**http://john16blog.blogspot.ru/2013/02/python-signal.html**

[**Python: модуль signal. Перевод документации. Примеры.**](http://john16blog.blogspot.ru/2013/02/python-signal.html)

Модуль предоставляет механизм использования обработчиков сигналов. Некоторые общие правила для работ с сигналами и их обработчиками:

* Обработчик для конкретного сигнала, однажды установленный, остается таковым пока не будет явного сброса (Python эмулирует интерфейс в стиле BSD, независимо от конкретной реализации), за исключением обработчика для SIGCHLD, который следует за базовой реализации.
* Не существует способа временно "заблокировать" сигналы от критических секций (так как это не поддерживается для всех разновидностей Unix).
* Хотя обработчики сигналов в Python вызываются асинхронно, в этом заинтересован пользователь Python, они могут проявиться только между атомарными инструкциями интерпретатора Python. Это означает что сигналы поступившие во время длительных вычислительным реализаций на чистом C (такие как поиск совпадение регулярного выражения в большом фрагменте текста) могут быть отложены на произвольное количество времени.
* Когда сигнал прибывает во время выполнения операций ввода/вывода, то ими возможно возбуждение исключительных ситуаций после возвращения обработчиков сигналов. Это зависит от базовой семантики UNIX подобных систем относительно прерванных системных вызовов.
* Поскольку обработчик сигнала в C всегда возвращается, вряд ли имеет смысл ловить синхронные ошибки такие как SIGFPE или SIGSEGV.
* Python устанавливает небольшое количество обработчиков сигналов по умолчанию: SIGPIPE игнорируется (так ошибки записи на каналах и сокетах могут быть представлены как обычные Python исключения) и SIGINT транслируется в KeyboardInterrupt исключение. Все они могут быть переопределены.
* Некоторое внимание должно быть уделено если сигналы и потоки используется вместе в одной программе. Основное что нужно запомнить при одновременном использовании сигналов и потоков: всегда выполнять операции signal() в основном потоке выполнения. Любой поток может выполнять alarm(), getsignal(), pause(), settimer(), gettimer(); только главный поток может устанавливать новые обработчики сигналов и он же единственный кто может получать сигналы (это обеспечивается модулем signal, даже если базовая реализация потоков поддерживает посылку сигналов индивидуальным потокам). Это означает что сигналы не могут быть использованы для межпотокового сообщения. Используйте блокировки вместо этого.

Переменные определенные в модуле:   
  
signal.**SIG\_DFL**

Это одна из двух стандартных опций обработки сигналов. Она будет исполнять функцию по умолчанию для данного сигнала. например, на большинстве систем стандартное действие на сигнал SIGUIT является дамп ядра и выход, в то время как для сигнала SIGCHLD это игнорирование сигнала.

signal.**SIG\_IGN**

Это другой стандартная опция обработчика сигналов. Она означает простое игнорирование получаемого сигнала.

**SIG\***

Все номера сигналов определены в модуле символически. Например сигнал hangup определен как signal.SIGHUP; имена переменных идентичны именам используемых в C, которые могут быть найдены в исходной файле <signal.h>. Unix страница помощи signal() предоставляет список существующих сигналов( на некоторых системах это signal(2) или signal(7)). Помните что не все системы определяют одинаковый список имен сигналов; только те имена что определены в системе определены в данном модуле.

signal.**CTRL\_C\_EVENT**

Сигнал сообщает о нажатии клавиши CTRL+C. Этот сигнал может использоваться только с os.kill().   
  
Доступно только на Windows OS   
Новое в версии 2.7

signal.**CTRL\_BREAK\_EVENT**

Сигнал сообщает о нажатии клавиши CTRL+BREAK. Этот сигнал может использоваться только с os.kill().   
  
Доступно только на Windows OS   
Новое в версии 2.7

signal.**NSIG**

Значение на единицу большее значения самого большого номера сигнала.

signal.**ITIMER\_REAL**

Уменьшает интервал таймера в режиме реального времени, и передает SIGALRM по истечении срока.

signal.**ITIMER\_VIRTUAL**

Уменьшает интервал таймера только когда процесс выполняется, и передает SIGVTALRM по истечении срока.

signal.**ITIMER\_PROF**

Уменьшает интервал таймера когда процесс выполняется и когда система выполняется от имени процесса. Вместе с ITIMER\_VIRTUAL этот таймер обычно используется для профилирования время потраченного приложением в пространстве пользователя и ядра. SIGPROF доставляется по истечению срока.

Модуль определяет одно исключение: signal.**ItimerError**

Возбуждается что бы сигнализировать об ошибке из базовых реализаций settimer() или gettimer(). Ошибка проявляется в случае передачи в settimer() некорректного интервала времени или отрицательного значения времени. Данное исключение является подтипом исключения IOError.

Модуль определяет следующие функции:   
  
signal.**alarm**(*time*)

Если *time* не равен нулю, то функция запрашивает посылку сигнала SIGALRM процессу через *time* секунд. Любые предыдущие запланированные тревоги будут отменены (только одна тревога может быть запланированна в одно и то же время). Возвращаемое значение является количеством секунд прежде чем будет вызвана предыдущая тревога. Если *time* равен нулю, то никаких тревог не будет запланировано, а запланированные ранее будут отменены. Если возвращаемое значение равно нулю, то никакая тревога на данный момент не запланированна. (Смотри man страницу alarm(2)). Доступно для UNIX.

signal.**getsignal**(*signalnum*)

Возвращает текущий обработчик сигнала для сигнала *signalnum*. Возвращаемое значение может быть вызываемым объектом Python или одним из специальных значений signal.SIG\_IGN,signal.SIG\_DFL, None. Здесь signal.SIG\_IGN означает что сигнал ранее был проигнорирован, signal.SIG\_DFL - что по-умолчанию способ обработки сигнала был ранее в использовании, None - предыдущий обработчик сигнала был установлен не из Python.

signal.**pause**()

Заставляет процесс уснуть до тех пор пока сигнал не будет получен; далее будет вызван соответствующий обработчик. Ничего не возвращает. Доступен в UNIX. (Смотри man страницу signal(2)).

signal.**settimer**(*which, seconds [, interval]*)

Устанавливает передаваемый временной таймер( один из signal.ITIMER\_REAL, signal.ITIMER\_VIRTUAL,signal.ITIMER\_PROF) указанный в *which* на активацию через *seconds* (примимаемое значение вещественное float, в отличии от alarm()) и каждые *interval* секунд после этого. Таймер может быть стерт если передать 0 в *seconds*.   
  
Когда таймер срабатывает сигнал посылается процессу. Посылаемый сигнал завист от таймера который используется; signal.ITIMER\_REAL будет посылать SIGALRM, signal.ITIMER\_VIRTUAL посылает SIGVTALRM, signal.ITIMER\_PROF будет посылать SIGPROF.   
  
Старые значения возвращаются в виде кортежа (задержка, интервал).   
  
Попытка передачи недествительного значения интерала таймера вызовет исключение ItimerError. Доступно в UNIX.   
  
*Новое в Python 2.6*

signal.**gettimer**(*which*)

Возвращает текущее значение заданного временного таймера *which*. Доступно в UNIX.   
  
*Новое в Python 2.6*

signal.**set\_wakeup\_fd**(*fd*)

Установить пробуждающий файловый дескриптор в *fd*. Когда процесс получает сигнал '\0' байт записывается в файловый дескриптор. Это может быть использованно библиотекой для пробуждения опрашивающего или выбираюшего вызова, позволяющего сигналу быть полностью обработанным.   
  
Старый пробуждающий файловый дескриптор возвращается. *fd* должен быть не блокирующий. Это должно быть до библиотеки, что бы удалить все байты перед очередным вызовом.   
  
В случае многопоточности эта функция может быть вызвана толкьо из основного потока; попытка вызова из других потоков вызовет исключение ValueError.   
  
*Новое в версии 2.6*

signal.**siginterrupt**(*signalnum, flag*)

Изменяет поведение системного вызова перезапуска: если *flag* равен False системные вызовы будут перезапущены когда произойдет прерывание сигналом *signalnum*, в противном случае системные вызовы будут прерваны. Ничего не возвращает. Доступно в UNIX (смотри man страницу siginterrupt(3)).   
  
Помните что устанавливая обработчик сигнала используя signal() будет сброшено поведение перезапуска на прерывающее неявным вызовом siginterrupt() c истинным значением *flag* для данного сигнала.   
  
*Новое в версии Python 2.6*

signal.**signal**(*signalnum, handler*)

Устанавливает обработчик для сигнала *signalnum* функцию *handler*, которая может быть вызываемым объектом Python принимающее два аргумента или одним из специальных значений: signal.SIG\_IGN,signal.SIG\_DFL. Возращает предыдущий обработчик сигнала. (Смотри man страницу signa(2)).   
  
В случае многопоточности эта функция может быть вызвана только из основного потока; попытка вызова из других потоков вызовет исключение ValueError.   
  
*handler* вызывается с двумя аргументами: номером сигнала и текущим фреймом стэка (None или объект фрейма, смотри модуль inspect и [модель данных Python](http://docs.python.org/2/reference/datamodel.html#frame-objects)).   
  
На Windows signal() может быть вызван только с значениями SIGABRT, SIGFPE, SIGILL, SIGINT, SIGSEGV, SIGTERM, в противном случае будет возбуждено исключение ValueError.

Примеры

Пример с использованием alarm, в качестве таймаута для исполняемого фрагмента кода.

#-\*- coding: utf-8 -\*-  
import signal, sys, time  
  
def handle(signum, frame):  
    print 'Alarm! job failed'  
    sys.exit(1)  
      
def job(n):  
    data=[]  
    for x in xrange(n):  
        data.append(x\*\*2)  
        time.sleep(0.1)  
          
signal.signal(signal.SIGALRM, handle)  
  
# Имитация простой работы, которая будет выполнена успешно  
signal.alarm(3)  
job(10)  
print 'Job 1 done'  
signal.alarm(0)  
  
# Имитация объемной работы, которая не будет успешно выполнена   
signal.alarm(3)  
job(100)  
print 'Job 2 done'  
signal.alarm(0)

В результате видно, что вторая иммитация работы будет прервана посылкой и обработкой сигнала, так как время выполнения кода будет больше, чем время таймаута:

Job 1 done  
Alarm! job failed

Сигналы и ООП. Пример работы с таймером.

Очень полезный пример, показывающий как применять обработку сигналов внутри экземпляра класса. Для этого создается вспомогательный класс Signal\_handler, в конструктор которого посылается экземпляр необходимого класса, для того что бы в последствии на него сослаться из функции обработчика сигнала.  
В примере, по средством установки таймера, провярющего прошло ли 10 секунд с момента запуска программы, осуществляется дискретный контроль, путем посылки сигнала с интервалом в 3 секунды.

#-\*- coding: utf-8 -\*-  
import signal, time, datetime  
  
class Signal\_handler(object):  
    """ Класс обрабатывающий системные сигналы """  
      
    def \_\_init\_\_(self, target):  
        """target - экземпляр класса"""  
        self.target=target  
          
    def handle\_sigalarm(self, signum, frame):  
        self.target.process\_alarm\_signal()  
      
          
class Foo(object):  
      
    def \_\_init\_\_(self, \*args, \*\*kwargs):  
        # Регистрируем обработчики сигналов   
        sh=Signal\_handler(self)  
        signal.signal(signal.SIGALRM, sh.handle\_sigalarm)  
        # Устанавливаем таймер  
        signal.setitimer(signal.ITIMER\_REAL, 3, 3)  
      
    def process\_alarm\_signal(self):  
        print '{} - alarm\_checking'.format(datetime.datetime.now())  
        if (datetime.datetime.now()-self.datetime\_mark).seconds>=10:  
            # устанавливаем переменную для выхода из цикла  
            self.need\_break=True  
            # отключаем таймер  
            signal.setitimer(signal.ITIMER\_REAL, 0)  
      
    def run(self):  
        self.need\_break=False  
        self.datetime\_mark=datetime.datetime.now()  
        print '{} - Run'.format(datetime.datetime.now())  
        while not self.need\_break:  
            # симуляция работы  
            time.sleep(1)  
          
if \_\_name\_\_=='\_\_main\_\_':  
    foo=Foo()  
    foo.run()

В результате видно, что несмотря на то что старт был осуществлен в 30 секунд, сигнал, пришел только на 4ой итерации по 3 секунды:

2013-02-27 20:26:30.958092 - Run  
2013-02-27 20:26:33.958116 - alarm\_checking  
2013-02-27 20:26:36.958111 - alarm\_checking  
2013-02-27 20:26:39.958112 - alarm\_checking  
2013-02-27 20:26:42.958108 - alarm\_checking

Небольшой пример работы с pause.

Код ниже, расчитанный на выполнение 10 итераций...

#-\*- coding: utf-8 -\*-  
import signal, time, datetime  
  
def handle(signum, frame):  
    print '{} - Signal arrived'.format(datetime.datetime.now())  
      
def job(n):  
    for x in xrange(n):  
        if x ==5:  
            signal.pause()  
        print '{} - value = {}'.format(datetime.datetime.now(), x\*\*2)  
        time.sleep(0.5)  
          
signal.signal(signal.SIGALRM, handle)  
  
job(10)

остановится на 6ой в ожидании поступающих сигналов...

2013-02-27 20:37:34.432161 - value = 0  
2013-02-27 20:37:34.933044 - value = 1  
2013-02-27 20:37:35.433649 - value = 4  
2013-02-27 20:37:35.934225 - value = 9  
2013-02-27 20:37:36.434812 - value = 16

После посыла сигнала из консоли linux с помощью команды kill -s SIGALRM *pid* возобновит прохождение цикла.

2013-02-27 20:37:47.788603 - Signal arrived  
2013-02-27 20:37:47.788659 - value = 25  
2013-02-27 20:37:48.288741 - value = 36  
2013-02-27 20:37:48.789336 - value = 49  
2013-02-27 20:37:49.289925 - value = 64  
2013-02-27 20:37:49.790082 - value = 81