ЛАБОРАТОРНЫЙ ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ НА ЯЗЫКЕ PYTHON

(Python Beginning)

Семь бед, один Reset.

Урок 11. (Lesson 11) Системные инструменты

Потоки выполнения представляют собой один из механизмов одновременного исполнения нескольких операций. При этом создаваемые потоки являются «дочерними» по отношению к породившей их программе, но

Управление процессом запуска скрипта

Язык программирования Python создан средствами языка *C* и несет в себе многие его свойства. Да и синтаксис отчасти выдает наличие «родственных связей». При запуске *C*-программа имеет возможность прочитать параметры функции `main()`: `int argc` - количество параметров, `char** argv` - массив строк параметров, `char** envp` - массив строк переменных окружения, передаваемые ей операционной системой через интерфейс командной оболочки. Похожие механизмы реализованы и в *Python*.

- ### Текущий рабочий каталог

При изучении основ программирования под Linux вам пришлось освоить системную утилиту `pwd` (*Print Working Directory*), выводившую в стандартный поток имя текущей директории. Понятие текущего рабочего каталога очень важно для системных приложений, так или иначе использующих доступ к файловой системе: создают, перемещают, редактирую файлы, – даже подключение к локальной БД требует от нас указания пути и имени файла БД. Программы – сценарии на языке Python, тоже используют понятие рабочего каталога: оно может быть использовано для организации группы взаимодействующих между собой приложений и для многого другого. Использование данных `current working directory` (*CWD*) позволяет формировать значения абсолютных имен файлов из относительных путей, организовывать правильное взаимодействие файлов сценариев.

Для получения данных о имени текущего рабочего каталога используется системный вызов `os.getcwd()`. Да, это именно системный вызов, со всеми вытекающими из этого последствиями – переключение режима работы приложения из пользовательского режима в режим ядра. Пусть это и скрыто от за оболочной *Python Virtual Machine* (*PVM*), но оно связано с дополнительными тактами работы центрального процессора, что, как вы понимаете, приводит к суммарному снижению производительности программной системы.

Текущий рабочий каталог не связан с значением системной переменной *PYTHON-PATH*. Также следует различать текущий рабочий каталог и каталог, содержащий файл сценария. Это можно наглядно продемонстрировать на следующем схематичном примере.

```
C:\...\python <Абсолютный_путь_к_файлу_сценария>/<Имя_сценария>.py
```

Рассмотрим следующий код.

```
"""filename: whereami01.py"""
import os, sys
print('My os.getcwd : %s' % os.getcwd())
print('My sys.path  : %s' % sys.path[:6])
input('Press Enter-key to quit')
```

Для улучшения наглядности вывода второй инструкции *print()* необходимо внести в код некоторые изменения – организовать цикл перебора списка строк и вывод строк по отдельности.

```
"""filename: whereami02.py"""
import os, sys
print('My os.getcwd : %s' % os.getcwd())
text = sys.path[:6]
print('My sys.path  : ')
for line in text:
    print(line)
input('Press Enter-key to quit')
```

В случае использования средств модуля *tkinter* код может принять следующий вид.

```
"""filename: whereami03.py"""
import os, sys
from tkinter import Tk, Label, Button, YES, TOP, BOTH
lines = ''  # ссылка на подготавливаемое текстовое сообщение
lines += ('My os.getcwd : %s' % os.getcwd() + '\n')
text = sys.path[:6]
lines += ('My sys.path  :' + '\n')
for line in text:
    lines +=(line + '\n')
# Строка lines содержит всю необходимую информацию
root = Tk()
lbl = Label(root, text=lines, font=('arial', 12)).pack(
    expand=YES, side=TOP, fill=BOTH)
btn = Button(root, text='Quit', command=root.quit).pack()
root.mainloop()
```

Оформление и наглядность сгенерированной оконной формы оставляют желать лучшего, ведь строки, содержащие данные *sys.path* выровнены по

ширине поля, а не по левому краю и это снижает наглядность отображаемых данных. Воспользуемся средствами виджета *Text*.

```
"""filename: whereami04.py"""
import os, sys
from tkinter import *
lines = ''  # ссылка на подготавливаемое текстовое сообщение
lines += ('My os.getcwd : ' + '\n' + os.getcwd() + '\n')
path_lines = sys.path[:6]
lines += ('My sys.path  :' + '\n')
for line in path_lines:
    lines +=(line + '\n')
# Строка lines содержит всю необходимую информацию
root = Tk()
text = Text(root, height=10, width=80)
scroll = Scrollbar(root, command=text.yview)
text.configure(yscrollcommand=scroll.set)
text.insert(END, lines)
text.pack(side=LEFT)
scroll.pack(side=RIGHT, fill=Y)
root.mainloop()
```

Теперь выводимая информация стала обладать приемлемой читаемостью и мы получили инструмент, который потом сможем запускать в параллельном процессе.

Однако вернемся к результату, сформированному в результате работы первого сценария. Обратите внимание, каким именно образом выведены данные о системных путях *Python*: обратные слеши, используемые при записи системных путей на платформе *MS Windows*, продублированы. Напоминаю, что это вызвано тем, что обратный слеш является признаком начала управляющей *Escape* – последовательности и для того, чтобы он был воспринят должным образом, он должен быть продублирован. Именно благодаря дублированию обратные слеши были правильно отображены во всех примерах.

Однако вернемся непосредственно к самой переменной *sys.path*, она может быть использована и модифицирована для её дальнейшего использования из дочерних процессов, которые мы можем запустить ранее рассмотренными средствами. Сама по себе переменная *sys.path* представляет список, а список, как вы должны помнить, относится к категории изменяемых типов данных. Используя встроенный метод *append()*, можно добавить новую запись в такой список. Например:

```
import sys
sys.path.append('C:\\Temp')
```

Аналогичным образом можно удалить некоторую часть информации из *sys.path*, формируя необходимые параметры системного окружения. Это дает

широкие возможности для запуска из *Python* различных системных утилит и автоматизированной обработки результатов работы оных.

- **Аргументы командной строки**

Механизм передачи аргументов командной строки программе-скрипту на языке *Python* реализован средствами модуля *sys*. Правила работы с аргументами командной строки также унаследованы из языка *ANSI C*, хоть и используют типы данных *Python*. Как вы помните, формат функции main(), через которую осуществляется передача параметров в С-программу, подразумевает следующий список параметров: *int argc*, *char** argv*. Наличие первого параметра обусловлено отсутствием возможности передать в функцию количество строк, адресуемых при помощи параметра *argv*. Некоторые могут возразить, что стандарт *POSIX* позволяет обойтись и без *argc*, но данный стандарт появился значительно позже языка *C*.

Из программы на языке *Python* доступ к аргументам командной строки осуществляется через переменную *sys.argv*. Соответственно, аргумент, адресуемый через *sys.argv[0]*, связан с именем программы. Получить доступ к остальным аргументам можно перебираю элементы списка *sys.argv.*

```
>>> import sys
>>> type(sys.argv)
<class 'list'>    # тип параметра - список
>>>
```

Очень часто параметры передаются в формате опций запуска, который определяет следующие правила обработки параметров: `-optionName optionValue` , требующие парсинга полученных параметров и формирования словаря опций. Например, для программы, использующей два файла: один файл в качестве входных данных, а другой – для вывода результата работы, использование опций в командной строке может принять следующий вид:

```
C:\...\python my_program.py –i source.dat –o dest.dat
```

Обратите внимание, что использование лидирующего символа `'-'` (тире) характерно для UNIX/Linux платформ, на платформе Microsoft Windows используется лидирующий символ слеш `'/'` . Изучите справку системной утилиты `'shutdown'` , введя в командной строке `'help shutdown'` .

Теперь снова вернемся к разбору опций в формате *POSIX* и рассмотрим следующий пример.

```
"""filename: testargv.py
Engl.: Collect command-line options in a dictionary
Russ.: Собирает параметра командной строки в словаре
"""
```

```
def getopts(argv):
    opts = {}    # пустой словарь для формирования опций запуска
    while argv:  # цикл перебора элементов списка
        if argv[0][0] == '-':        # find "-name value" pairs
            opts[argv[0]] = argv[1]  # dict key is "-name" arg
            argv = argv[2:]
        else:
            argv = argv[1:]
    return opts

if __name__ == '__main__':
    from sys import argv             # example client code
    myargs = getopts(argv)
    if '-i' in myargs:
        print(myargs['-i'])
    print(myargs)
```

Запустите данный скрипт со следующими опциями:

```
C:\...\python testargv.py –i source.dat –o dest.dat
```

Изучите работу следующего примера, заимствованного с сайта *java2s.com*.

```
"""
This program is part of "Dive Into Python", a free Python book for
experienced programmers.  Visit http://diveintopython.org/ for the
latest version.
"""

__author__ = "Mark Pilgrim (mark@diveintopython.org)"
__version__ = "$Revision: 1.4 $"
__date__ = "$Date: 2004/05/05 21:57:19 $"
__copyright__ = "Copyright (c) 2001 Mark Pilgrim"
__license__ = "Python"

import sys, os

print('sys.argv[0] =', sys.argv[0])
pathname = os.path.dirname(sys.argv[0])
print('path =', pathname)
print('full path =', os.path.abspath(pathname))
```

Стандартные потоки ввода-вывода Python

Операционная система взаимодействует с программами и пользователем посредством различных интерфейсов, среди которых традиционно выделяются стандартные потоки ввода-вывода. Эти потоки обеспечивают передачу данных между программами, равно как и между программами и пользователем. Рассмотрим следующий пример. Сначала запустите его в виде инструкций в *Python IDLE*, затем – в виде отдельной программы.

```
>>> import sys
>>> for f in (sys.stdin, sys.stdout, sys.stderr): print(f)
```

Результат работы в *Python IDLE*:

```
<idlelib.run.PseudoInputFile object at 0x0204AE10>
<idlelib.run.PseudoOutputFile object at 0x0204AE30>
<idlelib.run.PseudoOutputFile object at 0x0204AE50>
>>>
```

Результат работы консольной Python-программы, запущенной под управлением командного интерпретатора *MS Windows 7*:

```
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='cp866'>
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='cp1251'>
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='cp866'>
>>>
```

В обоих случаях выведена информация о том, что стандартным потокам ввода-вывода ставятся в соответствие, то есть присоединяются к ним, специальные объекты. Ранее был пример, показывающий вывод сообщения в консоль не посредством записью данных в стандартный поток вывода.

```
>>> sys.stdout.write("This is output stream")
This is output stream21
>>> sys.stdout.write("This is output stream"+'\n')
This is output stream
22
>>>
```

Выводимое после строкового сообщения числовое значение соответствует количество символов, успешно переданных в стандартный поток вывода.

Для чтения данных из стандартного потока ввода предназначен метод *sys.stdin.readline()*.

```
>>> print("Введите PIN_код : ", end=''); sys.stdin.readline()[:-1]
Введите PIN_код : 1234567
'1234567'
>>>
```

Вспомним полную спецификацию метода *print()*:

```
print( [<объект>][, sep=' '][, end='\n'][, file=sys.stdout])
```

По умолчанию параметр *file* инициализирован ссылкой на стандартный поток вывода *sys.stdout*. Использование данного параметра позволяет программисту организовать процесс перенаправления программного вывода.

Перенаправление стандартных потоков ввода-вывода

Ранее вам уже приходилось осуществлять процедуру перенаправления стандартного потока вывода программы в файл. Осуществлялось это примерно так:

```
[Приглашение оболочки]> Имя_программы > Имя_файла
```

При этом весь вывод, осуществляемый программой в стандартный поток вывода, перенаправлялся в файл. Данный подход может быть использован только на уровне управления процессами, для которых жестко определены правила доступа к общим данным. Такое взаимодействие называется синхронным. Но большинство современных интерактивных программ используют асинхронные механизмы взаимодействия, поэтому вышеприведенный механизм можно использовать только на уровне небольших узкоспециализированных программных сервисов. К таким программным сервисам можно отнести системные команды *ls*, *dir*, *type*, *help*, *cat*, *gcc* и многие другие. Эти команды являются отдельными программными модулями.
Аналогичным образом перенаправляется стандартный поток ввода.

```
[Приглашение оболочки]> Имя_программы < Имя_файла
```

Рассмотрим пример программного перенаправления стандартного потока вывода.

```
import sys
old_output_stream = sys.stdout  # сохраняем ссылку на sys.stdout
fp = open(r'result.txt', 'a')   # создаем файловый поток
sys.stdout = fp  # связываем ссылку на sys.stdout с файловым потоком
print("Software Output to file")
fp.close()
sys.stdout = old_output_stream # восстанавливаем ссылку на sys.stdout
print("Sostware Output to sys.stdout")
```

Данный пример несколько избыточен, рассмотрим его упрощенную версию.

```
import sys
fp = open(r'result1.txt', 'a')
print("Software Output to file", file=fp)
fp.close()
print("Sostware Output to sys.stdout")
```

В третьей строке скрипта методу *print()* явно передается ссылка на созданный файловый поток, а в пятой строке данный метод использует значение по умолчанию – *file=sys.stdout*. Аналогичным образом можно перенаправить и стандартный поток ввода *sys.stdin*. Для этого потребуется создать файловый поток и определить его в режиме чтения – входные данные читаются из файла.

```
import sys
old_stdin = sys.stdin  # сохраняем ссылку sys.stdin
fp = open(r'result1.txt', 'r')  # открываем файловый поток для чтения
sys.stdin = fp  # изменяем значение ссылки sys.stdin
while True:     # цикл построчного чтения данных их файла
    try:                # файловый поток содержит символ EOF
        text = input()  # чтение которого методом input() приводит
        print(text)     # к «выбросу» объекта исключения
    except EOFError:
        break           # достигнут конец файла, прервать цикл
sys.stdin = old_stdin  # восстанавливаем значение ссылки на sys.stdin
fp.close()             # закрываем файловый поток
input(r'Press Enter-key to quit.')
```

Объект *sys.stdin* обладает методом *isatty()*, возвращающим значение *True* в случае, если объект связан с системным стандартным потоком ввода, и *False* – в противном случае.

```
>>> old_stdin = sys.stdin
>>> fp = open('result1.txt', 'r')
>>> sys.stdin.isatty()
True
>>> sys.stdin = fp  # связываем sys.stdin с файловым потоком
>>> sys.stdin.isatty()
False
>>> sys.stdin = old_stdin  # восстанавливаем исходное значение ссылки
>>> fp.close()
>>>
```

Рассмотрим следующий пример, имитирующий процедуру «обратного отсчёта». Подобный подход может быть использован при имитации индикатора степени готовности процесса.

```
import sys, time
print("На старт...")
for i in range(9, -1, -1):
    sys.stdout.write("\rОбратный отсчет : %d" % i)
    sys.stdout.flush()
    time.sleep(1)
sys.stdout.write('\r                        ')
sys.stdout.write('\rПоехали!!!')
```

Как видно из результатов работы примера, запущенного в виде самостоятельной программы, символ `'\r'` позволяет заместить ранее выведенную в *sys.stdout* информацию.

Перенаправление стандартных потоков ввода-вывода является из одних простейших подходов, используемых при построении систем, реализующих архитектуру «клиент-сервер». В упрощенном виде можно сказать, что такие системы реализуют цикл операций «прочесть – вычислить - вывести»

```
"""
Читаем числа - координаты точек (x,y) и проверяет принадлежность
данной точки окружности с координатами центра заданого точкой (A,B)
и радиуса R. Работа программы прекращается при нажатии сочетания
клавиш Ctrl+Z.
"""
def my_process():
    import sys
    print('Проверка -----------------------------------\n'
          'приналдежит ли точка с заданными координатами\n'
          'определенной пользователем окружности.')
    print('-------------------------------------------')
    # Begin - Подготока данных -------------------------------------
    print('Введите координаты центра окружности')
    try:
        A = float(input('X : '))
    except ValueError:
        print('Ошибка ввода данных!')
        sys.exit(1)
    try:
        B = float(input('Y : '))
    except ValueError:
        print('Ошибка ввода данных!')
        sys.exit(1)
    try:
        R = float(input('Величина радиуса R : '))
    except ValueError:
        print('Ошибка ввода данных!')
        sys.exit(1)
    # Start of cycle - Начало цикла обработки данных ---------------
    while True:
        try:
            reply = input('Введите через пробел координаты X и Y : ')
        except EOFError:
            print('Введено Ctrl-Z')
            break  # Пользователь ввел Ctrl+Z - признак
                   # завершения ввода данных
        else:
            (x,y) = reply.split(' ')
            try:
                x = float(x)
```

```
            except ValueError:
                print('Ошибка ввода данных для координаты X')
                continue
            try:
                y = float(y)
            except ValueError:
                print('Ошибка ввода данных для координаты Y')
                continue
            if (x-A)*(x-A) + (y-B)*(y-B) < R*R:
                print('Точка с координатами (%f,%f) принадлежит'
                      ' указанной окружности' % (x,y))
            else:
                print('Точка с координатами (%f,%f) не принадлежит'
                      ' указанной окружности' % (x,y))
            print("Для выхода из цикла нажмите 'Ctrl-Z'")
    # End of cycle - Начало цикла обработки данных ------------------
    print('Bye')
if __name__ == '__main__':
    my_process()
```

Запомните, что при запуске примеров из среды *Python IDLE* происходит изменение «логики работы» оных по причине того, что *IDLE* сама перехватывает исключения и, в ряде случаев, секция *else* оператора *try-except* становится недостижимой. Отлаживать и тестировать программы, использующие механизм перехвата исключений, в среде *Python IDLE* нельзя! К слову, такие же проблемы возникают и при использовании *PyCharm*...

Рассмотрим более простой пример, использующий *sys.stdin*.

```
# filename: test_stream_01.py
def small_process():
    print('Testing stream sys.stdin.')
    while True:
        try:
            reply = input('Enter a number > ')
        except EOFError:
            break
        else:
            num = int(reply)
            print("%d squared is %d" % (num, num*num))
    print('Bye')
if __name__ == '__main__':
    small_process()
```

Для обеспечения нормальной работы данный сценарий необходимо запускать из командной строки:

```
C:\Users\...\python test_stream_01.py
```

Создайте текстовый файл, содержащий три – четыре целочисленные константы, разделенные символом новой строки `'\n'` , сохраните его под названием *'input.txt'*.

Используя механизм перенаправления потока ввода, мы можем организовать следующий вариант запуска данного сценария:

```
C:\Users\...\python test_stream_01.py < input.txt
```

Рассмотрим системную программу *type*, используемую на платформе *Windows*. По функционалу она очень близка команде *cat*, используемой в *UNIX/Linux*. Формат использования *type* можно узнать, введя в командной оболочке команду - `"type /?"`. Вывод будет примерно следующим:

```
Вывод содержимого одного или нескольких текстовых файлов.
TYPE [диск:][путь]имя_файла
```

Данная программа потребуется нам для автоматизации работы со скриптами, использующими ручной ввод данных из консоли, работающими на *MS Windows* платформе.

На уровне командного интерпретатора большинство операционных систем поддерживают две команды перенаправления потоков ввода-вывода `'>'` и `'<'`, а также «конвейер» `'|'`. Синтаксис оператора перенаправления потока вывода выглядит следующим образом: *'программа' > 'имя_файла'*. Командный интерпретатор позволяет объединять перенаправление потоков ввода и вывода в одной команде, например:

```
C:\...\python test_stream_01.py < input.txt > output.txt
```

Результат работы скрипта зависит от того, существовал ли в текущей директории файл *'output.txt'*, если нет, то будет создан новый файл и в него будет осуществлен вывод результата работы данного скрипта. Если файл *'output.txt'* существовал в текущей директории, то результат работы скрипта будет записан в конец данного текстового файла. Если вам необходимо осуществить процесс перезаписи содержимого текстового файла, то необходимо воспользоваться оператором `'>>'`.

```
C:\...\python test_stream_01.py < input.txt >> output.txt
```

Использование каналов для обмена данными между программами

Именованные каналы, или «конвейеры», являются одним из интерфейсов межпроцессного взаимодействия. Именованные каналы организуются с помощью физических файлов, отражаемых на файловую систему. Интерфейсы взаимодействия с именованными каналами в *UNIX/Linux* и *MS Windows* несколько отличаются, но в нашем случае эти различия не критичны.

На уровне командной оболочки программисту доступна системная команда `'|'`, позволяющая направлять стандартный вывод одной программы в

стандартный ввод другой. Данная команда создает «канал» или «конвейер», соединяющий ввод и вывод двух команд.

Изменим предыдущий пример:

```
C:\...\python test_stream_01.py < input.txt >> output.txt
```

перенаправив результат не в файл, а системной утилите *more*:

```
C:\...\python test_stream_01.py < input.txt | more
```

Для дальнейшей работы нам потребуются два вспомогательных скрипта *writer.py* и *reader.py*. Ниже приведен их исходный код.

```
"""filename: reader.py"""
import sys
print('From input stream readed this "%s"' % input())
data = sys.stdin.readline()[:-1]
print('From sys.input readed this "%d"' % int(data))

"""filename: writer.py"""
import os
my_pid = os.getpid()
parent_pid = os.getppid()
print("Hello from priocess number %d" % my_pid)
print(parent_pid)
```

Для закрепления опыта работы с утилитой *type* выполним следующие команды.

```
C:\...\type writer.py
C:\...\type reader.py
```

Теперь соединим работу созданных сценариев с помощью канала.

```
C:\...\python writer.py | python reader.py
```

Теперь, возможно, вы сможете представать способ автоматизации процесса перебора паролей с помощью нескольких программ, соединенных при помощи потоков: одна программа отвечает за диалог с «исследуемой» системой, а другая – обеспечивает доступ к файлу, содержащему словарь паролей.

Рассмотрим несколько примеров, позаимствованных из учебника Марка Лутца [1]

```
"""filename: sorter.py - сортирует данные считанных строк"""
import sys
lines = sys.stdin.readlines()  # читает строки данных из sys.stdin
line.sort()  # сортировка строк встроенным методом
for line in lines:  # цикл перебора считанных и отсортированных строк
    print(line, end='') # вывод строк в стандартный поток вывода
# END -----------------------------------------------------------

"""filename: adder.py
```

```
Description: читает ститанные из стандартного потока
ввода данныеи, преобразует их в целочисленные значения - int().
Возвращет значение суммы считанных значений"""
import sys
sum = 0
while True:
    try:
        line = input()  # или sys.input.readlines()
    except EOFError:    # или for line in sys.stdin:
        break           # input() отсекает символы \n в конце строки
    else:
        sum += int(line)
print(sum)
# END ----------------------------------------------------------------
```

Файл, содержащий исходные данные, может быть создан следующим образом (зачем, спрашивается, мы и учим Python).

```
>>> fp = open('data.txt', 'w')
>>> data = ['123', '000', '999', '042']
>>> for i in data:
     print('%s' % i, file=fp, end='\n')


>>> fp.close()
>>>
```

Хотя, возможно, кому то проще будет запустить «Блокнот» или другой текстовый редактор…

Протестируем созданные нами программные инструменты.

```
C:\...\type data.txt

C:\...\python sorter.py < data.txt

C:\...\python adder.py < data.txt

C:\...\type data.txt | python adder.py
```

Вы уже заметили, что данные, содержащиеся в файле *'data.txt'* представляют собой строковую запись целочисленных констант в формате `'%03d'`. Это обеспечивает возможность использования встроенного метода *sort()* для сортировки таких строк.

Создадим еще один программный инструмент – программу для записи последовательности в стандартный поток вывода.

```
"""filename: writer02.py"""
for data in (123, 0, 999, 42):
    print('%03d' % data)
```

Теперь у нас появилась возможность для следующего варианта использования «конвейера»:

```
C:\...\python writer02.py | python sotrer.py
```

```
C:\...\python writer02.py | pythor sorter.py | python adder.py
```

Литература и источники в Интернет

1. Лутц М. Изучаем Pyhon, 4-е издание. – Пер. с англ. [Текст] – СПб.: СимволПлюс, 2011. – 1280 с.
2. Мэтиз Эрик. Изучаем Python. Программирование игр, визуализация данных, веб-приложения. [Текст] – СПб.: Питер, 2017. – 496 с.
3. Любавич Билл. Простой Python. Современный стиль программирования. [Текст] – СПб.: Питер, 2016. – 480 с.
4. Прохоренок Н.А. Python 3 и PyQt 5. Разработка приложений [Текст] / Н.А. Прохоренок, В.А. Дронов. – СПб.: БХВ-Петербург, 2016. – 832.: ил.