**Lamda**

A lambda function is a small anonymous function.

**lambda arguments: expression**

* This function can have any number of arguments but only one expression, which is evaluated and returned.
* One is free to use lambda functions wherever function objects are required.
* You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.

**Threading**

**Multitasking**

* Multithreading
* Multiprocessing

For simplicity, you can assume that a thread is simply a subset of a process!

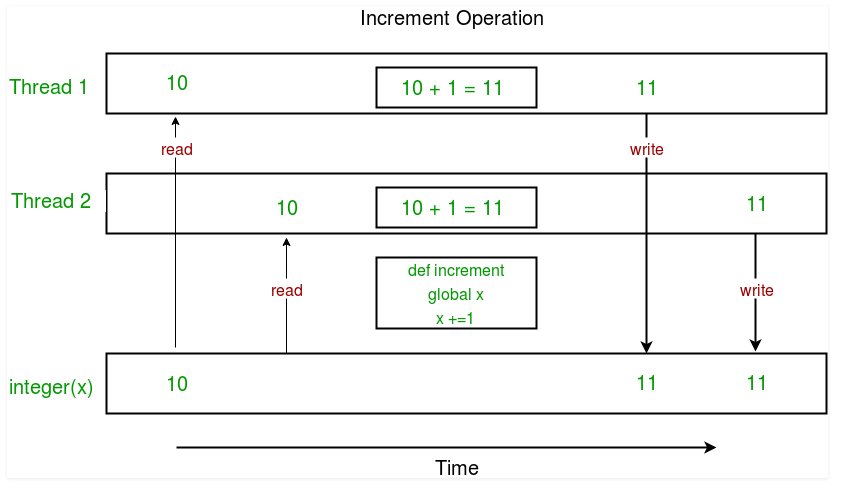
**Threads**

* Daemon
* Non Daemon Threads

**Synchronization between threads**

Thread synchronization is defined as a mechanism which ensures that two or more concurrent threads do not simultaneously execute some particular program segment known as **critical section**.

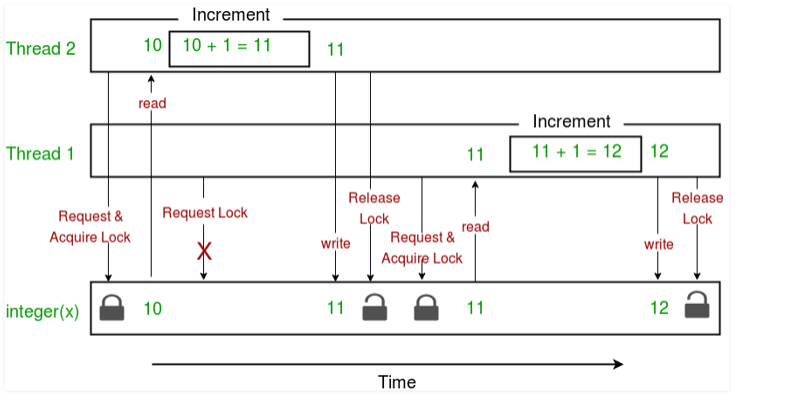
Critical section refers to the parts of the program where the shared resource is accessed.



Concurrent accesses to shared resource can lead to **race condition**.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. As a result, the values of variables may be unpredictable and vary depending on the timings of context switches of the processes.

**threading** module provides a **Lock** class to deal with the race conditions. Lock is implemented using a **Semaphore** (which is object provided by the Operating System.)



*In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed as****context switching****. In context switching, the state of a thread is saved and state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place. Context switching takes place so frequently that all the threads appear to be running parallelly.*

|  |
| --- |
| Threading: |
|  | - A new thread is spawned within the existing process |
|  | - Starting a thread is faster than starting a process |
|  | - Memory is shared between all threads |
|  | - Mutexes often necessary to control access to shared data |
|  | - One GIL (Global Interpreter Lock) for all threads |

|  |
| --- |
| Multiprocessing: |
|  | - A new process is started independent from the first process |
|  | - Starting a process is slower than starting a thread |
|  | - Memory is not shared between processes |
|  | - Mutexes not necessary (unless threading in the new process) |
|  | - One GIL (Global Interpreter Lock) for each process |

The Python Global Interpreter Lock or [GIL](https://wiki.python.org/moin/GlobalInterpreterLock), in simple words, is a mutex (or a lock) that allows only one thread to hold the control of the Python interpreter.

## [What Should You Use?](https://timber.io/blog/multiprocessing-vs-multithreading-in-python-what-you-need-to-know/" \l "what-should-you-use-)

If your code has a lot of I/O or Network usage:

* Multithreading is your best bet because of its low overhead

If you have a GUI

* Multithreading so your UI thread doesn't get locked up

If your code is CPU bound

* You should use multiprocessing (if your machine has multiple cores)

GIL Reference: <http://www.dabeaz.com/GIL/gilvis/linuxtwothread.html>

J**ava**S**cript**O**bject**N**otation**

The community at large adopted JSON because it’s easy for both humans and machines to create and understand.

{

"firstName": "Jane",

"lastName": "Doe",

"hobbies": ["running", "sky diving", "singing"],

"age": 35,

"children": [

{

"firstName": "Alice",

"age": 6

},

{

"firstName": "Bob",

"age": 8

}

]

}

The process of encoding JSON is usually called **serialization**

Naturally, **deserialization** is the reciprocal process of decoding data that has been stored or delivered in the JSON standard.

Think of it like this: encoding is for writing data to disk, while decoding is for reading data into memory.

Simple Python objects are translated to JSON according to a fairly intuitive conversion.

| **Python** | **JSON** |
| --- | --- |
| dict | object |
| list, tuple | array |
| str | string |
| int, long, float | number |
| True | true |
| False | false |
| None | null |

| **JSON** | **Python** |
| --- | --- |
| object | dict |
| array | list |
| string | str |
| number (int) | int |
| number (real) | float |
| true | True |
| false | False |
| null | None |

>>> blackjack\_hand = (8, "Q")

>>> encoded\_hand = json.dumps(blackjack\_hand)

>>> decoded\_hand = json.loads(encoded\_hand)

>>> blackjack\_hand == decoded\_hand

False

>>> type(blackjack\_hand)

<class 'tuple'>

>>> type(decoded\_hand)

<class 'list'>

>>> blackjack\_hand == tuple(decoded\_hand)

True

**XML**

|  |  |  |
| --- | --- | --- |
| 1) | JSON stands for JavaScript Object Notation. | XML stands for eXtensible Markup Language. |
| 2) | JSON is simple to read and write. | XML is less simple than JSON. |
| 3) | JSON is easy to learn. | XML is less easy than JSON. |
| 4) | JSON is data-oriented. | XML is document-oriented. |
| 5) | JSON doesn't provide display capabilities. | XML provides the capability to display data because it is a markup language. |
| 6) | JSON supports array. | XML doesn't support array. |
| 7) | JSON is less secured than XML. | XML is more secured. |
| 8) | JSON files are more human readable than XML. | XML files are less human readable. |
| 9) | JSON supports only text and number data type. | XML support many data types such as text, number, images, charts, graphs etc. Moreover, XML offeres options for transferring the format or structure of the data with actual data. |