**http://john16blog.blogspot.ru/2010/10/python-subprocess.html**

[**Python: модуль subprocess. Перевод документации с комментариями и примерами. Особенности применения.**](http://john16blog.blogspot.ru/2010/10/python-subprocess.html)

Начиная с версии 2.4, библиотека Python’а включает в себя модуль subprocess – унифицированное средство для запуска новых процессов, подключения к их потокам ввода, вывода, ошибок, и получения возвращаемого кода по завершению. Он призван заменить такие инструменты как os.system, os.spawn\*, os.popen\*, popen2.\*,commands.\* единым механизмом.

*Условимся что во всей статье мы работаем в каталоге ~ в котором расположена основная программа. Модуль импортировался командой from subprocess import \*, остальные модули простым импортированием,в коде присутствуют полные ссылки. Примеры рассматривались в Windows XP SP3 + python 2.6.2 и openSUSE 11.3 linux + python 2.6.5, справочный материал взят из версии 2.6, в версии 2.7 особо важных изменений нет.*

Определение класса Popen

Модуль определяет в себе основной для использования класс Popen, который вызывает процесс и возвращает управление программе не ожидая его завершения:

class subprocess.Popen(*args, bufsize=0, executable=None, stdin=None, stdout=None, stderr=None, preexec\_fn=None, close\_fds=False, shell=False, cwd=None, env=None, universal\_newlines=False, startupinfo=None, creationflags=0*)

args – строка или последовательность аргументов программы. Обычно первым указывают исполняемую программу, а затем аргументы, но также ее можно указать в параметре executable.

На UNIX-подобных ОС с *shell=False* (поведение по умолчанию): в этом случае Popen класс использует os.execvp() что бы выполнить дочернюю программу. *args* обычно представляется последовательностью. Строка будет обработана как последовательность с единственным элементом.

На UNIX-подобных ОС с *shell=True*: если *args* строка, то она является командной строкой для выполнения в оболочке. Если *args* последовательность, то первый элемент определяет команду, а остальные элементы обрабатываются как дополнительные аргументы оболочки.

На Windows: класс Popen использует CreateProcess() для выполнения дочерней программы, который работает на основе строк. Если *args* последовательность, то она будет конвертирована в строку, используя list2cmdline() метод.

*Я бы рекомендовал всегда оформлять аргумент args в виде списка. Это даст преимущества, в случае если в пути к исполняемому файлу будут, например, пробелы, а сам путь передается в Popen виде переменной, а не явно задается в коде. Что бы корректно обработать пробелы в случае строки нужно дополнительно ставить скобки (например, при вызове программы prog в директории my folder и передачи ей аргумента 4, вместо ‘./my folder/prog 4’ писать ‘”./my folder/prog” 4’), в случае последовательности Питон сделает все сам.*

*#Пример ниже вызвает программу prog находящуюся в каталоге my folder и передает ей параметр 4:   
child=Popen([‘./my folder/prog’, ’4’])   
  
# пример вызывает встроенную функцию UNIX-подобных ОС ls   
child=Popen([‘ls -l’], shell=True)*

bufsize если задан, имеет тоже значение, что и передаваемый параметр в встроенную функцию open(): 0 – без буферизации, 1 – линейная буферизация, любое другое положительное значение – использование буфера заданным размером (значение будет аппроксимировано), любое другое отрицательное значение -; использование системных настроек по умолчанию, что обычно означает полную буферизацию.

*разработке*

executable – определяет программу для выполнения. Этот параметр редко нужен: обычно программу для выполнения определяют в аргументе *args*. Если *shell=True*, то *executable* определяет оболочку, которая будет использоваться. В UNIX-системах по умолчанию оболочкой является /bin/sh, в Windows по умолчанию оболочка определяется COMSPEC переменой среды.

#Пример ниже выполнит процесс arg передав ему параметры.   
child=Popen(['-n', '2', 'my.txt'], executable='arg')   
  
#Пример ниже показывает как сменить оболочку для выполнения   
child=sp.Popen(['--version'], executable='/bin/zsh', shell=True)   
zsh 4.3.10 (i686-pc-linux-gnu)   
  
# Однако с bash результат работы оказывается неверным задуманному, причина такого поведения мне на данный момент не ясна:   
child=sp.Popen(['--version'], executable='/bin/bash', shell=True)   
/bin/sh: --: недопустимая опция

stdin, stdout, stderr определяют стандартные потоки ввода, вывода, ошибок соответственно. Аргументы могут принимать значения PIPE, существующий файловый дескриптор, существующий файловый объект, None. Значение PIPE указывает на создание нового канала (pipe). Со значением None перенаправление происходить не будет, дочерний процесс наследует поток от предка. stderr – может принимать значение STDOUT, означающее, что вывод ошибок будет происходить в поток stdout.

*#Пример ниже вызовет процесс, перехватит его выводы в канал и считав выведет на устройство вывода.   
child=sp.Popen(['ls'],shell=True,stdout=sp.PIPE)   
s=' '   
while s:   
    s=child.stdout.readline()   
    print s.rstrip()*

*Здесь стоит помнить, что цикл будет работать до тех пор пока вызываемое приложение не отошлет в канал символ конца файла EOF, сигнализируя о завершении работы. В этом случае метод readline вернет пустую строку и тогда перестанет выполняться условия цикла.*

*ВНИМАНИЕ: Если вызываемая программа работает в бесконечном цикле или ожидает ввода данных от пользователя или просто долго не завершает свою работу (не посылает EOF)то любой метод чтения перейдет в режим ожидания данных от вызываемой программы и «повесит» основную программу.*

*#пример ниже создаст для потока вывода лог файл, в который будет осуществляться вывод.   
child=sp.Popen(['ls'],shell=True,stdout=open(‘out.log’,’a’)   
  
# Предположим что у вас есть приложение prog которое ожидает ввода от пользователя данных для продолжения работы, тогда ее вызов будет выглядеть так:   
сhild=sp.Popen(["./prog"],stdin=sp.PIPE, shell=False)   
child.stdin.write(“100\n”)*

Если значению аргумента *preexec\_fn* присвоен вызываемый объект, то этот объект будет вызван в дочернем процессе перед его выполнением (только для UNIX).

*# пример основан на демо в самом модуле subprocess.py. Перед вызовом процесса меняем его UID (выполнять скрипт нужно с соответствующими правами).   
sp.Popen(["id"])   
sp.Popen(["id"], preexec\_fn=lambda: os.setuid(1000))   
sp.Popen(["id"])   
  
# Результат работы:   
uid=0(root) gid=0(root) groups=0(root)   
uid=1000(john\_16) gid=0(root) groups=0(root)   
uid=0(root) gid=0(root) groups=0(root)*

Если аргументу *close\_fds* присвоено значение True, то все дескрипторы, исключая 0, 1 и 2 будут закрыты перед выполнением дочернего процесса (только для UNIX).

*Данная опция полезна если в вызывающей программе присутствуют, например, открытые сокеты, а запустив процесс мы передадим ему дескрипторы родителя тем самым сделав этот сокет занятым обоими процессами и закрыть сокет можно будет только после того как завершит работу дочерний процесс*

*# пример ниже создает сокет и привязывает его к адресу, затем создает процесс, предварительно закрыв для дочернего процесса все дескрипторы. В момент паузы можно воспользоваться командой оболочки lsof –i что бы посмотреть все открытые сокеты.   
soc = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)   
soc.bind(('localhost', 9988))   
soc.listen(5)   
child=sp.Popen(['./loop'], close\_fds=True)   
print 'child PID', child.pid   
soc.close()   
print 'see result'   
time.sleep(10)*

Если аргументу *shell* присвоено значение True, то команда будет выполнена через оболочку.

*Можно сказать, что данная опция наиболее характерна для ОС Windows.*

*# пример ниже воспользуется командой оболочки OS Windows для просмотра содержимого текущего каталога   
child=sp.Popen(['dir'],shell=True)   
  
# пример ниже показывает аналогичное в UNIX   
child=sp.Popen(['ls -l'],shell=True)   
  
# однако эта же команда работает и без указания shell. Стоит обратить внимание что в случае shell=True необходимая команда задается целиком одной строкой, в случае False это может быть последовательность (об этом говорилось в начале):   
child=sp.Popen(['ls','-l'],shell=False)*

Если значение аргумента *cwd* не является None, то директория дочернего процесса будет изменена на это значение перед его выполнением. Однако это директория не рассматривается при поиске исполняемого файла (это утверждение я опровергну ниже).

*Я рекомендую всегда указывать этот параметр, объясню на примере.*

*#(пример для OS Windows) Допустим у нас есть основная программа в каталоге ~ которая вызывает программу fl.exe находящуюся в каталоге ~\folder, работа которой состоит в создании файла tmp в текущей для нее директории. Если мы воспользуемся следующим кодом):   
child=sp.Popen(["folder\\fl.exe", "3"])   
#, то убедимся, что файл tmp был создан не в каталоге ~\folder, где находится создавшая его программа, а в каталоге ~, где находится вызывающая программа. Часто это не желаемое поведение программы. Это произошло потому что дочерний процесс унаследовал от предка его рабочую директорию. В противном случае код будет выглядеть следующим образом:   
child=sp.Popen(["folder\\fl.exe", "3"], cwd=’folder’)   
# или в случае если полный путь до программы задается переменной удобно воспользоваться следующим:   
path=”D:\\~\\folder\\fl.exe”   
child=Popen([path,3], cwd=os.path.dirname(path), shell=False)   
  
#В случае с Линуксом я нашел расхождение между документацией и реальным поведением. При таком же расположении файлов вызов:   
child=sp.Popen(["./folder/fl", "3"], cwd="./folder")   
# привет к ошибке отсутствия файла, но вызов:   
child=sp.Popen(["./fl", "3"], cwd="./folder", shell=False)   
# произойдет успешно. Так как в случае с Windows подобного не наблюдается, я не исключаю возможности бага как в самом питоне, так и в самой документации.*

Если значение аргумента *env* не является None, то оно должно быть mapping, который определяет переменные окружения для нового процесса. Используется вместо наследования окружения текущим процессом, что является поведением по умолчанию.

*# Пример ниже передает дочернему процессу измененное окружение:   
env = os.environ.copy()   
env["PROJECTPATH"] = “/something”   
child=subprocess.Popen(["./someexe"], env=env)*

Если аргументу *universal\_newlines* присвоено значение True,то файловые объекты stdout,stderr открываются как текстовые файлы, но строки могут быть окончены любым способом представления: ‘\n’ – UNIX, ‘\r’ – старые версии Macintosh, ‘\r\n’ – Windows. Все эти представления конвертируются в ‘\n’ подобном представлении внутри Питона.

*После некоторого обдумывания я пришел к выводу, что не пользоваться этой возможностью нет смысла т.е. вреда от нее нет даже теоретического, а польза вполне очевидна*

Если заданы аргументы *startupinfo* и *creationflags*, то их значение передаются в функцию CreateProcess(). Аргументы отвечают за поведение основного окна и приоритета процесса (только для Windows).

*Подробная информация о возможных значениях параметров и их цифровых кодах приведена в MSDN или в ее русскоязычном переводе:* [*флаги*](http://vsokovikov.narod.ru/New_MSDN_API/Process_thread/flags_create_process.htm) *и* [*стартовая информация.*](http://vsokovikov.narod.ru/New_MSDN_API/Process_thread/str_startupinfo.htm)

*Отмечу, что переменные находятся в пространстве имен модуля win32process.*

*В одном из своих проектов я писал приложение с GUI которое запускало бы приложение и весь его вывод помещала в текстовой виджет. Это приложение находилось, предположим, в каталоге ~, вызываемый скрипт в ~\folder, который циклично раз в сутки запускал, рядом лежащую, консольную программу prog.exe с заданными параметрами. Сам скрипт работал, как задумано, но в случае если его самого запускал сторонний программный код, то появлялось пустая консоль от программы prog.exe (пустая потому что вывод от нее корректно перехватывался скриптом). Тогда я не понимал, почему такое происходит, ведь я же задал stdout. Как оказалось это суждение ошибочно и MS Windows имеет свои нюансы. Проблема решилась установкой параметра crateflags=8, что соответствует значению DETACHED\_PROCESS. Внимание: при программировании под Windows крайне желательно принимать во внимание эти параметры и не опираться на поведение по умолчанию.*

subprocess.**call** (*\*popenargs,\*\*kwargs*)

Запуск команды с аргументами (такими же как и в случае с Popen), ожидание завершения работы и возвращение кода завершения *returncode*.

*Данная функция является всего лишь вызовом Popen(\*popenargs,\*\*kwargs).wait()*

subprocess.**check\_call**(*\*popenargs, \*\*kwargs*)

Запуск команды с аргументами, ожидание завершения работы, Если код завершения равен 0, то возвращает его, в другом случае возбуждается исключение CalledProcessError, объект которого возвращает код завершения атрибутом *returncode*.

Исключительные ситуации.

Исключения, возбужденные в дочернем процессе перед началом выполнения процесса будут переданы предку. Также, объект исключения будет иметь дополнительный атрибут *child\_traceback*, представляющий собой строку содержащую информацию об обратной трассировке с точки зрения дочернего процесса.

Наиболее часто встречающееся исключение OSError происходит, например, при попытке выполнения несуществующего файла [Error 2]. Приложения должны быть подготовлены для обработки таких случаев.

ValueError будет возбуждено если в Popen передать неверные аргументы.

Check\_all() будет вызывать исключение CalledProcessError если вызываемый процесс вернет не нулевое значение.

Методы экземпляра класса Popen.

Popen.**poll()**

если процесс завершил работу возвращает returncode, в другом случае None.

Popen.**wait()**

ожидает завершения работы процесса и возвращает *returncode*.Предупреждение: это может вызвать блокировку(зависание), если дочерний процесс генерирует достаточное количество данных в stdout,stderr канал так что блокирует ожидание буфера канала ОС для принятия еще данных. Использование communicate() позволит избежать этого.

Popen.**comminucate**(*input=None*)

взаимодействовать с процессом, послать данные содержащиеся в input (должна быть строка) в stdin процесса, ожидает завершения работы процесса, возвращает кортеж потока вывода и ошибок. При этом в Popen необходимо задать значение PIPE для stdin,stdout,stderr. Прочитанные данные буферизируются в память, поэтому не стоит применять этот метод в случае огромных или без лимитных выходных данных.

#Пример ниже вызовет приложение prog ожидающее ввода 2 значений, перехватит потоки выводов и по окончании выведет на стандартное устройство вывода содержание потоков.   
stdout,stderr=sp.Popen(["./inp"], stdin=sp.PIPE, stdout=sp.PIPE).communicate('3\n5\n')   
print 'output pipe:\n',stdout   
print 'error pipe:\n',stderr

Popen.**send\_signal**(*signal*)

послать сигнал *signal* дочернему процессу. Для Windows поддерживается только сигнал SIGTERM, являющийся синонимом для terminate().

Popen.**terminate**()

остановка дочернего процесса. На POSIX системах метод посылает процессу сигнал SIGTERM, на Windows вызывается Win32 API функция TerminateProcess.

Popen.**kill**()

убить дочерний процесс. На POSIX системах метод посылает процессу сигнал SIGKILL, на Windows метод является синонимом terminate().

Атрибуты экземпляра класса Popen:

Popen.**stdin/stdout/stderr**

если в Popen stdin/stdout/stderr был задан как PIPE, то является файловым объектом обеспечивающий поток ввода/вывода/ошибок с процессом.

Popen.**returncode**

если процесс завершил работу принимает значение возвращаемого кода, иначе None. В UNIX возможны отрицательные значения, свидетельствующие о том, что процесс был завершен сигналом данного номера.

Некоторые хитрости.

*Как мы могли убедиться при stdout=PIPE вывод print child.stdout.readline() осуществляется только после того, как поток получил EOF. А что если необходимо получить вывод по мере его поступления? Возможно bufsize для этого и предназначен, однако у меня это не работает. Но если речь идет о запуске скриптов на питоне то можно поступить несколькими способами:   
  
1) передать интерпретатору ключ –i, который активизирует интерактивный режим, после выполнения скрипта появится интерактивная консоль. Нам этого не надо, поэтому, что бы этого избежать, в вызываемом скрипте в конце нужно добавить os.\_exit(0).в Итоге имеем:*

*child=sp.Popen(["python", '-i', "./folder/four\_line.py"], stdout=sp.PIPE)   
s=' '   
while s:   
    s=child.stdout.readline()   
    if s :   
        print s.strip()*

*2) в вызываемом скрипте после необходимых print “something” выталкивать буферы потока вывода, например, sys.stdout.flush() в случае если вывод не был переопределен.*