Threading

Tim Ojo
@tim_ojo www.timojo.com

Introduction to Threads

Process

the execution context of a running program

Process

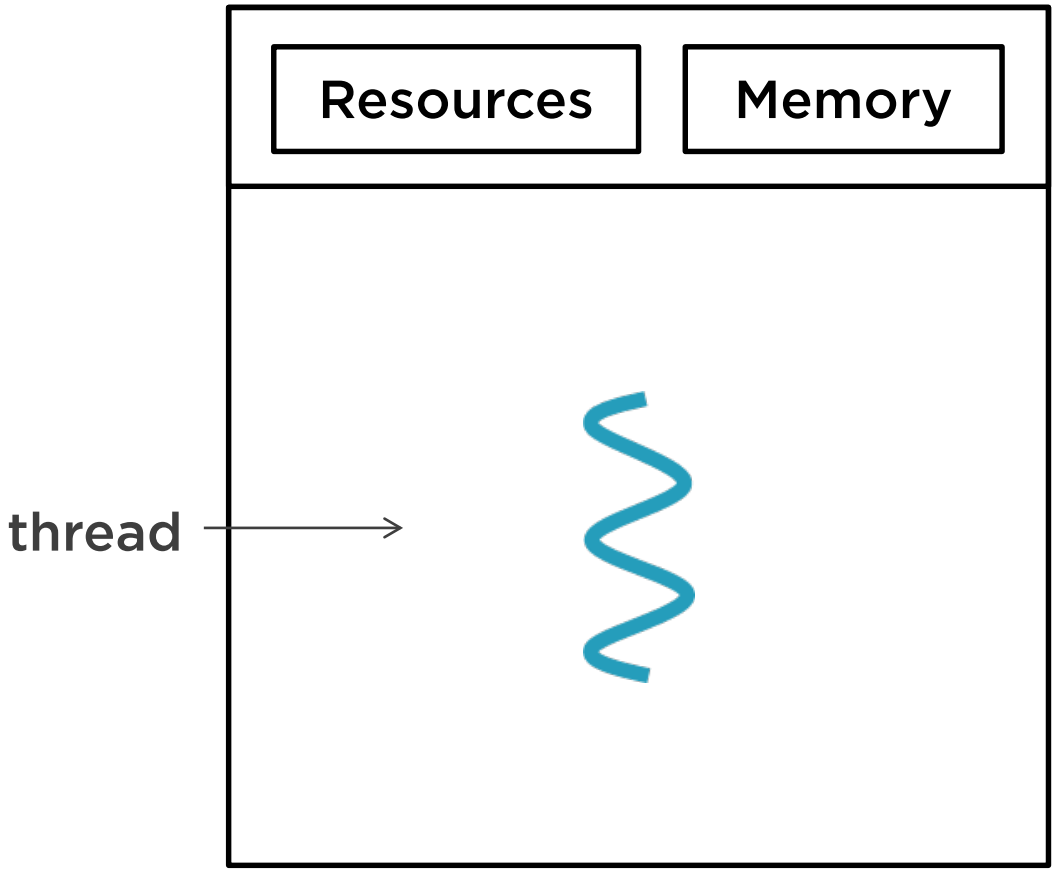
a running instance of a computer program

Thread

the smallest sequence of instructions that can be managed by the operating system

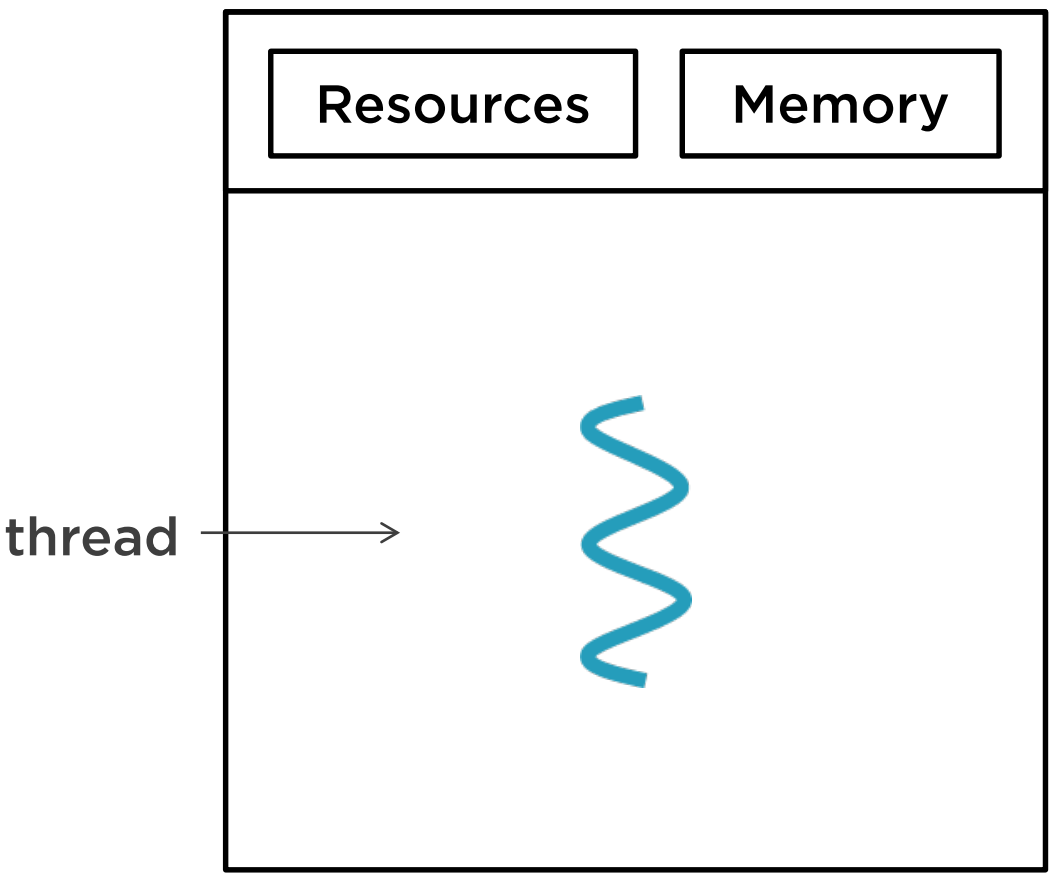
Thread

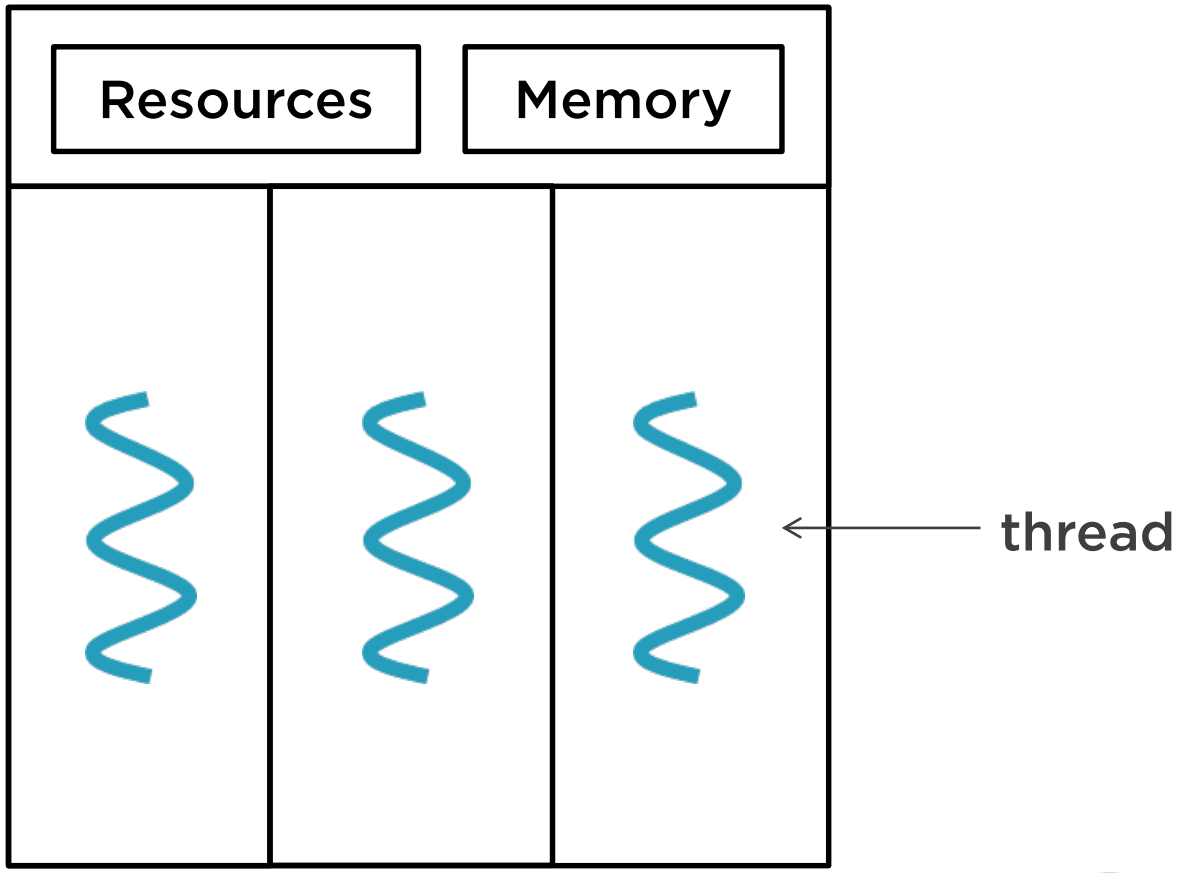
the smallest sequence of instructions that can be managed by the operating system



Thread

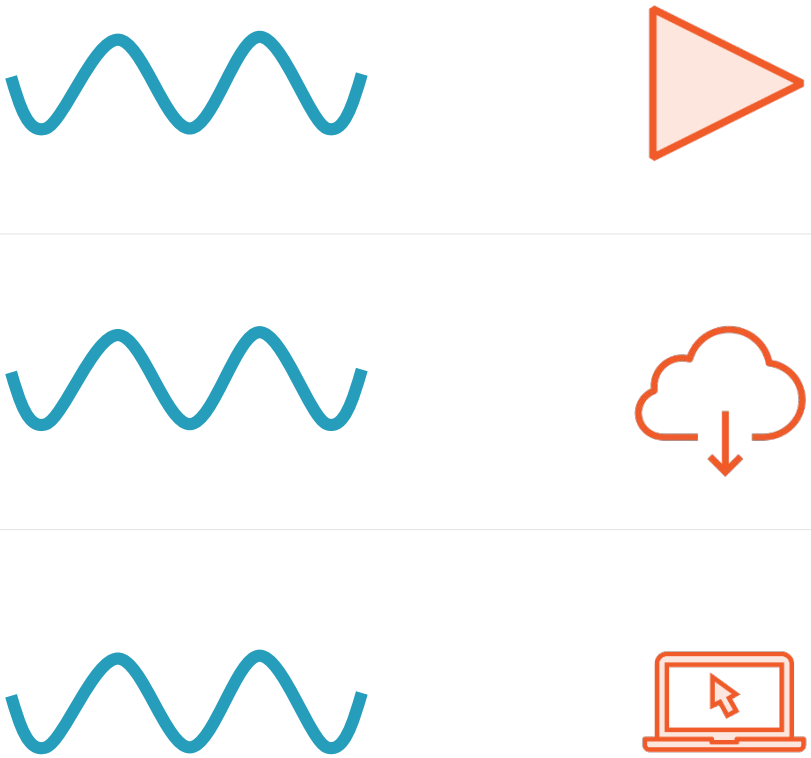
the smallest sequence of instructions that can be managed by the operating system



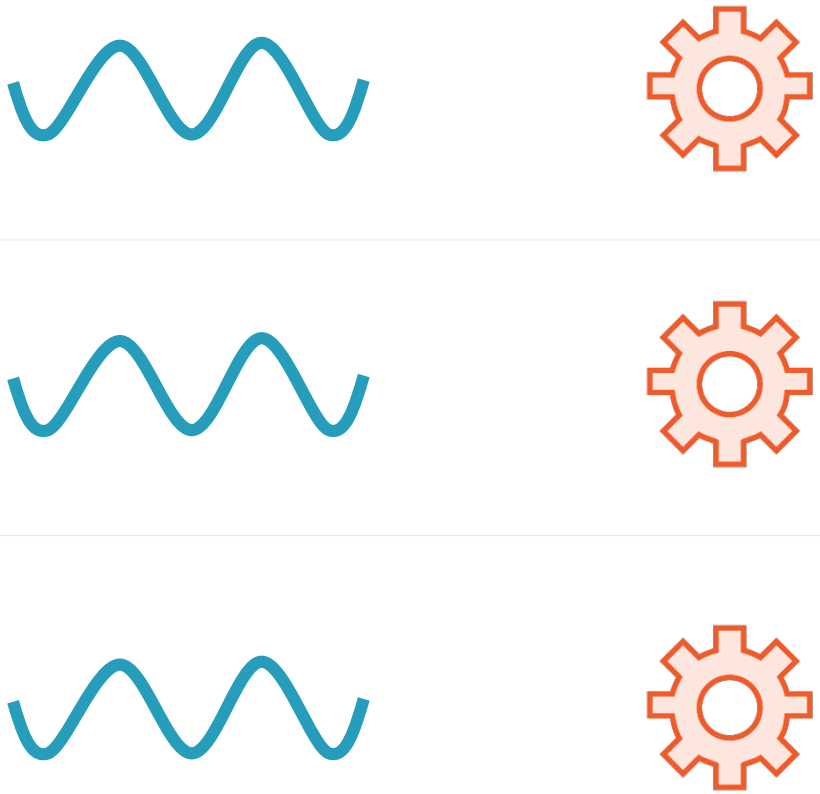


Running Threads

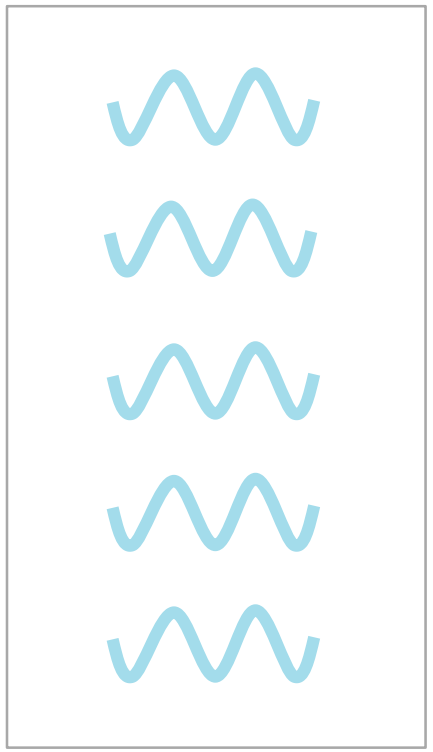
Running Threads



Running Threads



Thread Pool



Creating Threads in Python

```
1:  import threading

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

6:  val =  "text"
7:  t = threading.Thread(target=do_some_work, args=(val,))
8:  t.start()
9:  t.join()
```

```
1:  import threading

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

6:  val =  "text"
7:  t = threading.Thread(target=do_some_work, args=(val,))
8:  t.start()
9:  t.join()
```

```
1:  import threading

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

6:  val =  "text"
7:  t = threading.Thread(target=do_some_work, args=(val,))
8:  t.start()
9:  t.join()
```

```
1:  import threading

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

6:  val =  "text"
7:  t = threading.Thread(target=do_some_work, args=(val,))
8:  t.start()
9:  t.join()
```

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

```
1:  import threading

2:  class FibonacciThread(threading.Thread):
3:      def __init__(self, num):
4:          Thread.__init__(self)
5:          self.num = num

6:      def run(self):
7:          fib = [0] * (self.num + 1)
8:          fib[0] = 0
9:          fib[1] = 1
10:         for i in range(2, self.num+1):
11:             fib[i] = fib[i - 1] + fib[i - 2]
12:             print fib[self.num]

13: myFibTask1 = FibonacciThread(9)
14: myFibTask2 = FibonacciThread(12)

15: myFibTask1.start()
16: myFibTask2.start()

17: myFibTask1.join()
18: myFibTask2.join()
```

```
1:  import threading

2:  class FibonacciThread(threading.Thread):
3:      def __init__(self, num):
4:          Thread.__init__(self)
5:          self.num = num

6:      def run(self):
7:          fib = [0] * (self.num + 1)
8:          fib[0] = 0
9:          fib[1] = 1
10:         for i in range(2, self.num+1):
11:             fib[i] = fib[i - 1] + fib[i - 2]
12:             print fib[self.num]

13: myFibTask1 = FibonacciThread(9)
14: myFibTask2 = FibonacciThread(12)

15: myFibTask1.start()
16: myFibTask2.start()

17: myFibTask1.join()
18: myFibTask2.join()
```

```
1:  import threading

2:  class FibonacciThread(threading.Thread):
3:      def __init__(self, num):
4:          Thread.__init__(self)
5:          self.num = num

6:      def run(self):
7:          fib = [0] * (self.num + 1)
8:          fib[0] = 0
9:          fib[1] = 1
10:         for i in range(2, self.num+1):
11:             fib[i] = fib[i - 1] + fib[i - 2]
12:             print fib[self.num]

13: myFibTask1 = FibonacciThread(9)
14: myFibTask2 = FibonacciThread(12)

15: myFibTask1.start()
16: myFibTask2.start()

17: myFibTask1.join()
18: myFibTask2.join()
```

```
1:  import threading

2:  class FibonacciThread(threading.Thread):
3:      def __init__(self, num):
4:          Thread.__init__(self)
5:          self.num = num

6:      def run(self):
7:          fib = [0] * (self.num + 1)
8:          fib[0] = 0
9:          fib[1] = 1
10:         for i in range(2, self.num+1):
11:             fib[i] = fib[i - 1] + fib[i - 2]
12:             print fib[self.num]

13: myFibTask1 = FibonacciThread(9)
14: myFibTask2 = FibonacciThread(12)

15: myFibTask1.start()
16: myFibTask2.start()

17: myFibTask1.join()
18: myFibTask2.join()
```

```
1:  import threading

2:  class FibonacciThread(threading.Thread):
3:      def __init__(self, num):
4:          Thread.__init__(self)
5:          self.num = num

6:      def run(self):
7:          fib = [0] * (self.num + 1)
8:          fib[0] = 0
9:          fib[1] = 1
10:         for i in range(2, self.num+1):
11:             fib[i] = fib[i - 1] + fib[i - 2]
12:             print fib[self.num]

13: myFibTask1 = FibonacciThread(9)
14: myFibTask2 = FibonacciThread(12)

15: myFibTask1.start()
16: myFibTask2.start()

17: myFibTask1.join()
18: myFibTask2.join()
```

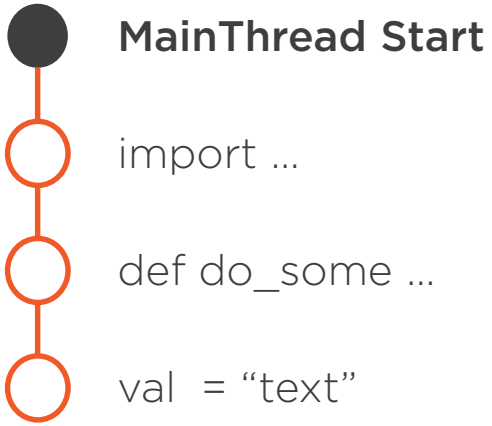
How Threads Work

```
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()
print(“done")
```

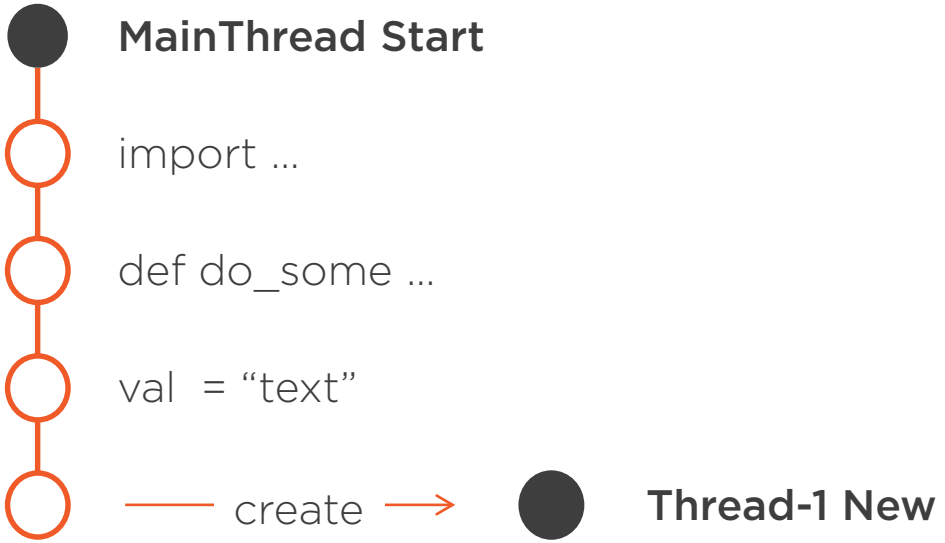
MainThread Start

```
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()
print(“done")
```

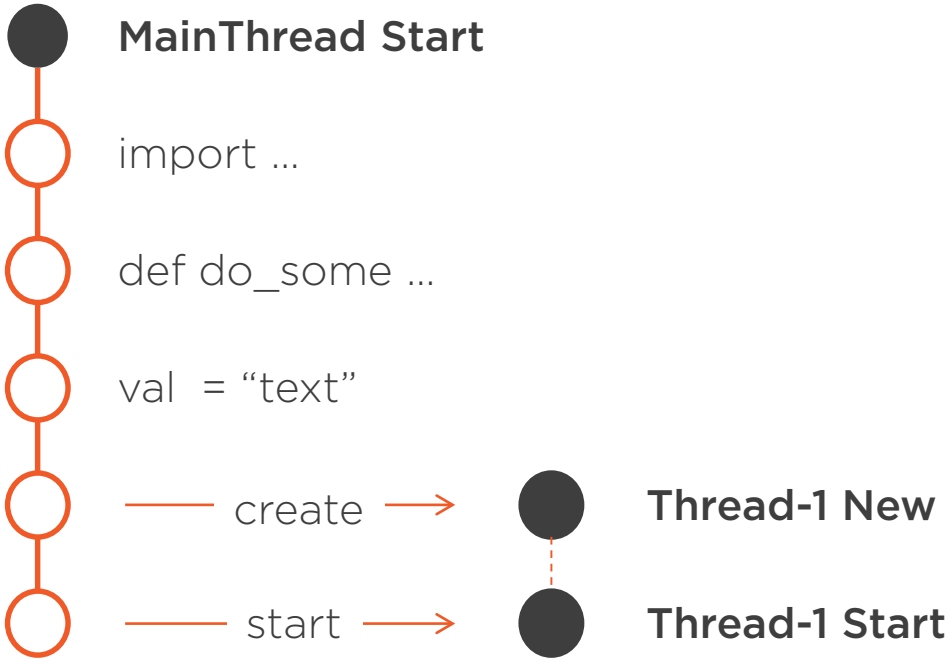
```
import threading

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()
print("done")
```



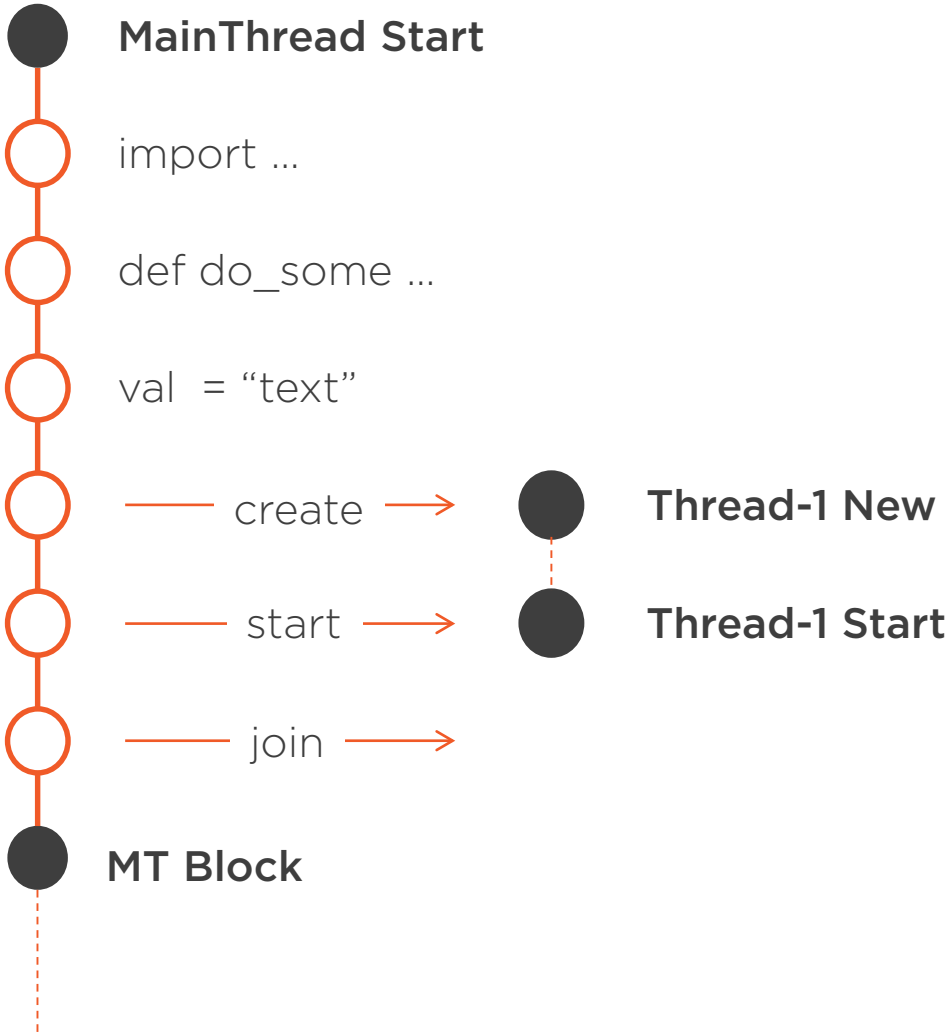
```
import threading

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()
print("done")
```



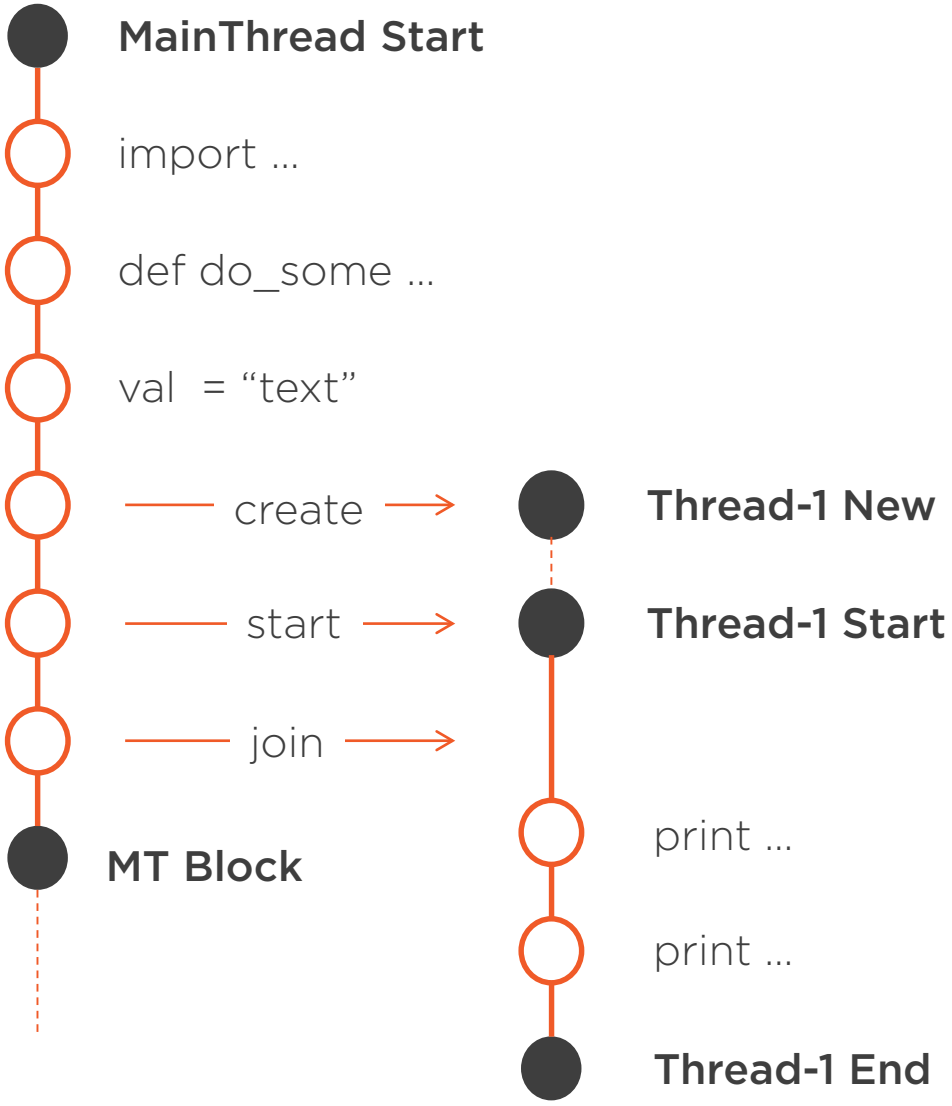
```
import threading

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()
print("done")
```



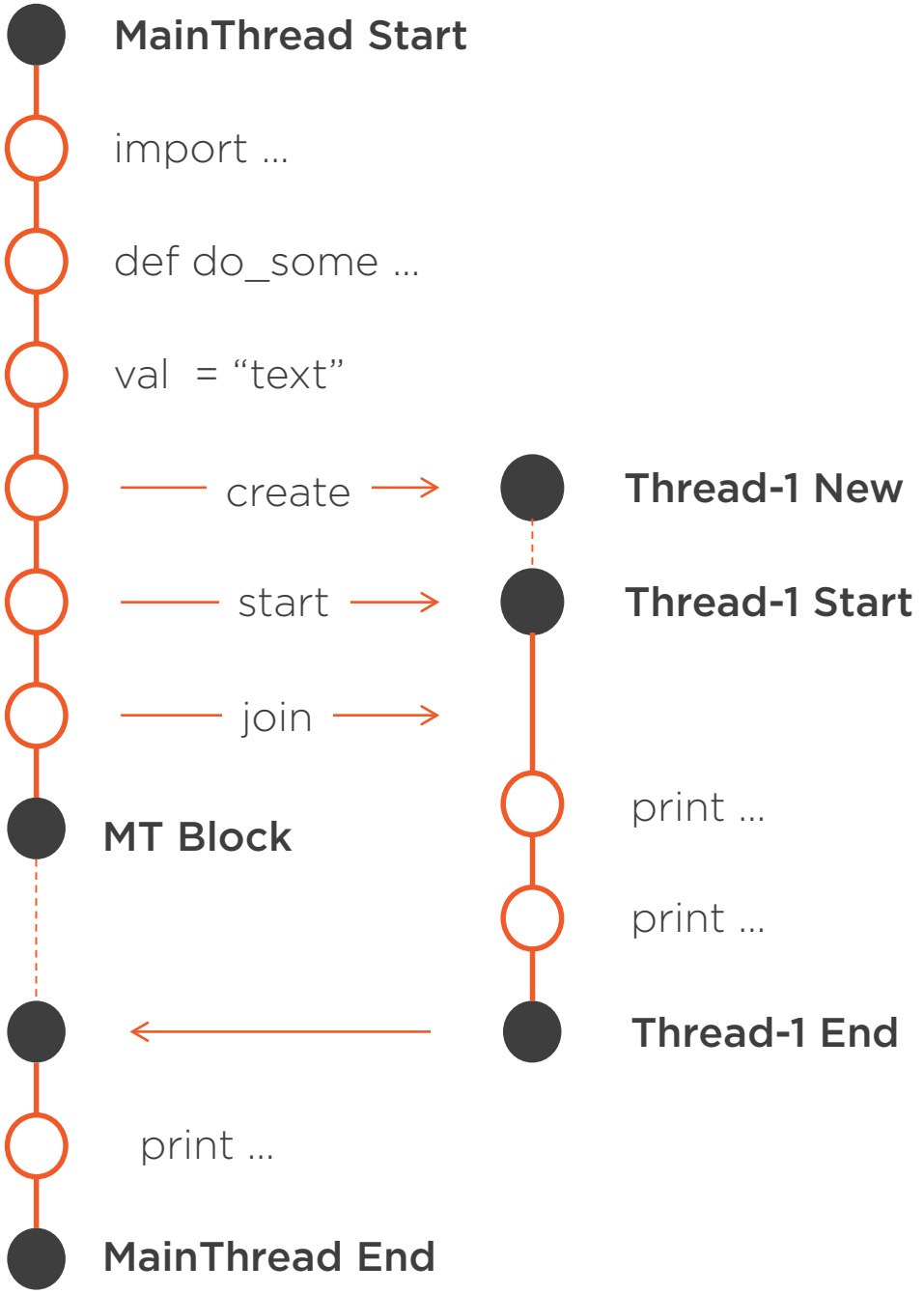
```
import threading

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()
print("done")
```



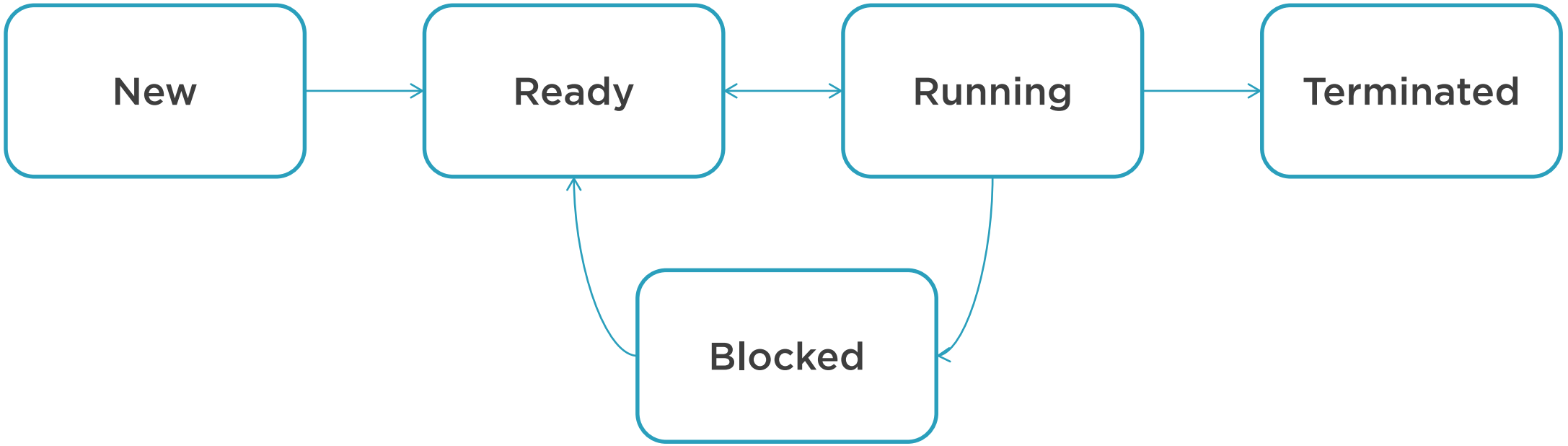
```
import threading

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()
print("done")
```

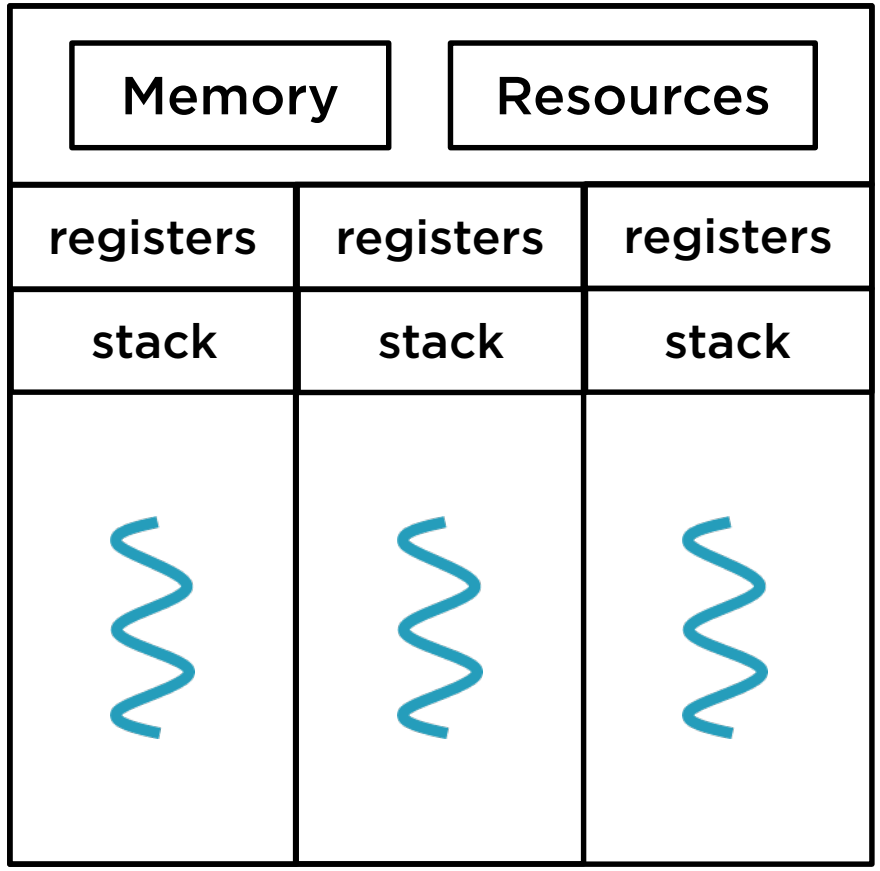


```
import threading

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()
print("done")
```



Thread Lifecycle




Memory
Resources
registers
registers
registers
stack
stack
stack

The Scheduler

an operating system module that selects the next jobs to be admitted into the system and the next process to run

Context Switch

the process of saving and restoring the state of a thread or process

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread 2 (withdraw 50) subtracts 50 from 0 resulting in -50

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread 2 (withdraw 50) subtracts 50 from 0 resulting in -50

Thread 2 (withdraw 50) stores resulting balance as -50 and exits

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread 2 (withdraw 50) subtracts 50 from 0 resulting in -50

Thread 2 (withdraw 50) stores resulting balance as -50 and exits

Thread 1 (deposit 100) gets switched back on the CPU

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread 2 (withdraw 50) subtracts 50 from 0 resulting in -50

Thread 2 (withdraw 50) stores resulting balance as -50 and exits

Thread 1 (deposit 100) gets switched back on the CPU

Thread 1 (deposit 100) adds 100 to 0 resulting in 100

Thread Interference

```
class BankAccount:
    def __init__(self):
        self.bal = 0

    def deposit(self, amt):
        balance = self.bal
        self.bal = balance + amt

    def withdraw(self, amt):
        balance = self.bal
        self.bal = balance – amt

b = BankAccount()
t1 = threading.Thread(target=b.deposit, args=(100,))
t2 = threading.Thread(target=b.withdraw, args=(50,))
t1.start()
t2.start()
```

Thread 1 (deposit 100) reads balance as 0

Thread 1 (deposit 100) gets suspended

Thread 2 (withdraw 50) reads balance as 0

Thread 2 (withdraw 50) subtracts 50 from 0 resulting in -50

Thread 2 (withdraw 50) stores resulting balance as -50 and exits

Thread 1 (deposit 100) gets switched back on the CPU

Thread 1 (deposit 100) adds 100 to 0 resulting in 100

Thread 1 (deposit 100) stores the resulting balance as 100 and exits

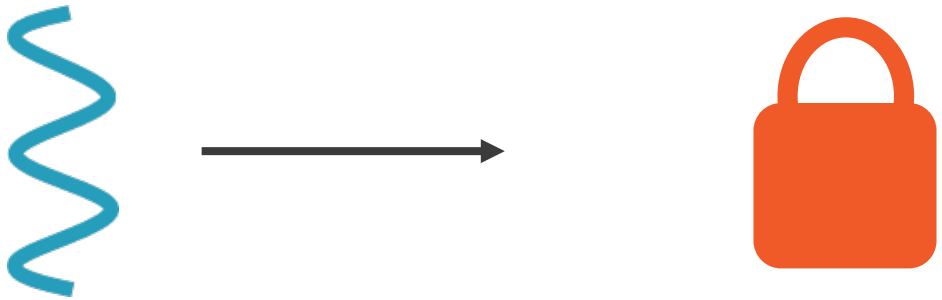
Thread Synchronization

threading.Lock

threading.Lock

threading.Lock

threading.Lock



threading.Lock

threading.Lock

```
lock = threading.Lock()

lock.acquire()
... access shared resource ...
lock.release()
```

```
lock = threading.Lock()

lock.acquire()
try:
    ... access shared resource ...
finally:
    lock.release()
```

```
lock = threading.Lock()

with lock:
        ... access shared resource ...
```

```
if lock.acquire(False):
    ... lock acquired, do stuff with lock
else:
    ... could not acquire lock, do other stuff
```

```
if lock.locked():
    ... do other stuff
else:
#    Note: there's no guarantee that by the time this line runs,
#    the lock is still free
    lock.acquire()
```

threading.RLock

```
lock = threading.Lock()
lock.acquire()
lock.acquire() # this call will block

lock = threading.RLock()
lock.acquire()
lock.acquire() # this call won't block
```

threading.Semaphore

```
semaphore = threading.Semaphore()

semaphore.acquire() # decrements the counter
... access the shared resource
semaphore.release() # increments the counter
```

```
num_permits = 3
semaphore = threading.BoundedSemaphore(num_permits)

semaphore.acquire() # decrements the counter
... up to 3 threads can access the shared resource at a time ...
semaphore.release() # increments the counter
```

threading.Event

```
event = threading.Event()

# a client thread can wait for the flag to be set
event.wait()   # blocks if flag is false

# a server thread can set or reset it
event.set()    # sets the flag to true
event.clear() # resets the flag to false
```

threading.Condition

threading.Condition

- acquire()
- release()

threading.Condition

- acquire()
- release()

- wait()
- notify()
- notify_all()

```
cond = Condition()

# Consume one item
cond.acquire()
while not an_item_is_available():
    cond.wait()
get_an_available_item()
cond.release()

# Produce one item
cond.acquire()
make_an_item_available()
cond.notify()
cond.release()
```

Inter-thread Communication Using Queues

Queue

- put(): Puts an item into the queue
- get(): Removes an item from the queue and returns it
- task_done(): Marks an item that was gotten from the queue as completed / processed
- join(): Blocks until all the items in the queue have been processed

```
from threading import Thread
from queue import Queue

def producer(queue):
    for i in range(10):
        item = make_an_item_available(i)
        queue.put(item)

def consumer(queue):
    while True:
        item = queue.get()
        # do something with the item
        queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread(target=producer, args=(queue,))
t2 = Thread(target=consumer, args=(queue,))
t1.start()
t2.start()
```

```
from threading import Thread
from queue import Queue

def producer(queue):
    for i in range(10):
        item = make_an_item_available(i)
        queue.put(item)

def consumer(queue):
    while True:
        item = queue.get()
        # do something with the item
        queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread(target=producer, args=(queue,))
t2 = Thread(target=consumer, args=(queue,))
t1.start()
t2.start()
```

```
from threading import Thread
from queue import Queue

def producer(queue):
    for i in range(10):
        item = make_an_item_available(i)
        queue.put(item)

def consumer(queue):
    while True:
        item = queue.get()
        # do something with the item
        queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread(target=producer, args=(queue,))
t2 = Thread(target=consumer, args=(queue,))
t1.start()
t2.start()
```

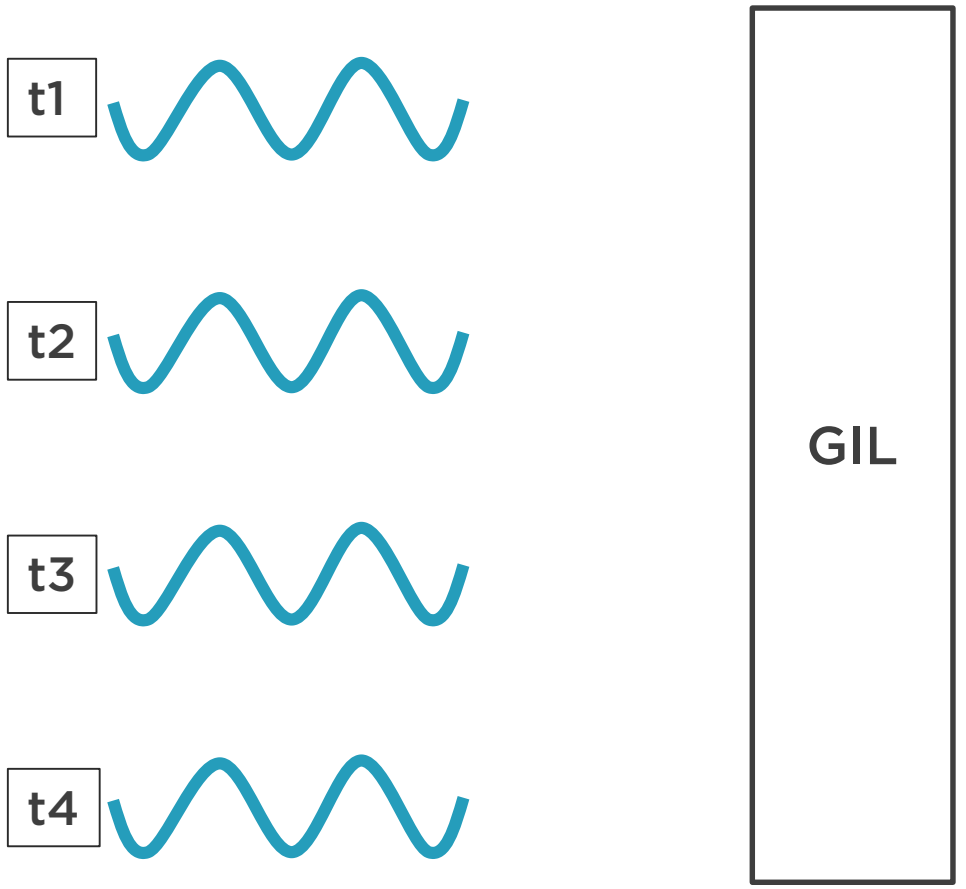
```
from threading import Thread
from queue import Queue

def producer(queue):
    for i in range(10):
        item = make_an_item_available(i)
        queue.put(item)

def consumer(queue):
    while True:
        item = queue.get()
        # do something with the item
        queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread(target=producer, args=(queue,))
t2 = Thread(target=consumer, args=(queue,))
t1.start()
t2.start()
```

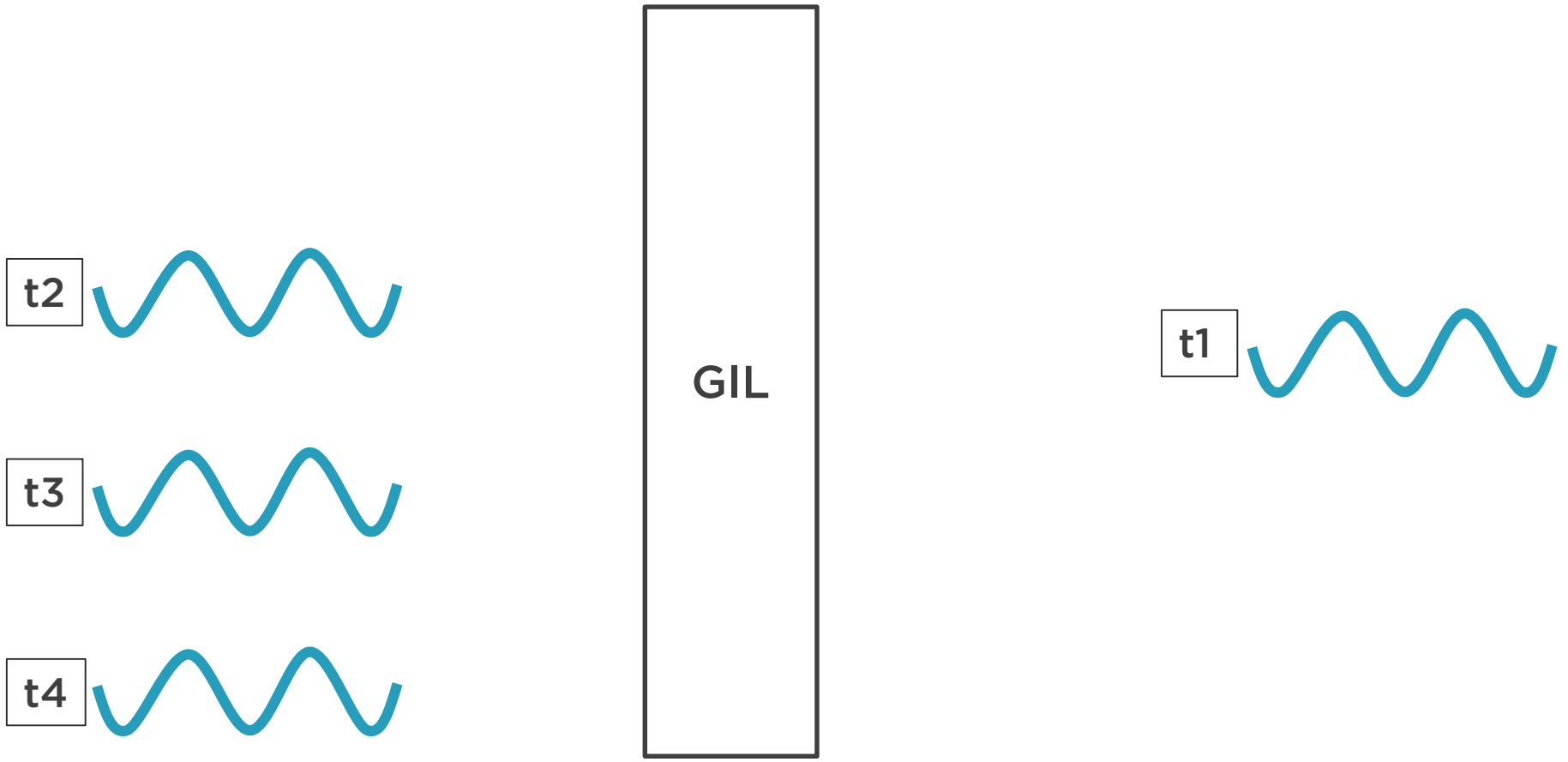
```
from threading import Thread
from queue import Queue

def producer(queue):
    for i in range(10):
        item = make_an_item_available(i)
        queue.put(item)

def consumer(queue):
    while True:
        item = queue.get()
        # do something with the item
        queue.task_done() # mark the item as done

queue = Queue()
t1 = Thread(target=producer, args=(queue,))
t2 = Thread(target=consumer, args=(queue,))
t1.start()
t2.start()
```

The Global Interpreter Lock

Global Interpreter Lock

a lock that prevents multiple native threads from executing Python code at the same time


t1
t2
t3
t4
GIL
Eligible Threads
Acquire Lock


t2
t3
t4
GIL
t1
Eligible Threads
Acquire Lock
Access Python Data Structures

GIL Workarounds

GIL-less Python Interpreters

- Jython
- IronPython

Python Multiprocessing

Summary

- Intro to threads in Python
- How threads work
- Thread synchronization
- Inter-thread communication using queues
- Global interpreter lock