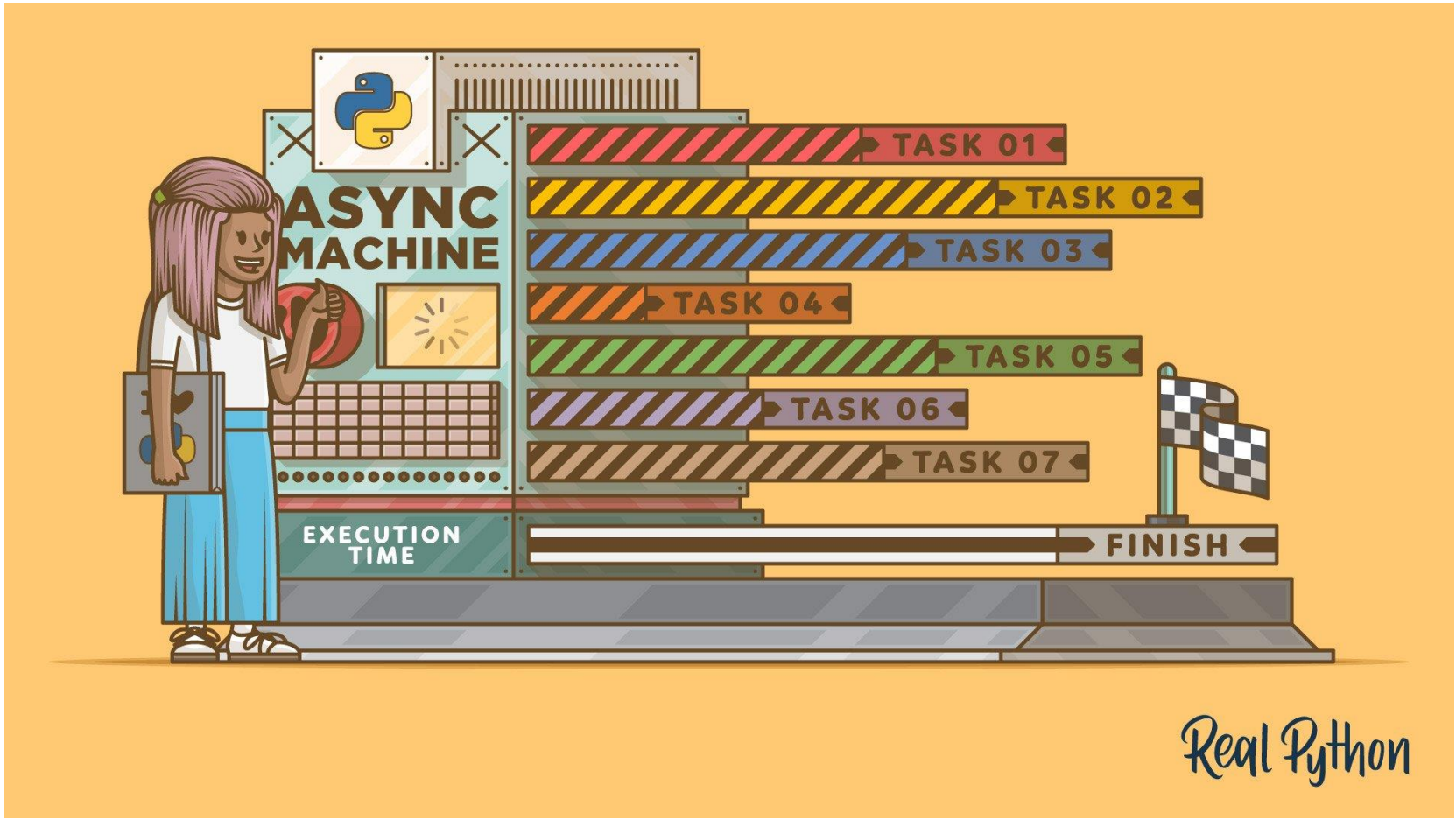
A CLASSIC THREADING PROBLEM

CONSUMER / PRODUCER


ASYNC
MACHINE
TASK 01
TASK 02
TASK 03
TASK 04
TASK 05
TASK 06
TASK 07
EXECUTION
TIME
FINISH
Real Python

Synchronous vs Asynchronous

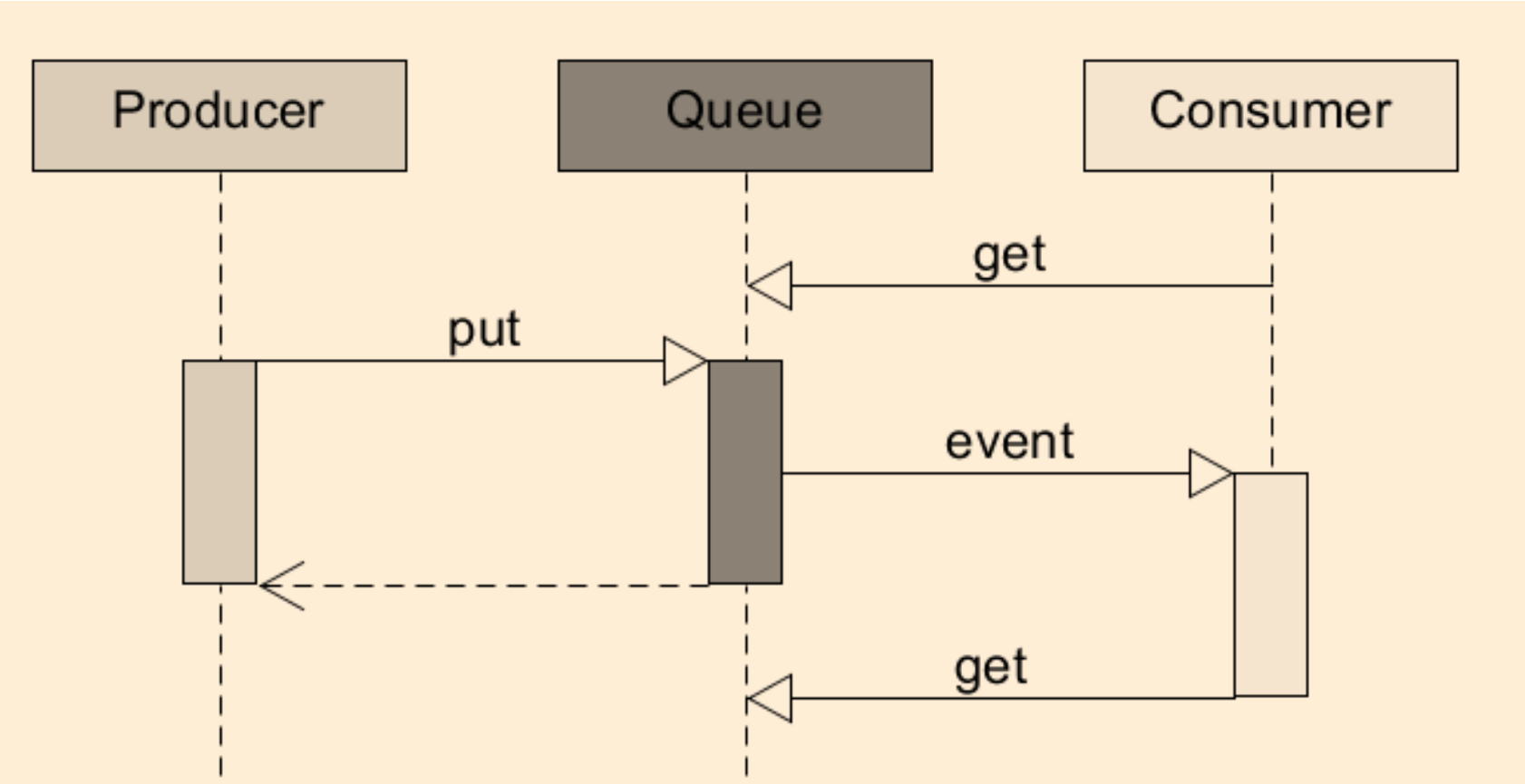
Background

- The producer–consumer problem (also known as the bounded-buffer problem) is a classic example of a multi-process synchronization problem.
- The problem describes two processes, the producer and the consumer, who share a common buffer such as a queue.

Producer/Consumer Design Pattern

- The Producer/Consumer design pattern is based on the Master/Slave pattern, and enhances data sharing between multiple loops running at different rates.
- This design pattern can also be used effectively when analyzing network communication.

Sequence Diagram



Producer

The producer's job is to generate a piece of data, put it into the buffer and start again.

```
def marge(idnum):
    for msgnum in range(nummessages):
        time.sleep(idnum)
        cookiejar.put(idnum)
```

Bounded Buffer

cookiejar = queue.Queue(maxsize = 10)

Consumer

The consumer removes the data from the buffer one piece at a time.

```
def homer(idnum):
    while True:
        time.sleep(0.1)
        try:
            cookie = cookiejar.get()
        except queue.Empty:
            pass
        else:
            print('consumer ', idnum+1, ' got => ', cookie)
```

Lock

- A lock is in one of two states, "locked" or "unlocked".
- When the state is unlocked, **acquire()** changes the state to locked and returns immediately.
- When the state is locked, **acquire()** blocks until a call to **release().**

```
threadLock.acquire()
self.cookies.append(1)
print("{}: produced".format(self.name))
threadLock.release()
```

Semiphore

- A semaphore manages an internal counter which is decremented by each acquire() call and incremented by each release() call.
- The counter can never go below zero; when acquire() finds that it is zero, it blocks, waiting until some other thread calls release().

```
wait()
semiproduce.acquire()
produce()
semiconsume.release()
```

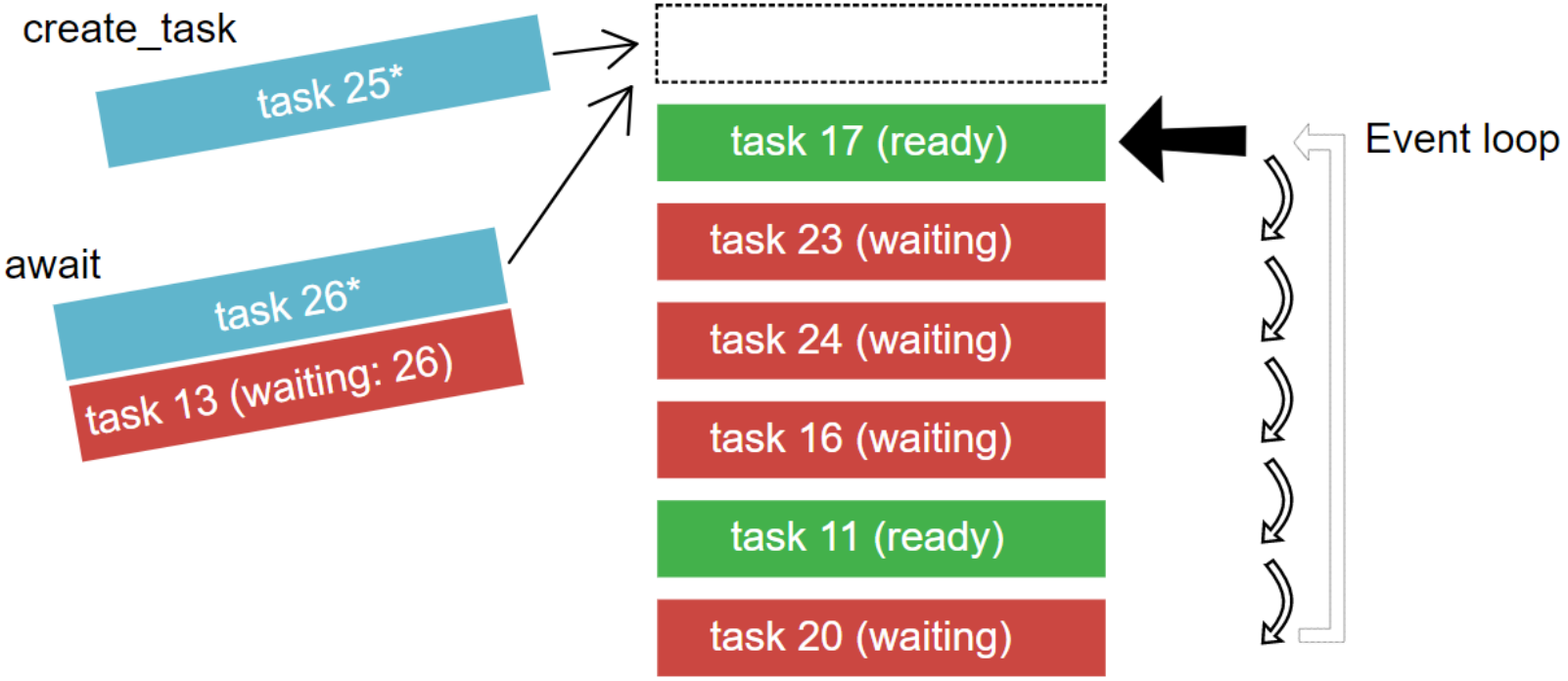
Condition

- Conditional Variable: A *condition variable* is an object able to block the calling thread until *notified* to resume.

```
condition.acquire()
if not cookiejar:
        print("Nothing in cookiejar.")
        condition.wait()
        print("Producer added cookie.")
cookie = cookiejar.pop(0)
print("Consumed", cookie)
condition.release()
```

Async

- asyncio uses threading constructs: event loops, coroutines and futures.
- An event loop manages and distributes the execution of different tasks.
- Coroutines (covered are special functions that work similarly to Python generators, on **await** they release the flow of control back to the event loop.
- Futures represent the result of a task that may or may not have been executed.


create_task
task 25*
await
task 26*
task 13 (waiting: 26)
task 17 (ready)
task 23 (waiting)
task 24 (waiting)
task 16 (waiting)
task 11 (ready)
task 20 (waiting)
Event loop