Asynchronous programming. Threads

Table of content

Concept: Threads

What most programmers think of when they hear about "concurrent programming“?

- An independent task running inside a program
- Shares resources with the main program (memory, files, network connections, etc.)
- Has its own independent flow of execution (stack, current instruction, etc.)

Thread Basics

```
% python program.py
```

↓

statement
statement
...

↓

"main thread"

Program launch. Python loads a program and starts executing statements

Thread Basics

```
% python program.py
```

↓

```
statement
statement
...
```

↓

create thread(foo) ⇢ `def foo():`

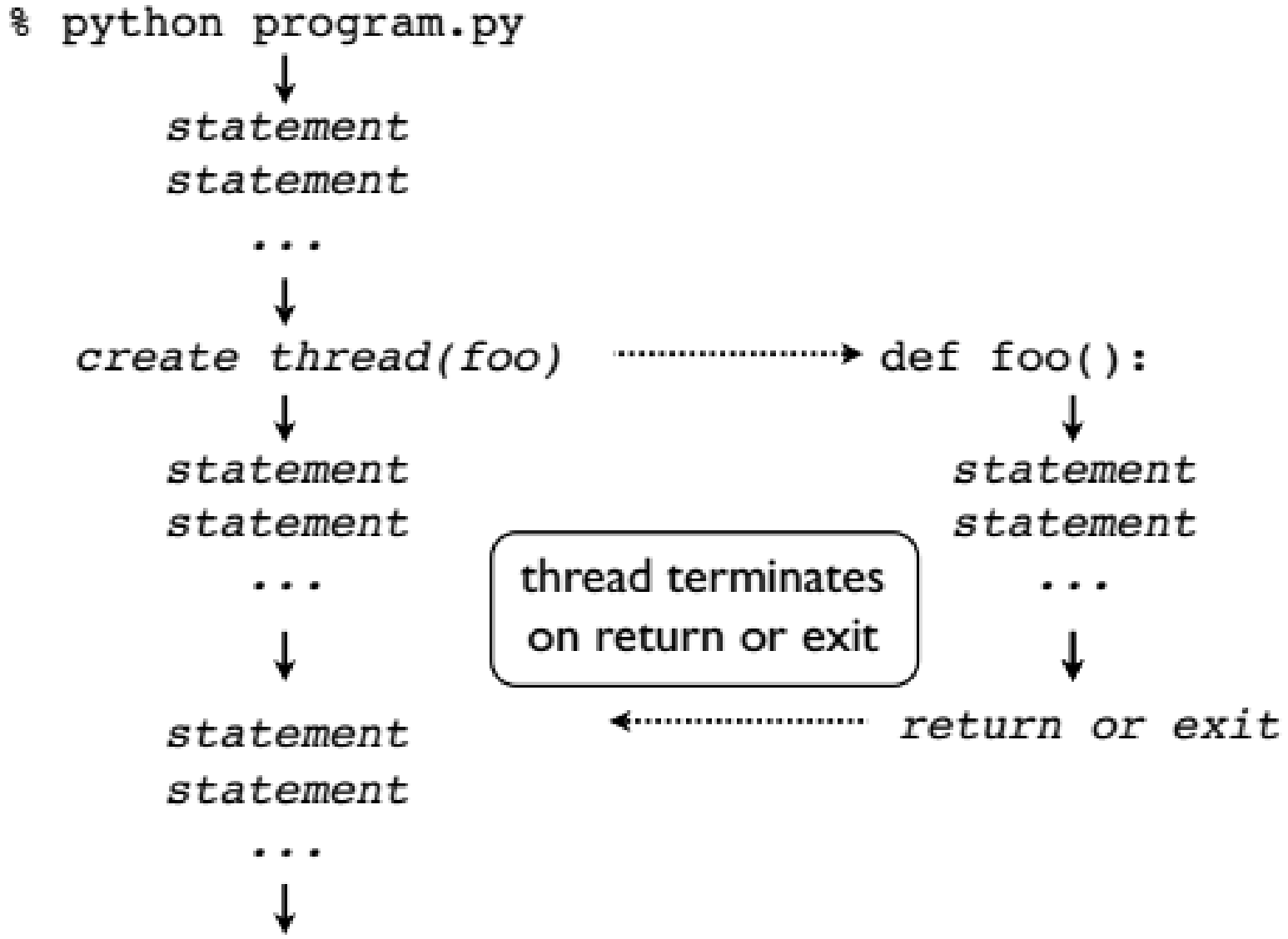
Creation of a thread.
Launches a function.

Thread Basics

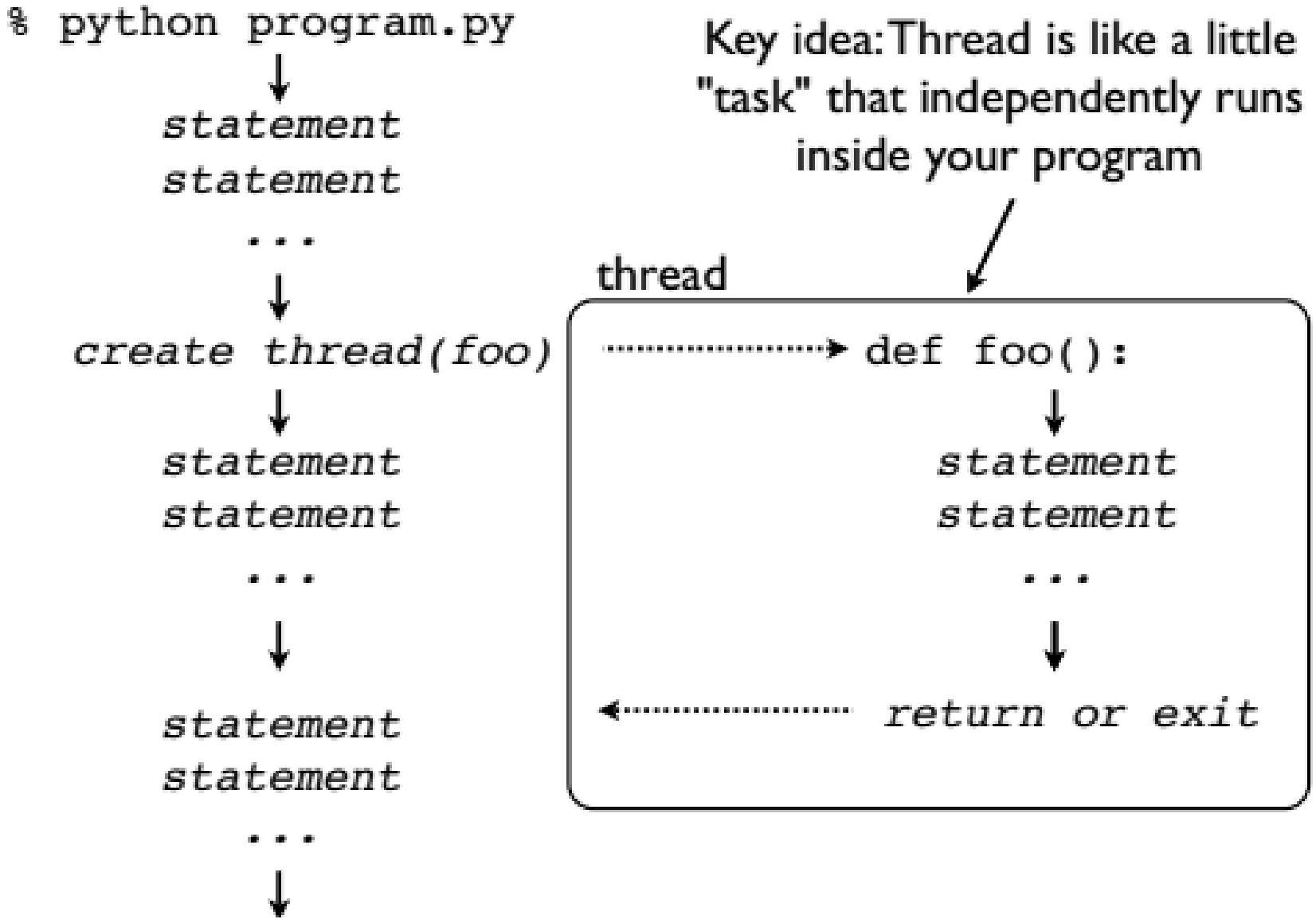
```
% python program.py
        ↓
    statement
    statement
        ...
        ↓
 create thread(foo)  ····· def foo():
        ↓                      ↓
    statement              statement
    statement              statement
        ...                    ...
        ↓                      ↓
```

Concurrent execution of statements

Thread Basics



Thread Basics



Threading Module

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

- You inherit from Thread and redefine run()

- Python threads are defined by a class

```
import time
import threading

class CountdownThread(threading.Thread):
    def __init__(self,count):
        threading.Thread.__init__(self)
        self.count = count
    def run(self):
        while self.count > 0:
            print "Counting down", self.count
            self.count -= 1
            time.sleep(5)
        return
```

This code executes in the thread

- You inherit from Thread and redefine run()

Threading Module

- To launch, create thread objects and call start()

```
t1 = CountdownThread(10)  # Create the thread object
t1.start()                # Launch the thread

t2 = CountdownThread(20)  # Create another thread
t2.start()                # Launch
```

- Threads execute until the run() method stops

Functions as threads

- Alternative method of launching threads

```
def countdown(count):
    while count > 0:
        print "Counting down", count
        count -= 1
        time.sleep(5)

t1 = threading.Thread(target=countdown,args=(10,))
t1.start()
```

- Creates a Thread object, but its run() method just calls the given function

Joining a Thread

- Once you start a thread, it runs independently
- Use t.join() to wait for a thread to exit

```
t.start()          # Launch a thread
...
# Do other work
...
# Wait for thread to finish
t.join()           # Waits for thread t to exit
```

- This only works from *other* threads
- A thread can't join itself

Daemonic Thread

- If a thread runs forever, make it "daemonic"

```
t.daemon = True
t.setDaemon(True)
```

- If you don't do this, the interpreter will lock when the main thread exits---waiting for the thread to terminate (which never happens)
- Normally you use this for background tasks

Interlude

- Creating threads is really easy
- You can create thousands of them if you want
- Programming with threads is hard
- <u>Really hard</u>

***Q**: Why did the multithreaded chicken cross the road?*

***A:** to To other side. get the*

-- Jason Whittington

Access to Shared Data

- Threads share all of the data in your program
- Thread scheduling is non-deterministic
- Operations often take several steps and might be interrupted mid-stream (non-atomic)
- Thus, access to any kind of shared data is also non-deterministic (which is a really good way to have your head explode)

Accessing Shared Data

- Consider a shared object

```
x = 0
```

- And two threads that modify it

```
Thread-1                    Thread-2
--------                    --------
...                         ...
x = x + 1                   x = x - 1
...                         ...
```

- It's possible that the resulting value will be unpredictably corrupted

Accessing Shared Data

- The two threads

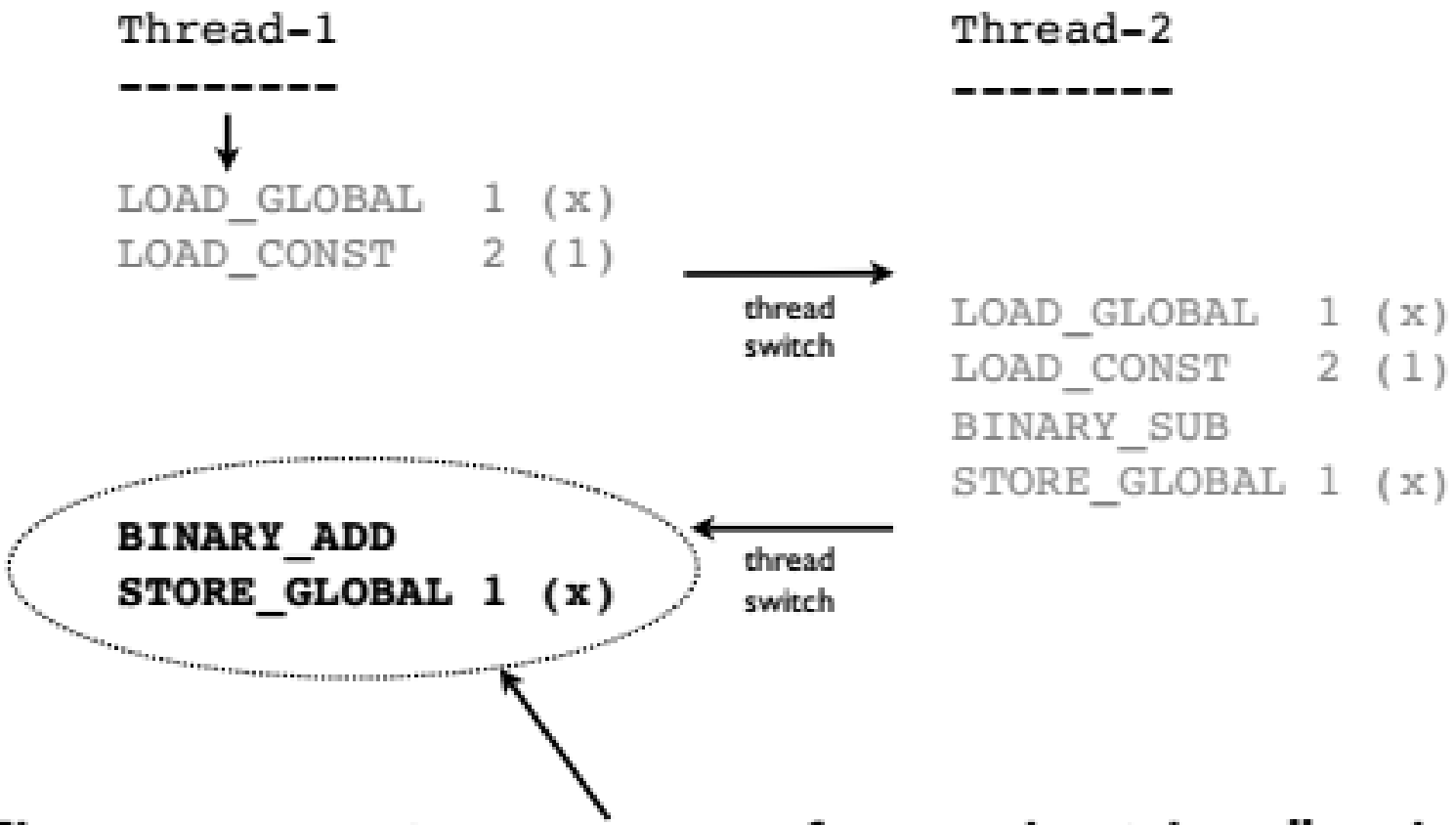
```
Thread-1                    Thread-2
--------                    --------
...                         ...
x = x + 1                   x = x - 1
...                         ...
```

- Low level interpreter execution

```
Thread-1                    Thread-2
--------                    --------
    ↓
LOAD_GLOBAL  1 (x)
LOAD_CONST   2 (1)
                      ──────→
                      thread
                      switch
                            LOAD_GLOBAL  1 (x)
                            LOAD_CONST   2 (1)
                            BINARY_SUB
                            STORE_GLOBAL 1 (x)
BINARY_ADD            ←──────
STORE_GLOBAL 1 (x)    thread
                      switch
```

Accessing Shared Data

- Low level interpreter code



These operations get performed with a "stale" value of x. The computation in Thread-2 is lost.

Accessing Shared Data

- Is this actually a real concern?

```
x = 0            # A shared value
def foo():
     global x
     for i in xrange(100000000): x += 1

def bar():
     global x
     for i in xrange(100000000): x -= 1

t1 = threading.Thread(target=foo)
t2 = threading.Thread(target=bar)
t1.start(); t2.start()
t1.join(); t2.join()   # Wait for completion
print x                # Expected result is 0
```

- Yes, the print produces a random nonsensical value each time (e.g., -83412 or 1627732)

Race Conditions

- The corruption of shared data due to thread scheduling is often known as a "race condition.“
- It's often quite diabolical--a program may produce slightly different results each time it runs (even though you aren't using any random numbers)
- Or it may just flake out mysteriously once every two weeks

Thread Synchronization

- Identifying and fixing a race condition will make you a better programmer (e.g., it "builds character")
- However, you'll probably never get that month of your life back...
- To fix : You have to synchronize threads