threading管理并发操作

概念:

什么是进程?

计算机程序是磁盘中可执行的二进制数据，他们被读取到内存中，被操作系统调用的时候就开始了他们的生命周期；进程是程序的一次执行，每个进程都有自己的地址空间 内存 数据栈；以及其他记录其运行轨迹的辅助数据．因为假设我们用的电脑是单核的，cpu同时只能执行一个进程．当程序处于I/O阻塞的时候，CPU如果和程序一起等待，那就太浪费了，所以此时cpu会去执行其他的程序，就涉及到程序的切换，切换前要保存上一个程序运行的状态才能恢复，所以就需要有个东西来记录这些切换了的程序的状态．进程就是一个程序在一个数据集上的一次动态执行过程；进程由程序 数据集 进程控制块三部分组成；程序用来描述进程哪些功能以及如何完成，数据集是程序执行过程中所使用的资源，进程控制块用来保存程序运行的状态．操作系统管理在上运行的所有进程 为这些进程公平的分配工作时间；因为各个进程都有自己的内存空间和数据栈，所以进程之间的通信需要使用特殊的方式来实现IPC(进程之间通信 inter-process Communication 比如queue socket等)所以进程间是不能直接共享信息的．

什么是线程?

线程跟进程有些类似，多个不同的是线程运行再同一个进程中，共享相同的运行环境的；可以把它们想象称为主进程中并行运行的多个迷你进程，一个进程中可以开多个线程，因为一个程序中，线程共享一套数据，如果都做成进程，每个进程独占一块内存，那这套数据就要复制好几份给每个程序，不合理，所以有了线程.线程的出现突破一个进程只能干一样事的缺陷，使到进程内并发成为可能.假设，一个文本程序需要接受键盘输入，将内容显示在屏幕上,还需要保存信息到硬盘中．若只有一个进程，势必造成同一时间只能干一样事的尴尬(当保存时 就不能通过键盘输入)．若有多个进程，每个进程负责一个任务，进程A负责接收键盘输入的任务，进程B负责将内容显示在屏幕上的任务，进程C负责保存内容到硬盘中的任务．这里进程A，B，C间的协作涉及到了进程通信问题，而且有共同都需要拥有的东西—文本内容；进程不停的切换造成性能上的损失，若有一种机制可以使任务A，B，C共享资源，这样上下文切换所需要保存和恢复的内容就少了，同时又可以减少通信所带来的性能损耗，这种机制就是线程．线程有开始 顺序执行 和结束三部分，线程有一个自己的指令指针，记录自己运行到了什么地方；线程的运行可能被抢占或暂时被挂起让其他线程运行，这叫做让步；一个进程中的每个线程之间共享同一片数据空间，所以线程之间可以比进程之间更方便的共享数据以及相互通讯．线程都是并发执行的，正是由于这种并行和数据共享机制使得多个任务合作变为可能，实际上在单CPU的系统中，真正并发是不可能的，每个线程会被安排成每次只运行一小会，然后把CPU让出来，让其他线程去运行．

什么是并发和并行?

并发：是指系统具有处理多个任务(动作)的能力(CPU通过切换来完成并发)，并发是指一个处理器同时处理多个任务。
并行：是指系统具有同时处理多个任务(动作)的能力，并行是指多个处理器或者是多核的处理器同时处理多个不同的任务．

什么是协程?

协程是一种用户态的轻量级线程，协程的调度完全由用户控制；协程拥有自己的寄存器上下文，协程调度切换时，将寄存器上下文保存到其他地方，在切回来的时候，恢复先前保存的寄存器上下文．协程的执行效率非常高，因为子程序切换不是线程切换，而是由程序自身控制；因此，没有线程切换的开销，和多线程比线程数量越多，协程的性能优势就越明显．协程不需要多线程的锁机制，在协程中控制共享资源不需加锁，因为协程本身就是工作在同一个线程中的．

什么是全局解释器锁(GIL)?
Python代码的执行由Python虚拟机(也叫解释器主循环)来控制，Python再设计之初就考虑到要在主循环中同时只有一个线程在执行，就像单CPU的系统中运行多个进程那样，内存中可以存放多个程序，但是任意时刻只有一个程序在CPU中运行．同样的虽然Python解释器可以运行多个线程，但是在任意时刻只有一个线程在解释器中运行．(这也是说Python的多线程是比较鸡肋的一个原因，是因为Python的多线程只能应用在特定的场景)．

对Python虚拟机的访问由全局解释器锁GIL来控制，正因为有这个锁所以才能保证同一个时刻只能有一个线程在运行．在多线程环境中，Python虚拟机按照以下方式来执行：

1.设置 GIL
2.切换到一个线程去运行
3.运行指定数量字节码的指令
4.把线程设置为睡眠状态
5.解锁GIL
6.再次重复以上的所有步骤

什么是I/O密集型与计算密集型?

对于所有面向I/O的程序来说，GIL会在这个I/O调用之前被释放，以允许其他的线程在当前这个线程的I/O等待时运行．如果某线程并未使用很多I/O操作，那么他会在自己的时间片内移植占用处理器和GIL，所以综上I/O密集型的Python程序比计算密集型的Python程序能更充分的利用多线程环境的好处(可能会难以理解，但是很重要我们会通过实例演示)．

什么是I/O密集型?
IO密集型：涉及到网络、磁盘IO的任务都是IO密集型任务，这类任务的特点是CPU消耗很少，任务的大部分时间都在等待IO操作完成(因为IO的速度远远低于CPU和内存的速度)．比如一个HTTP请求，丢一个请求过去之后其余工作就是等服务端响应了．

什么是计算密集型?
计算密集型任务的特点是要进行大量的计算，消耗CPU资源，这种计算密集型任务虽然也可以用多任务完成，但是任务越多，花在任务切换的时间就越多，CPU执行任务的效率就越低．所以，要最高效地利用CPU，计算密集型任务同时进行的数量应当等于CPU的核心数．在使用多线程处理日志时，同一时刻Python解释器中只能运行一个线程，那么就会存在读一个文件一段时间，然后锁定切换到下一个文件继续读一部分，这样将会有时间损耗在线程切换的工作中，所以通过多线程的方式去读取文件，并不能提升文本处理效率．

什么是同步与异步?

同步方法：调用一旦开始，调用者必须等到方法调用返回后，才能继续后续的行为．
异步方法：指发送一个请求，不需要等待返回，随时可以再发送下一个请求，即不需要等待．

比如我们去购物，如果你去商场实体店买一台空调，当你到了商场看中了一款空调，你就向售货员下单．售货员去仓库帮你调配物品， 这天你热的实在不行了，就催着商家赶紧给你配送，于是你就等在商场里候着他们，直到商家把你和空调一起送回家，一次愉快的购物就结束了，这就是同步调用．另外一个场景，如果我们在网上订购了一台空调，当你完成网上支付的时候，对你来说购物过程已经结束了，虽然空调还没有送到家，但是你的任务都已经完成了，商家接到你的订单后，就会加紧安排送货，当然这一切已经跟你无关了，你已经支付完成，想什么就能去干什么了，出去溜达几圈都不成问题，等送货上门的时候，接到商家电话，回家一趟签收即可，这就是异步调用．

1.1 线程的创建与启动

In [2]:
```
import threading

help(threading.Thread.__init__)
```

```
Help on function __init__ in module threading:

__init__(self, group=None, target=None, name=None, args=(), kwargs=None, *, daemon=None)
    This constructor should always be called with keyword arguments. Arguments are:

    *group* should be None; reserved for future extension when a ThreadGroup
    class is implemented.

    *target* is the callable object to be invoked by the run()
    method. Defaults to None, meaning nothing is called.

    *name* is the thread name. By default, a unique name is constructed of
    the form "Thread-N" where N is a small decimal number.

    *args* is the argument tuple for the target invocation. Defaults to ().

    *kwargs* is a dictionary of keyword arguments for the target
    invocation. Defaults to {}.

    If a subclass overrides the constructor, it must make sure to invoke
    the base class constructor (Thread.__init__()) before doing anything
    else to the thread.
```

In [12]:
```
import time
import random
def f1(x):
    current = threading.current_thread().getName()
    print('Task start, thread is %s value is %s' %(current, x))
    time.sleep(random.randint(1, 3))
    print('Task End,  thread is %s value is %s' %(current, x))

task = []
for j in range(5):
    t = threading.Thread(target=f1, args=(j, ), name="Mythread-%s" %j)
    task.append(t)

for j in task:
    j.start()
```

```
Task start, thread is Mythread-0 value is 0
Task start, thread is Mythread-1 value is 1Task start, thread is Mythread-2 value is 2
Task start, thread is Mythread-3 value is 3

Task start, thread is Mythread-4 value is 4
Task End,  thread is Mythread-0 value is 0
Task End,  thread is Mythread-3 value is 3
Task End,  thread is Mythread-2 value is 2
Task End,  thread is Mythread-1 value is 1Task End,  thread is Mythread-4 value is 4
```

1.1.2 派生线程
线程开始工作时要完成一些基本初始化, 然后调用run()方法, 派生线程就是创建Thread的子类, 需要覆盖run()来完成所需的工作.

示例代码:

In [1]:
```
import threading
import time

class Worker(threading.Thread):
    def run(self):
        time.sleep(1)
        print('This is %s' %threading.current_thread().getName())

if __name__ == '__main__':
    beg  = time.time()
    jobs = []
    for j in range(5):
        w = Worker()
        jobs.append(w)
    for j in jobs:
        j.start()
    for j in jobs:
        j.join()
    end = time.time()
    print('Cost time %s' %(end-beg))
```

```
This is Thread-5This is Thread-4
This is Thread-7
This is Thread-6
This is Thread-8

Cost time 1.0068747997283936
```

1.2 确定当前线程

每个Thread实例都有一个名称, 它有一个默认值, 可以在创建线程时改变. 如果服务器的进程由处理不同操作的多个线程组成, 在这样的业务中, 对线程命名就很有用.

In [10]:
```
#改变线程的名字
t = threading.Thread(target=f1, args=(j, ), name="Mythread-%s" %j)
```

In [11]:
```
#获取线程的名字
current = threading.current_thread().getName()
```

1.3 守护与非守护线程
到目前为止, 我们的多线程代码都隐含的等待所有子线程的工作都完成后主线程才退出, 因为默认情况下的子线程都是非守护线程, 默认情况下setDaemon()为false. 如果你设置一个线程为守护线程, 就表示你在说这个线程是不重要的, 在主线程退出的时候, 不用等待这个线程(守护线程)退出. 那就设置这些线程的daemon属性为true. 即在线程开始(thread.start())之前, 调用setDeamon()函数, 设定线程的daemon标志, (thread.setDaemon(True))就表示这个线程"不重要".

守护线程: 主线程不会等待守护线程中的任务完成并退出后才退出.
非守护线程: 主线程需等待所有的子线程任务完成并退出后才退出.

In [22]:
```
#示例代码: 守护线程
import threading
import time

def f1(sec):
    current = threading.current_thread().getName()
    print('Task start, thread is %s value is %s' %(current, sec))
    time.sleep(sec)
    print('Task End,  thread is %s value is %s' %(current, sec))


tasks = []
for j in [2, 5]:
    t = threading.Thread(target=f1, args=(j, ))
    t.setDaemon(True)
    tasks.append(t)
beg = time.time()
print('Main Thread start..')
for j in tasks:
    j.start()
time.sleep(1)
print('Main Thread end..')
end = time.time()
print(end - beg)
```

```
Main Thread start..
Task start, thread is Thread-1 value is 2
Task start, thread is Thread-2 value is 5
Task End,  thread is Thread-1 value is 2
Main Thread end..
1.00000001
```

1.4 join()阻塞主线程退出

主线程3秒结束, 两个子线程分别耗时2秒和5秒, 所以主线程退出后, 耗时1秒的线程可以正常结束, 耗时5秒的线程将会在主线程退出后而自动被退出. 要等待一个守护线程完成工作, 需要使用join方法来阻塞主线程的退出, 默认情况下join()会无限阻塞

In [ ]:
```
import threading
import time

def f1(sec):
    current = threading.current_thread().getName()
    print('Task start, thread is %s value is %s' %(current, sec))
    time.sleep(sec)
    print('Task End,  thread is %s value is %s' %(current, sec))


tasks = []
for j in [2, 5]:
    t = threading.Thread(target=f1, args=(j, ))
    t.setDaemon(True)
    tasks.append(t)
beg = time.time()
print('Main Thread start..')
for j in tasks:
    j.start()

time.sleep(3)

for j in tasks:
    j.join()

print('Main Thread end..')
end = time.time()
print(end - beg)
```

```
Main Thread start..
Task start, thread is Thread-1 value is 2
Task start, thread is Thread-2 value is 5
Task End,  thread is Thread-1 value is 2
Task End,  thread is Thread-2 value is 5
Main Thread end..
5.006648540496826
```

可以为join()传入一个浮点数值，表示等待守护线程退出的秒数，如果在这个时间内守护线程工作没结束，那么就退出.

In [ ]:
```
import threading
import time

def f1(sec):
    beg = time.time()
    current = threading.current_thread().getName()
    print('Task start, thread is %s value is %s' %(current, sec))
    time.sleep(sec)
    print('Task End,  thread is %s value is %s' %(current, sec))
    end = time.time()
    print(current, end - beg)


tasks = []
for j in [2, 5]:
    t = threading.Thread(target=f1, args=(j, ))
    t.setDaemon(True)
    tasks.append(t)


beg = time.time()
print('Main Thread start..')
for j in tasks:
    j.start()

for j in tasks:
    j.join(3)

print('Main Thread end..')
end = time.time()
print(end - beg)
```

```
Main Thread start..
Task start, thread is Thread-1 value is 2
Task start, thread is Thread-2 value is 5
Task End,  thread is Thread-1 value is 2
Thread-1 2.003145456314087
Main Thread end..
5.004770278930664
```

定时器线程:
通过线程定时器Timer，让线程在指定的时间后运行，在线程开始运行之前，可以通过cancel()方法将其取消.
示例代码:

In [2]:
```
#coding:utf8
import threading
import time
def delayed():
    print('threading name is %s' %threading.current_thread().getName())
    return
beg = time.time()
t1 = threading.Timer(3, delayed)
t1.setName("d1")
t1.start()
t1.join()
#t1.cancel() ##取消定时器线程
end = time.time()
print('Cost time %s' %(end-beg))
```

```
threading name is d1
Cost time 3.004868984222412
```

1.5 线程日志信息

可以将线程执行时的日志信息打印到指定的日志文件中，或是输出到屏幕上，通过logging模块可以实现该功能. logging是线程安全的，也就是可以通过多线程往logging中输出日志信息

In [ ]:
```
#示例代码:
import time
import threading
import logging



def f1():
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s',
                        datefmt='%Y-%m-%d %H:%M:%S',
                        filename='myapp.log',
                        filemode='w')

    console = logging.StreamHandler()
    console.setLevel(logging.DEBUG)
    formatter = logging.Formatter(
        '%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s')
    console.setFormatter(formatter)
    logging.getLogger().addHandler(console)

    try:
        time.sleep(1)
        x = 1 / 1
    except Exception as e:
        logging.error(str(e))
    else:
        logging.info("hello")
        return 'hello'


def f2():
    logging.basicConfig(level=logging.DEBUG,
                        format='%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s',
                        datefmt='%Y-%m-%d %H:%M:%S',
                        filename='myapp.log',
                        filemode='w')

    console = logging.StreamHandler()
    console.setLevel(logging.DEBUG)
    formatter = logging.Formatter(
        '%(asctime)s %(filename)s[line:%(lineno)d] %(process)d %(threadName)s %(levelname)s %(message)s')
    console.setFormatter(formatter)
    logging.getLogger().addHandler(console)

    try:
        time.sleep(1)
        x = 1 / 0
    except Exception as e:
        logging.error(str(e))
    else:
        logging.info("world")
        return 'world'


beg = time.time()
task = []
for _ in range(100):
    t1 = threading.Thread(target=f1)
    t2 = threading.Thread(target=f2)
    task.extend([t1, t2])

for j in task:
    j.start()

for j in task:
    j.join()

end = time.time()
print end - beg
```

日志文件 myapp.log 中出现200条日志信息，所以logging模块是线程安全的

2.1 线程锁--互斥锁
每个线程互相独立，相互之间没有任何关系．现在假设这样一个例子：有一个全局的计数num，每个线程获取这个全局的计数之后将其加1．很容易写出这样的代码：

In [32]:
```

```

```
Count is 0
Count is 0
Count is 1
Count is 3
Count is 4
Count is 5
Count is 5
Count is 6Count is 7

Count is 8
```

问题产生的原因就是没有控制多个线程对同一资源的访问，对数据造成破坏，使得线程运行的结果不可预期，这种现象称为"线程不安全"．

互斥锁同步
上面的例子引出了多线程编程的最常见问题--数据共享，当多个线程都修改某一个共享数据的时候，需要进行同步控制．

线程同步能够保证多个线程安全访问竞争资源，最简单的同步机制是引入互斥锁．互斥锁为资源引入一个状态：锁定/非锁定．某个线程要更改共享数据时，先将其锁定；此时资源的状态为"锁定"，其他线程不能更改；直到该线程释放资源，将资源的状态变成"非锁定"，其他的线程才能再次锁定该资源．互斥锁保证了每次只有一个线程进行写入操作，从而保证了多线程情况下数据的正确性．

threading模块中定义了Lock类，可以方便的处理锁定：

In [35]:
```
mutex = threading.Lock()
#锁定
mutex.acquire()
#释放
mutex.release()
```

同步阻塞
当一个线程调用锁的acquire()方法获得锁时，锁就进入"locked"状态，每次只有一个线程可以获得锁，如果此时另一个线程试图获得这个锁，该线程就会变为"blocked"状态，称为"同步阻塞"，直到拥有锁的线程调用锁的release()方法释放锁之后，锁进入"unlocked"状态，线程调度程序从处于同步阻塞状态的线程中选择一个来获得锁，并使得该线程进入运行(running)状态。

In [44]:
```python
#示例代码: 互斥锁

import threading
import time

count = 0
lock = threading.Lock()
def f1():
    lock.acquire()
    global count
    print("Count is %s" %count)
    time.sleep(0.0001)
    count += 1
    lock.release()

tasks = []
for _ in range(10):
    t =  threading.Thread(target=f1)
    tasks.append(t)

beg = time.time()
for j in tasks:
    j.start()

end = time.time()
print(end - beg)
```

```
Count is 0
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
0.0037958621978759766
Count is 6
Count is 7
Count is 8
Count is 9
```

2.2 死锁

所谓死锁: 是指两个或两个以上的进程或线程在执行过程中, 因争夺资源而造成的一种互相等待的现象, 若无外力作用, 它们都将无法推进下去, 此时称系统处于死锁状态或系统产生了死锁.

In [49]:
```python
#示例代码: 死锁

#coding:utf8
import threading
import time

lockA = threading.Lock()
lockB = threading.Lock()
def f1():
    lockA.acquire()
    print("A锁: 已锁定")  ##A锁已锁定
    time.sleep(1)
    lockB.acquire()
    print("B锁: 已锁定")  ##等待锁B释放 并重新对B加锁
    lockB.release()
    lockA.release()


def f2():
    lockB.acquire()
    print("B锁: 已锁定")  ##B锁已锁定
    time.sleep(1)
    lockA.acquire()
    print("A锁: 已锁定")  ##等待锁A释放 并重新对A加锁
    lockA.release()
    lockB.release()

t1 = threading.Thread(target=f1)
t2 = threading.Thread(target=f2)

t1.start()
t2.start()
```

2.4 线程池

在很多情况下, 我们需要控制线程的数量, 可以通过以下的方法.

In [ ]:
```python
import threading
import time

print time.time()
def thread_a():
    time.sleep(5)
    print 'this is function a'

def thread_b():
    time.sleep(2)
    print 'this is function b'

def thread_c():
    time.sleep(3)
    print 'this is function c'

def thread_d():
    time.sleep(4)
    print 'this is function d'
func_list = filter((lambda x: x.startswith('thread_')), globals())

def main():
    thread_num = 2
    while True:
        thread_count = min([thread_num, len(func_list)])
        threads = []
        if not func_list:
            break
        for i in range(thread_count):
            t = threading.Thread(target=globals().get(func_list.pop()), args=())
            threads.append(t)
        for i in range(thread_count):
            threads[i].start()
        for i in range(thread_count):
            threads[i].join()

if __name__ == '__main__':
    main()
```

2.5 Condition线程条件阻塞

传统线程技术实现互斥只能是一个线程单独工作, 不能实现一个线程工作一段时间停止后再通知另一个线程来工作, Condition就是解决这个问题, 线程1执行(cond), 线程1执行一半等待(cond.wait()), 线程2开始执行(cond), 线程2执行完毕后(cond.notify()), 线程1再接着执行.

示例代码：

In [3]:
```
import threading
import time

def go1():
    with cond:  #使用条件变量(资源 Lock)
        for  i  in range(8):
            time.sleep(1)
            print(threading.current_thread().name,i,"go11")
            if i==5:
                cond.wait() #等待cond.notify(), 再继续执行。（释放条件变量(资源 Lock)）

def go2():
    with  cond:  #使用条件变量(资源 Lock)
        for i in range(7):
            time.sleep(1)
            print(threading.current_thread().name, i)
        cond.notify()  #通知，触发 cond.wait()。（释放条件变量(资源 Lock)）

cond=threading.Condition()  #线程条件变量
threading.Thread(target=go1).start()  #和下面的线程的次序不能调。这个线程先拿到cond条件变量(资源 Lock)
threading.Thread(target=go2).start()
```

```
Thread-6 0 go11
Thread-6 1 go11
Thread-6 2 go11
Thread-6 3 go11
Thread-6 4 go11
Thread-6 5 go11
Thread-7 0
Thread-7 1
Thread-7 2
Thread-7 3
Thread-7 4
Thread-7 5
Thread-7 6
Thread-6 6 go11
Thread-6 7 go11
```

3.3 线程间信号传输

线程对象包含一个可由线程设置的信号标志，它允许线程等待某些事件的发生．在初始情况下Event对象中的信号标志flag被设置为false，如果有线程等待一个Event对象，而这个Event对象的标志为假，那么这个线程将会被一直阻塞直至该标志为true．一个线程如果将一个Event对象的信号标志设置为真，它将唤醒所有等待这个Event对象的线程，如果一个线程等待一个已经被设置为真的Event对象,那么它将忽略这个事件，继续执行．

Event几种方法：

event.isSet()：返回event的状态值

event.wait()： 如果event.isSet()==False将阻塞线程

event.set()：   设置event的状态值为True，所有阻塞池的线程激活进入就绪状态，等待操作系统调度．

event.clear()：恢复event的状态值为False。

示例代码：

In [3]:
```
import threading
import time

event=threading.Event()

def light():
    print('红灯正亮着')
    time.sleep(3)
    event.set() #绿灯亮

def car(name):
    print('车%s正在等绿灯' %name)
    event.wait() #等灯绿 此时event为False,直到event.set()将其值设置为True,才会继续运行.
    print('车%s通行' %name)

if __name__ == '__main__':
    # 红绿灯
    t1=threading.Thread(target=light)
    t1.start()
    # 车
    t=threading.Thread(target=car,args=(1,))
    t.start()
```

```
红灯正亮着
车1正在等绿灯
车1通行
```

注：
内置函数globals，可以看到当前脚本中的所有变量与函数．

In [2]:
```
def f1():
    return 1

globals()
```

Out[2]:
```
{'__name__': '__main__',
 '__doc__': 'Automatically created module for IPython interactive environment',
 '__package__': None,
 '__loader__': None,
 '__spec__': None,
 '__builtin__': <module 'builtins' (built-in)>,
 '__builtins__': <module 'builtins' (built-in)>,
 '_ih': ['', 'globals()', 'def f1():\n    return 1\n\nglobals()'],
 '_oh': {1: {...}},
 '_dh': ['/Users/liushuo/python_training/Python编程进阶'],
 'In': ['', 'globals()', 'def f1():\n    return 1\n\nglobals()'],
 'Out': {1: {...}},
 'get_ipython': <bound method InteractiveShell.get_ipython of <ipykernel.zmqshell.ZMQInteractiveShell object at 0x110adaba8>>,
 'exit': <IPython.core.autocall.ZMQExitAutocall at 0x110baafd0>,
 'quit': <IPython.core.autocall.ZMQExitAutocall at 0x110baafd0>,
 '_': {...},
 '__': '',
 '___': '',
 '_i': 'globals()',
 '_ii': '',
 '_iii': '',
 '_i1': 'globals()',
 '_1': {...},
 '_i2': 'def f1():\n    return 1\n\nglobals()',
 'f1': <function __main__.f1()>}
```

In [ ]: