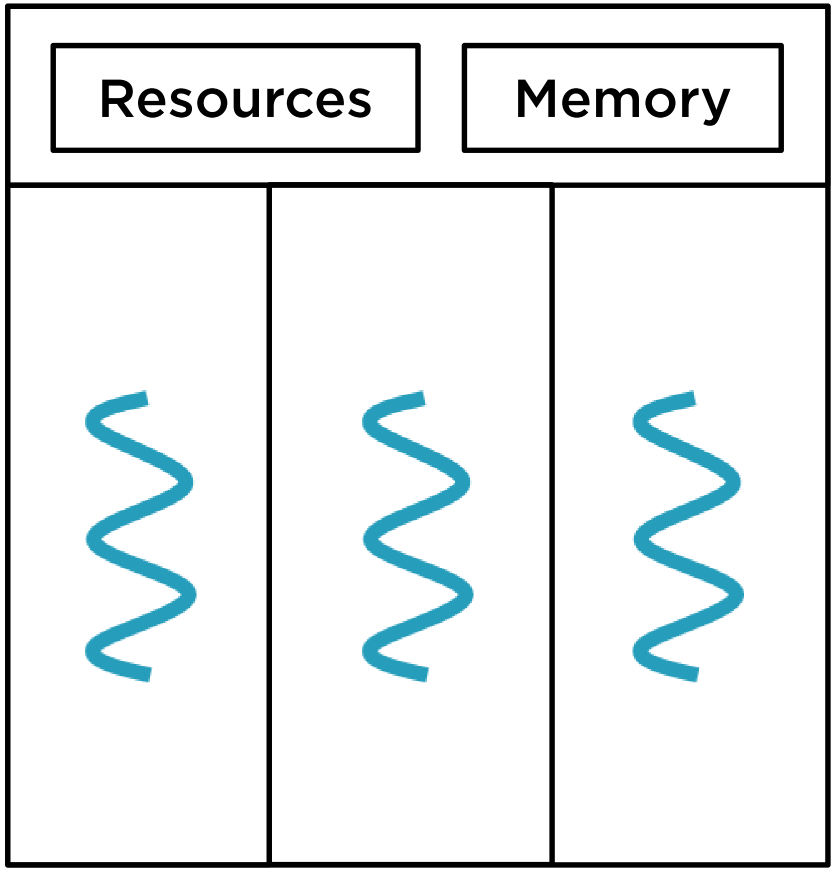
Multiprocessing

Tim Ojo
@tim_ojo www.timojo.com

Processes vs. Threads

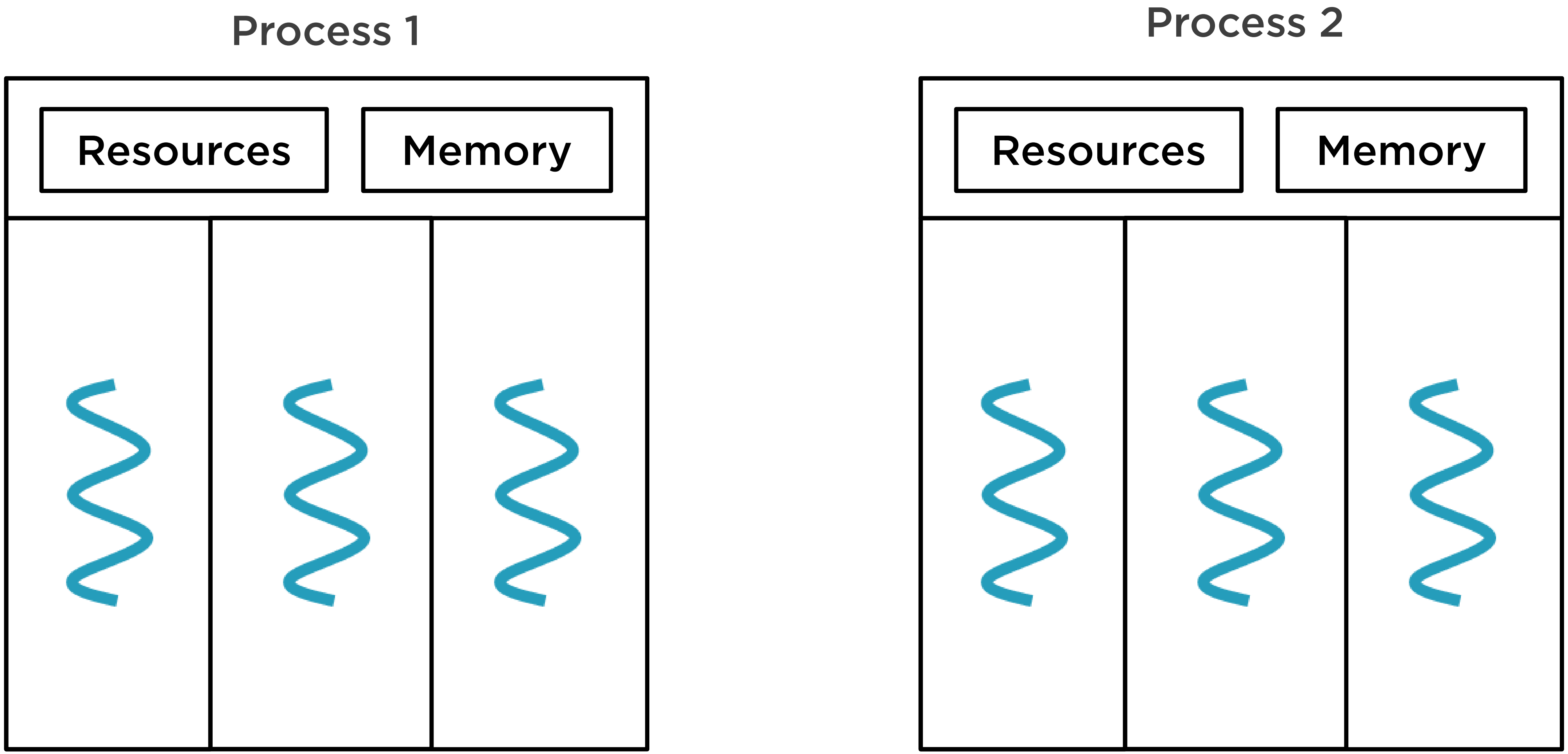
Process

a running instance of a computer program



Resources

Memory



Process 1

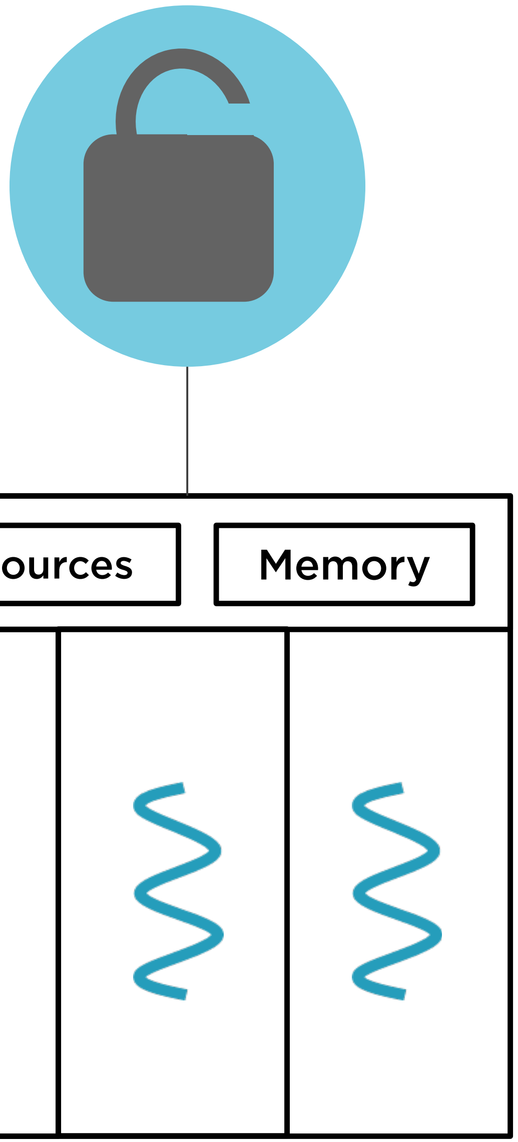
Resources

Memory

Process 2

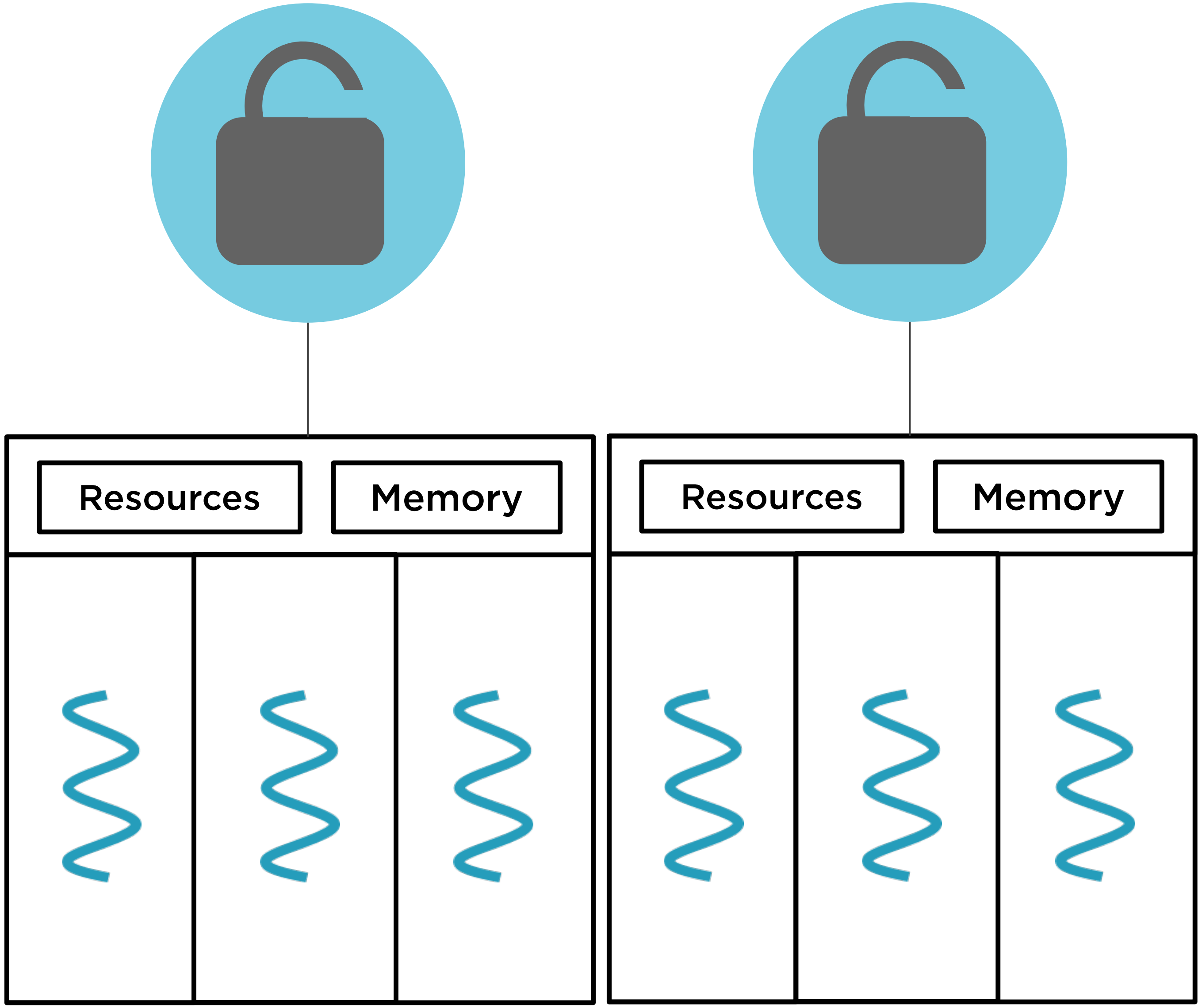
Resources

Memory



Resources

Memory


Resources
Memory
Resources
Memory

Processes vs. Threads

- **Sidesteps GIL**

Processes vs. Threads

- Sidesteps GIL
- Less need for synchronization

Processes vs. Threads

- Sidesteps GIL
- Less need for synchronization
- Can be paused and terminated

Processes vs. Threads

- Sidesteps GIL
- Less need for synchronization
- Can be paused and terminated
- More resilient

Processes vs. Threads

- Sidesteps GIL
- Less need for synchronization
- Can be paused and terminated
- More resilient

- Higher memory footprint
- Expensive context switches

Multiprocessing API

```
import threading

def do_some_work(val):
    print ("doing some work in thread")
    print ("echo: {}".format(val))
    return

    val =  "text"
    t = threading.Thread(target=do_some_work, args=(val,))
    t.start()
    t.join()

```

```
import multiprocessing

def do_some_work(val):
    print ("doing some work in thread")
    print ("echo: {}".format(val))
    return

if __name__ == '__main__':
    val =  "text"
    t = multiprocessing.Process(target=do_some_work, args=(val,))
    t.start()
    t.join()
```

Pickling

is the process whereby a Python object hierarchy is converted into a byte stream. “unpickling” is the inverse operation

Picklable Objects

None, True, False

Integers, floats, complex numbers

Normal and Unicode strings

Collections containing only picklable objects

Top level functions

Classes with picklable attributes

```
import multiprocessing

def do_some_work(val):
    print ("doing some work in thread")
    print ("echo: {}".format(val))
    return

if __name__ == '__main__':
    val =  "text"
    t = multiprocessing.Process(target=do_some_work, args=(val,))
    t.start()
    t.join()
```

```
import multiprocessing

print("outer: this will print when run and when imported")

def func():
    print("func: this will print only when run")
    return

if __name__ == '__main__':
    print("main: this will print only when run")
    p = multiprocessing.Process(target=func)
    p.start()
    p.join()
```

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

```
class multiprocessing.Process(group=None,
                             target=None,
                             name=None,
                             args=(),
                             kwargs={},
                             daemon=None)
```

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

```
class multiprocessing.Process(group=None,
                             target=None,
                             name=None,
                             args=(),
                             kwargs={},
                             daemon=None)
```

```
class multiprocessing.Process(group=None,
                             target=None,
                             name=None,
                             args=(),
                             kwargs={},
                             daemon=None)
```

```
class multiprocessing.Process(group=None,
                              target=None,
                              name=None,
                              args=(),
                              kwargs={},
                              daemon=None)
```

Daemon Process

is a child process that does not prevent its parent process from exiting

Terminating Processes

Terminating Processes

- is_alive()
- terminate()

```
import multiprocessing
import time

def do_work():
    print (’Starting do_work function’)
    time.sleep(5)
    print (’Finished do_work function’)

if __name__ == ’__main__’:
    p = multiprocessing.Process(target=func)
    print (”[Before Start] Process is alive: {}”.format(p.is_alive()))
    p.start()
    print (”[Running] Process is alive: {}”.format(p.is_alive()))
    p.terminate()
    p.join()
    print (”[After Termination] Process is alive: {}”.format(p.is_alive()))
    print (”Process exit code: {}”.format(p.exitcode))
```

```python
import multiprocessing
import time

def do_work():
   print ('Starting do_work function')
   time.sleep(5)
   print ('Finished do_work function')

if __name__ == '__main__':
   p = multiprocessing.Process(target=func)
   print ("[Before Start] Process is alive: {}".format(p.is_alive()))
   p.start()
   print ("[Running] Process is alive: {}".format(p.is_alive()))
   p.terminate()
   p.join()
   print ("[After Termination] Process is alive: {}".format(p.is_alive()))
   print ("Process exit code: {}".format(p.exitcode))
```

```
import multiprocessing
import time

def do_work():
    print (’Starting do_work function’)
    time.sleep(5)
    print (’Finished do_work function’)

if __name__ == ’__main__’:
    p = multiprocessing.Process(target=func)
    print (”[Before Start] Process is alive: {}”.format(p.is_alive()))
    p.start()
    print (”[Running] Process is alive: {}”.format(p.is_alive()))
    p.terminate()
    p.join()
    print (”[After Termination] Process is alive: {}”.format(p.is_alive()))
    print (”Process exit code: {}”.format(p.exitcode))
```

```
import multiprocessing
import time

def do_work():
    print (’Starting do_work function’)
    time.sleep(5)
    print (’Finished do_work function’)

if __name__ == ’__main__’:
    p = multiprocessing.Process(target=func)
    print (”[Before Start] Process is alive: {}”.format(p.is_alive()))
    p.start()
    print (”[Running] Process is alive: {}”.format(p.is_alive()))
    p.terminate()
    p.join()
    print (”[After Termination] Process is alive: {}”.format(p.is_alive()))
    print (”Process exit code: {}”.format(p.exitcode))
```

```
import multiprocessing
import time

def do_work():
    print (’Starting do_work function’)
    time.sleep(5)
    print (’Finished do_work function’)

if __name__ == ’__main__’:
    p = multiprocessing.Process(target=func)
    print (”[Before Start] Process is alive: {}”.format(p.is_alive()))
    p.start()
    print (”[Running] Process is alive: {}”.format(p.is_alive()))
    p.terminate()
    p.join()
    print (”[After Termination] Process is alive: {}”.format(p.is_alive()))
    print (”Process exit code: {}”.format(p.exitcode))
```

Process.terminate() gotchas

Process.terminate() gotchas

- Shared resources may be put in an inconsistent state

Process.terminate() gotchas

- Shared resources may be put in an inconsistent state
- Finally clauses and exit handlers will not be run

Process Pools

```
class multiprocessing.Pool([num_processes
                            [, initializer
                            [, initargs
                            [, maxtasksperchild ]]]] ))
```

```
class multiprocessing.Pool([num_processes
                            [, initializer
                            [, initargs
                            [, maxtasksperchild ]]]] ))
```

```
class multiprocessing.Pool([num_processes
                             [, initializer
                             [, initargs
                             [, maxtasksperchild ]]]] ))
```

```
class multiprocessing.Pool([num_processes
                            [, initializer
                            [, initargs # picklable not required
                            [, maxtasksperchild ]]]] ))
```

```
class multiprocessing.Pool([num_processes
                            [, initializer
                            [, initargs
                            [, maxtasksperchild ]]]] ))
```

```
map(func, iterable[, chunksize])
```

```python
import multiprocessing

def do_work(data):
    return data**2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process)
    inputs = list(range(10))
    outputs = pool.map(do_work, inputs)
    print('Outputs :', outputs)

    pool.close() # no more tasks
    pool.join()  # wait for the worker processes to exit
```

```python
import multiprocessing

def do_work(data):
    return data**2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                       initializer=start_process)
    inputs = list(range(10))
    outputs = pool.map(do_work, inputs)
    print('Outputs :', outputs)

    pool.close() # no more tasks
    pool.join()  # wait for the worker processes to exit
```

```
import multiprocessing

def do_work(data):
    return data**2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process)
    inputs = list(range(10))
    outputs = pool.map(do_work, inputs)
    print('Outputs :', outputs)

    pool.close() # no more tasks
    pool.join()  # wait for the worker processes to exit
```

```
import multiprocessing

def do_work(data):
    return data**2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                                initializer=start_process)
    inputs = list(range(10))
    outputs = pool.map(do_work, inputs)
    print('Outputs :', outputs)

    pool.close() # no more tasks
    pool.join()  # wait for the worker processes to exit
```

```python
import multiprocessing

def do_work(data):
    return data**2

def start_process():
    print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
    pool_size = multiprocessing.cpu_count() * 2
    pool = multiprocessing.Pool(processes=pool_size,
                     initializer=start_process)
    inputs = list(range(10))
    outputs = pool.map(do_work, inputs)
    print('Outputs :', outputs)

    pool.close() # no more tasks
    pool.join()  # wait for the worker processes to exit
```

```python
import multiprocessing

def do_work(data):
   return data**2

def start_process():
   print('Starting', multiprocessing.current_process().name)

if __name__ == '__main__':
   pool_size = multiprocessing.cpu_count() * 2
   pool = multiprocessing.Pool(processes=pool_size,
                               initializer=start_process)
   inputs = list(range(10))
   outputs = pool.map(do_work, inputs)
   print('Outputs :', outputs)

   pool.close() # no more tasks
   pool.join()  # wait for the worker processes to exit
```

Demo

Process Pool Demo

```
map_async(func, iterable[, chunksize[, callback]]) returns AsyncResult
map_async(func, iterable[, chunksize[, callback]]) returns AsyncResult
```

```
AsyncResult.get([timeout]) # returns the result when it arrives
```

```
apply(func[, args[, kwargs]])
```

```
apply_async(func[, args[, kwargs[, callback[, error_callback]]]])
```

```
from multiprocessing import Pool
import time

def multiply(x, y):
    return x*y

if __name__ == '__main__':
    pool = Pool(processes=4)
    result = pool.apply_async(mult, (7,7)) # evaluate "multiply(7,7)"
                                                  asynchronously in a single process
    print result.get()    # prints 49
```

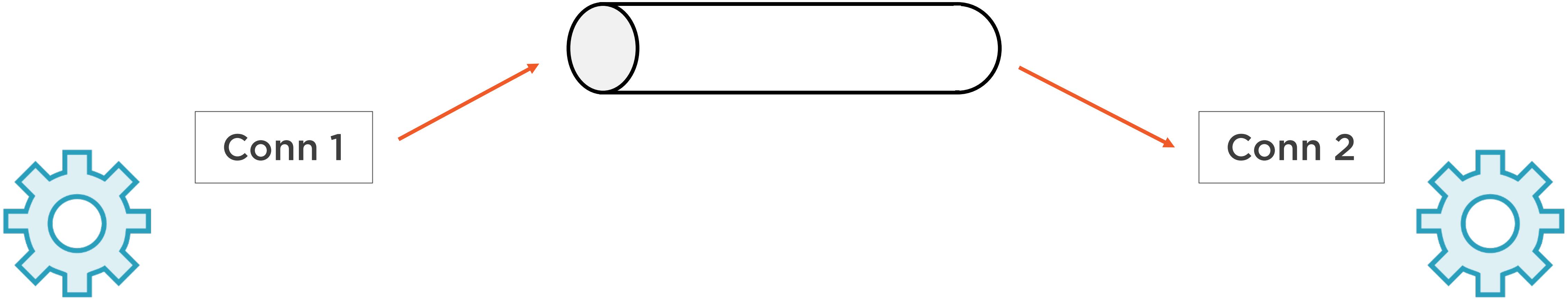
Inter-process Communication

Inter-process Communication Channels

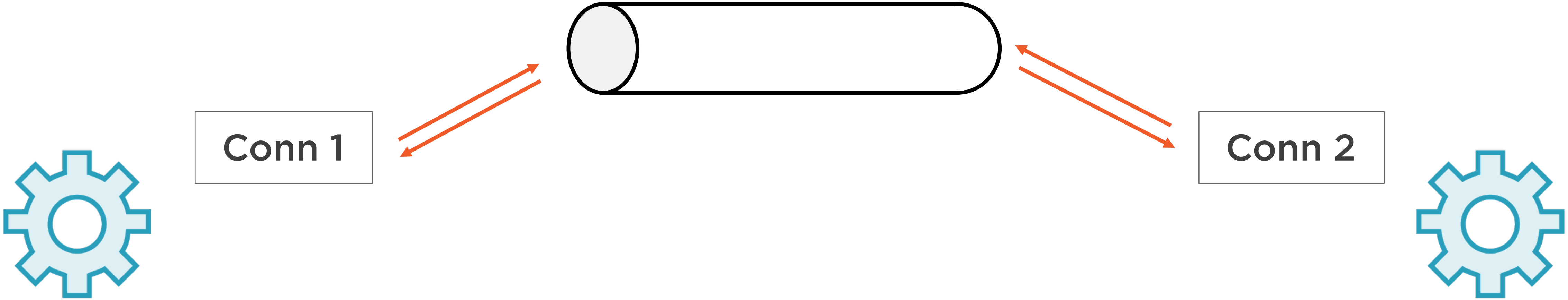
Pipe

Queue

multiprocessing.Pipe



multiprocessing.Pipe

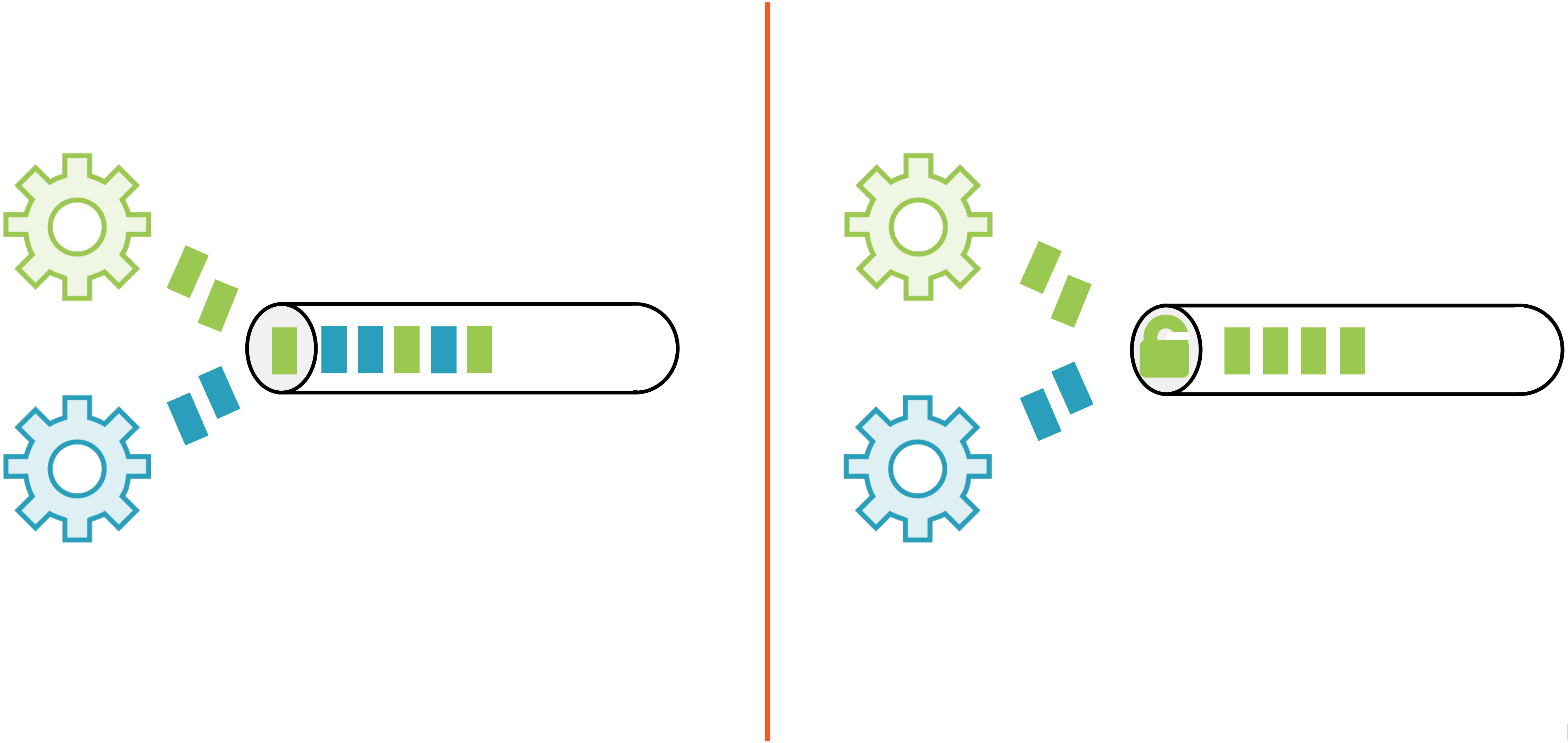


```
def make_tuple(conn):
   num = random.randint(1, 9)
   conn.send(('Hi ', num))
   print(conn.recv())

def make_string(conn):
   tup = conn.recv()
   result = ''
   substr, num = tup
   for _ in range(num):
       result += substr
   conn.send(result)

if __name__ == '__main__':
   conn1, conn2 = Pipe(True)
   p1 = Process(target=make_tuple, args=(conn1,))
   p2 = Process(target=make_string, args=(conn2,))
   p1.start()
   p2.start()
```

```python
def make_tuple(conn):
    num = random.randint(1, 9)
    conn.send(('Hi ', num))
    print(conn.recv())

def make_string(conn):
    tup = conn.recv()
    result = ''
    substr, num = tup
    for _ in range(num):
        result += substr
    conn.send(result)

if __name__ == '__main__':
    conn1, conn2 = Pipe(True)
    p1 = Process(target=make_tuple, args=(conn1,))
    p2 = Process(target=make_string, args=(conn2,))
    p1.start()
    p2.start()
```

```
def make_tuple(conn):
    num = random.randint(1, 9)
    conn.send(('Hi ', num))
    print(conn.recv())

def make_string(conn):
    tup = conn.recv()
    result = ''
    substr, num = tup
    for _ in range(num):
        result += substr
    conn.send(result)

if __name__ == '__main__':
    conn1, conn2 = Pipe(True)
    p1 = Process(target=make_tuple, args=(conn1,))
    p2 = Process(target=make_string, args=(conn2,))
    p1.start()
    p2.start()
```

Pipe vs. Queue



Queue Methods

threading.Queue	multiprocessing.Queue
qsize()	qsize()
put()	put()
get()	get()
empty()	empty()
full()	full()
task_done()	~~task_done()~~
join()	~~join()~~

Queue Methods

threading.Queue	multiprocessing.Queue	multiprocessing.JoinableQueue
qsize()	qsize()	qsize()
put()	put()	put()
get()	get()	get()
empty()	empty()	empty()
full()	full()	full()
task_done()	~~task_done()~~	task_done()
join()	~~join()~~	join()

```python
def make_tuple(queue):
    num = random.randint(1, 9)
    queue.put(('Hi ', num))
    print(queue.get())

def make_string(queue):
    tup = queue.get()
    result = ''
    substr, num = tup
    for _ in range(num):
        result += substr
    queue.put(result)

if __name__ == '__main__':
    queue = Queue()
    p1 = Process(target=make_tuple, args=(queue,))
    p2 = Process(target=make_string, args=(queue,))
    p1.start()
    p2.start()
```

```
Queue.get([block[, timeout]])
Queue.put(obj,[[block[, timeout]]])
```

Demo

Inter-Process Communication Demo

Sharing State Between Processes

Shared State

Shared Memory

Manager Process

Shared Memory

- multiprocessing.Value
- multiprocessing.Array

Shared Memory

```
multiprocessing.Value(typecode_or_type, *args[, lock])
```

Shared Memory

```
multiprocessing.Value(typecode_or_type, *args[, lock])
```

ctypes

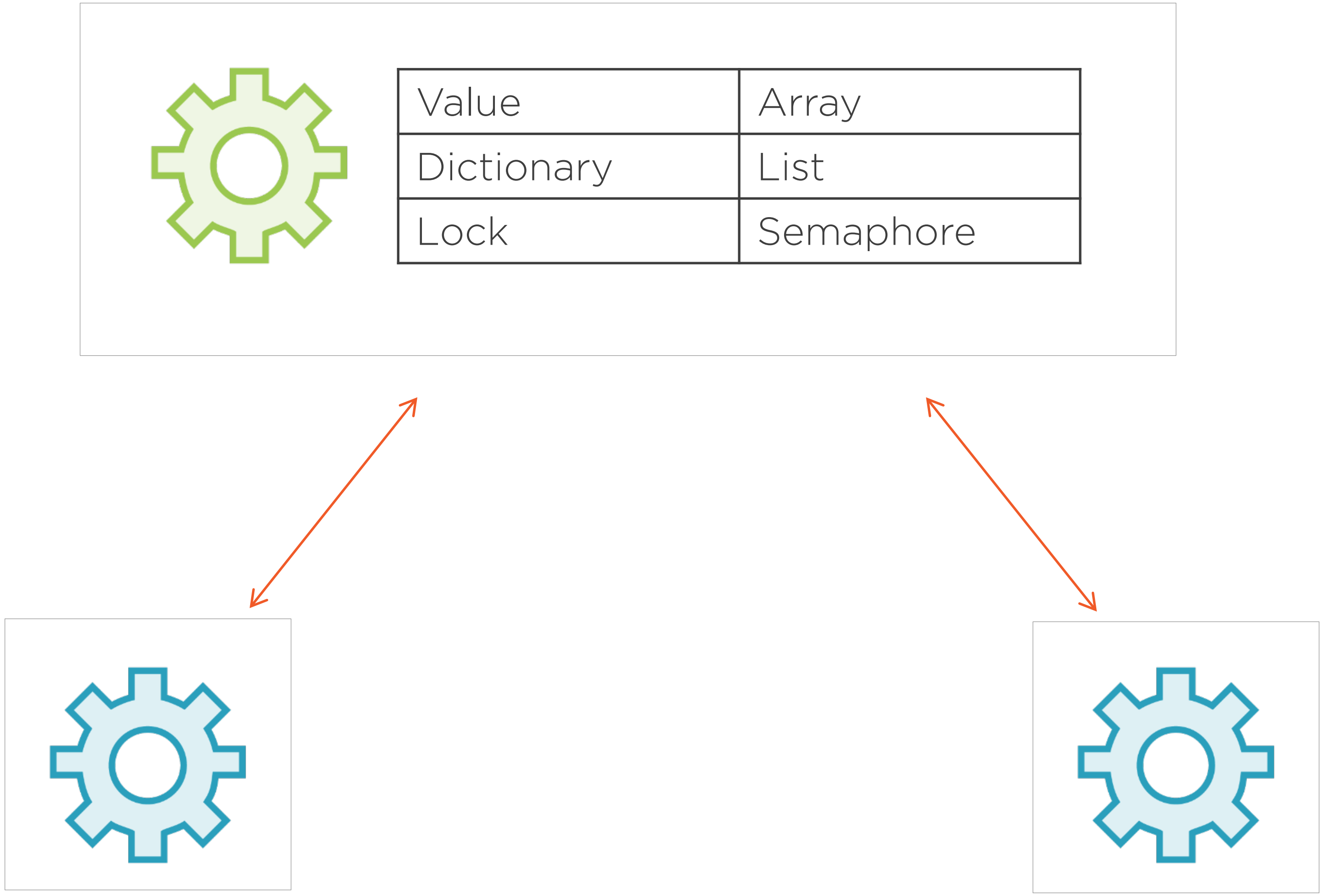
ctypes type	C type	Python type	Type code
c_bool	_Bool	bool(1)	
c_char	char	1-character bytes object	‘c’
c_wchar	wchar_t	1-character string	‘u’
c_int	int	int	‘i’
c_long	long	int	‘l’
c_float	float	float	‘f’
c_char_p	char * (NUL terminated)	bytes object or None	
c_wchar_p	wchar_t * (NUL terminated)	string object or None	
c_void_p	void *	int or None	

```
counter = Value(‘i’)  # shared object of type int, defaults to 0

is_running = Value(ctypes.c_bool, False, lock=False) # shared
object of type boolean, defaulting to False, unsynchronized

my_lock = multiprocessing.Lock()
size_counter = Value(‘l’, 0, lock=my_lock) # shared object of
type long, with a lock specified
```

```
counter = Value(‘i’)  # shared object of type int, defaults to 0

is_running = Value(ctypes.c_bool, False, lock=False) # shared object of type boolean, defaulting to False, unsynchronized

my_lock = multiprocessing.Lock()
size_counter = Value(‘l’, 0, lock=my_lock) # shared object of type long, with a lock specified
```

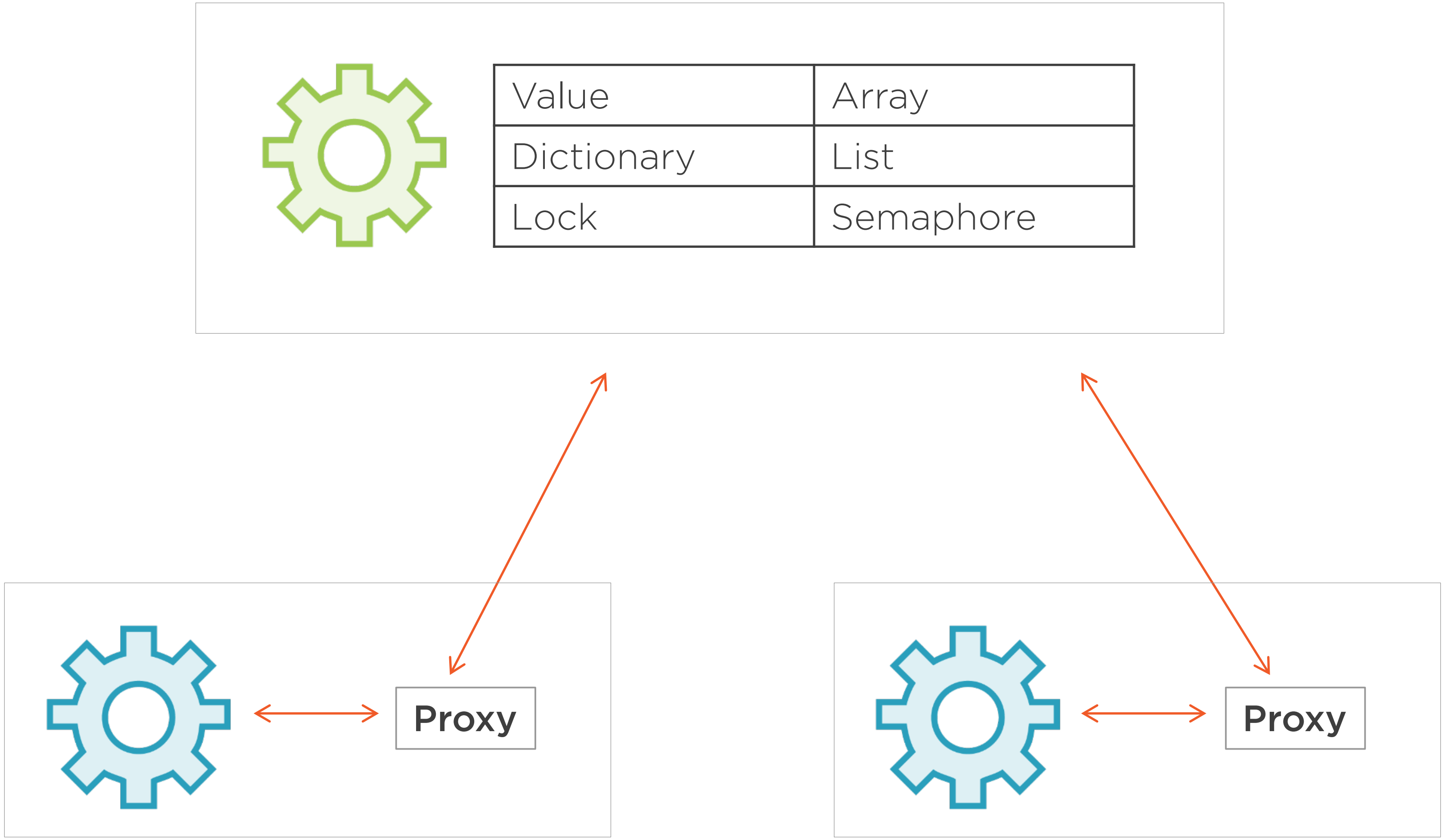
```
counter = Value(‘i’)  # shared object of type int, defaults to 0

is_running = Value(ctypes.c_bool, False, lock=False) # shared
object of type boolean, defaulting to False, unsynchronized

my_lock = multiprocessing.Lock()
size_counter = Value(‘l’, 0, lock=my_lock) # shared object of
type long, with a lock specified
```

Manager



Value	Array
Dictionary	List
Lock	Semaphore

Manager

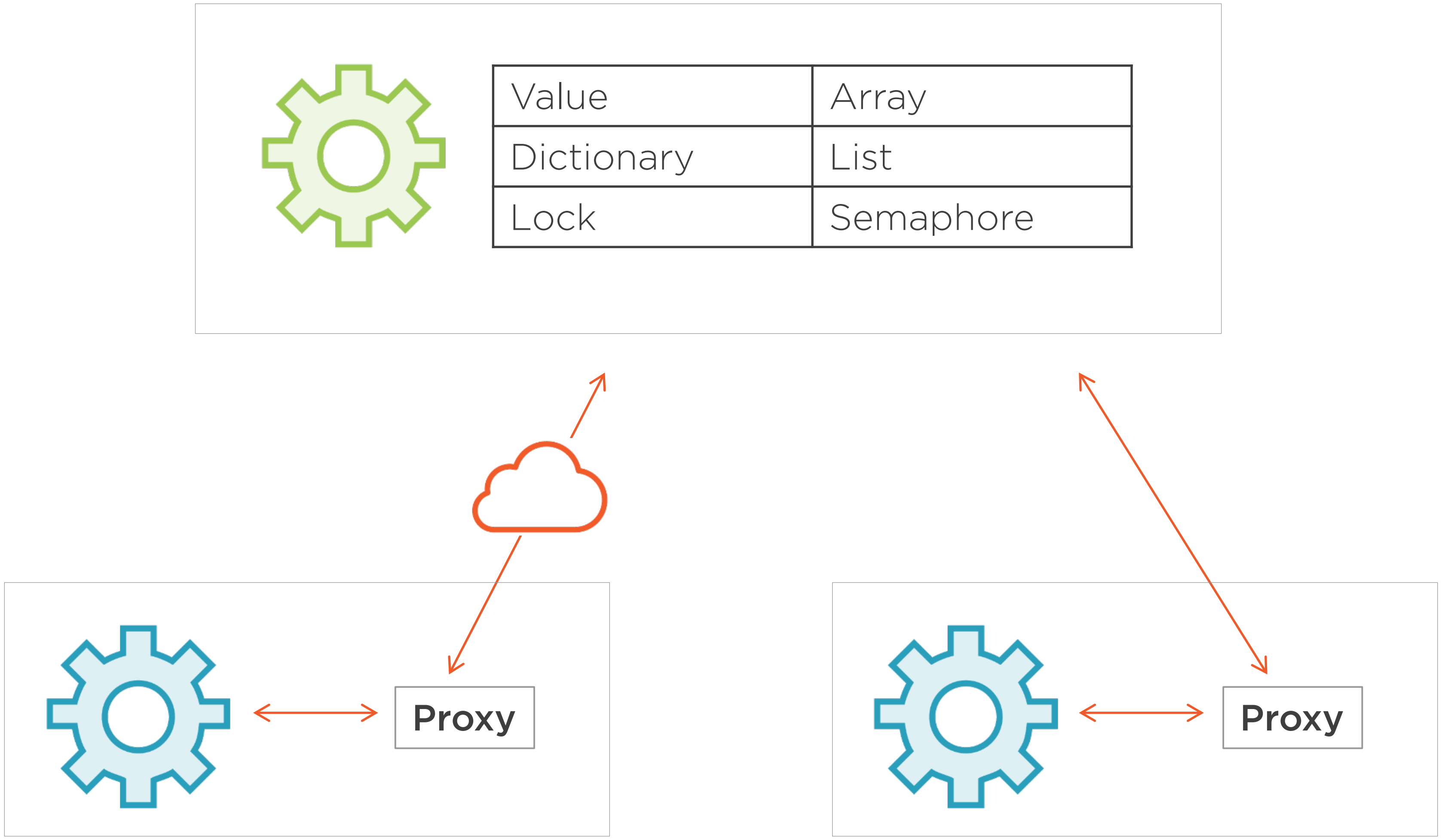


Value	Array
Dictionary	List
Lock	Semaphore

Proxy

Proxy

Manager



Value	Array
Dictionary	List
Lock	Semaphore

Manager

```
multiprocessing.Manager() # spins up a new process
```

Manager Shared Objects

Data Structures	Synchronization Mechanisms
Value	Lock
Array	RLock
List	BoundedSemaphore
Dict	Event
Namespace	Condition
Queue	Barrier

```
import multiprocessing

def do_work(dictionary, item):
    dictionary[item] = item ** 2

if __name__ == '__main__':
    mgr = multiprocessing.Manager()
    d = mgr.dict()
    jobs = [
        multiprocessing.Process(target=do_work, args=(d, i))
        for i in range(8)
    ]
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    print('Results:', d)
```

Demo

Shared State Demo

Process Synchronization

Process Synchronization

```
with self.resized_size.get_lock():
    self.resized_size += os.path.getsize(out_filepath)
```

Process Synchronization

threading.	multiprocessing.
Lock	Lock
RLock	RLock
Semaphore	Semaphore
BoundedSemaphore	BoundedSemaphore
Event	Event
Condition	Condition
Barrier	Barrier

Summary

Processes vs. Threads

Python Multiprocessing API

Process Pool

Inter-process Communication

Shared State Between Processes

Process Synchronization