# Деякі загальні методи організації сценаріїв

## Консольні програми і Windows-програми

Сценарій (Python-програму) можна налаштувати для роботи в консольному режимі і в режимі Windows. Різницю між режимами визначає розширення імені файла: .py – консольна програма; .pyw – Windows-програма. При розробці сценаріїв в середовищі IDLE за замовчуванням будують консольну програму. Закривши IDLE, заміну розширення виконують "вручну" в провіднику Windows чи іншим способом.

Такий спосіб виконання використовують для сценаріїв, які виконують операції через функції операційної системи, яка має свої засоби відображення результатів виконання.

## Запуск програм з сценаріїв

Для такого запуску використовують модулі os і sys, які забезпечують більшість інтерфейсів системного рівня. Зокрема, модуль os має величини і функції, які відповідають операційній системі, в якій виконується Python. Модуль os намагається також надати мобільний інтерфейс програмування для поточної операційної системи. Його функції можуть бути по-різному реалізовані на різних платформах, але для сценаріїв Python вони виглядають однаково. Крім того, модуль os має вкладений підмодуль os.path, який надає мобільний інтерфейс для опрацювання файлів і каталогів. Отже, треба підключити `import os`.

Запуск файла зі зв'язаною аплікацією. Такий запуск виконує метод

```
os.startfile( path [, operation] )
```

Наприклад:

```
os.startfile("pythonds.txt")  # за замовчуванням  operation = 'open'
```

Якщо *operation*='open', тоді цей метод працює аналогічно до подвійного натискання мишкою на імені файла в провіднику Windows. Аплікація виконується асинхронно в паралельному потоці, сценарій Python не чекає закриття аплікації і продовжує свою роботу. Отже, після цього можна запустити на виконання одночасно інші програми.

Параметр *path* є відносним до поточної папки. Його можна обчислювати різними способами. Наприклад, якщо вказаний файл розташований в поточній папці, можна записати так:

```
filename = os.path.join(os.path.abspath('.'), 'pythonds.txt')
os.startfile(filename)
```

Насамкінець зазначимо, що цей метод доступний лише для Windows.

Передача керування іншій програмі. Сценарій може передати керування іншій сторонній exe-програмі (власній або системній) припинивши при цьому поточний процес – дочірній чи основний. Для цього є група методів модуля os:

```
os.execl(path, arg0, arg1, ...)
os.execv(path, args)
os.execlp(file, arg0, arg1, ...)
os.execvp(file, args)
```

а також інші – див. документацію. Буква "l" означає, що треба явно записати всі параметри, які будуть передані викликаній програмі через командний рядок операційної системи. Буква "v" означає, що кількість параметрів для передачі програмі може бути змінною, і їх треба позначити як список чи кортеж. Буква "p" означає, що за потреби використати *PATH environment variable* для пошуку програмного файла. Це стосується системних програм і прикладних програм, зафіксованих в реєстрі Windows.

Приклади:

```
calc = "d:\Temp_\Калькулятор комплексних чисел\PrjCompl.exe"
os.execl(calc, " empty")  # перед empty пропуск
```

Запускається на виконання власна програма-калькулятор. Переважно в таких випадках треба визначати повний шлях до файла програми. Якщо б програма була зафіксована в реєстрі Windows, тоді можна було б записати:

```
os.execlp("PrjCompl.exe", " empty")
```

Зауважимо, що наша програма не має зовнішніх параметрів для запуску, проте формально треба позначити другий параметр `" empty"` методу `os.execl()` чи `os.execlp()`.

```
filename = os.path.join(os.path.abspath('.'), 'pythonds.txt')
os.execlp( "notepad.exe", " ", filename )
```

Обчислюють шлях до власного текстового файла `pythonds.txt` і безпосередньо викликають системну програму `notepad.exe`. Якщо записник (чи іншу системну програму) запускають з параметрами командного рядка, тоді системна програма зразу відкриває зазначений файл. Зауважимо, що між іменем програми та іменем файла обов'язково треба записати пропуск, бо він автоматично не дописується.

Якщо ж записати

```
os.execlp( "notepad.exe", " " )  # між лапками обов'язково пропуск !
```

тоді запускається `notepad.exe` в режимі будови нового файла. Проте треба бути уважним: запис

```
os.execlp( "notepad.exe", " empty" )
```

трактують як команду програмі створити новий файл з таким іменем.

```
filename = os.path.join(os.path.abspath('.'), 'pythonds.txt')
os.execvp( "notepad.exe", [" ", filename] )  # буква v
```

Те ж саме, що й `os.execlp()`, лише параметри позначені як єдиний список.

**Дочірні процеси**

Сценарій, який працює в потоці і є окремим процесом, може запустити на виконання дочірній (підлеглий) процес. Для цього використовують модуль subprocess. Цей модуль забезпечує виконання інших програм, але не окремих функцій сценарію.

Взагалі, треба відрізняти два терміни: дочірній процес і потік виконання. *Дочірній процес* забезпечує виконання незалежних програм, а *потік виконання* забезпечує виконання функцій або інших об'єктів того самого сценарію в межах одного процесу. Обидва працюють паралельно до процесу чи потоку.

Є два способи виконання дочірнього процесу.

1) subprocess.run(args,...) – запустити процес і чекати його закінчення, після чого продовжити виконання основного процесу.

2) class subprocess.Popen(args,...) – створити процес, запустити на виконання асинхронно (паралельно), повернутись в основний процес і мати можливість контролю дочірнього процесу.

Повне визначення метода і класу потрібно переглянути за довідковою системою.

Основний і дочірній процеси є незалежні. Прямого зв'язку між ними немає. Зв'язок можна організувати через треті об'єкти, наприклад, через файли. При цьому кожен процес має в загальному випадку щоразу відкривати і закривати спільний файл.

Зауважимо, що в мові Python немає засобів організації спільної внутрішньої пам'яті декількох процесів. Якщо ж все таки потрібна спільна внутрішня пам'ять чи синхронізація, тоді замість процесу треба будувати потік і використати модулі _thread або threading.

В стандартному модулі _thread не використовують прийоми ООП і він дуже простий для використання. Модуль threading кращий для розв'язування складніших задач, які вимагають зберігання даних в контексті потоків або спостереження за потоками. Рекомендують використовувати модуль threading

### Очікування закінчення дочірнього процесу

```
#  Основні і дочірні процеси: чекати закінчення дочірнього
#  тут є основний процес

import subprocess
# запуск дочірнього процесу: приклад
#  subprocess.run(["notepad.exe", " empty"])  # можна запускати exe-програми

fn = "comdata.txt"  # спільний файл даних
ab = [ 4,5,6,0,-1,-2,-3 ]  # початковий список
print("main process - step 1:\n", ab)
# записати у файл
with open(fn,"w") as data :
    data.write("  ".join(map(str,ab)) + "\n")

# можна запускати інші Python-сценарії

import os  # треба обчислювати повних шлях до модуля дочірнього процесу
dirname = os.path.abspath('.')  # шлях від кореня до поточної папки
filename = os.path.join(dirname, 'L19a_3.py')

subprocess.run(["python", filename])  # дочірній процес - паралельний сценарій
# для режиму Windows замість "python" потрібно "pythonw"
# повернення до основного процесу - після закінчення дочірнього
# додати до значень 1
with open(fn,"r") as subdata :
  ef = [ int(x) for x in subdata.readline().rstrip().split() ]
for k in range(len(ef)) : ef[k] += 1
print("main process - step 2:\n", ef)
# записати у файл
with open(fn,"w") as data :
    data.write("  ".join(map(str,ef)) + "\n")

print("to end main process press Enter")
input()  # для виконання в консольному режимі
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

# модуль дочірнього процесу: файл  L19a_3.py
# помножити значення елементів на 2
fn = "comdata.txt"  # спільний файл даних
with open(fn,"r") as subdata :
  cd = [ int(x) for x in subdata.readline().rstrip().split() ]
for k in range(len(cd)) :  cd[k] *= 2
print("subprocess:\n", cd)
with open(fn,"w") as subdata :
  subdata.write("  ".join(map(str,cd)) + "\n")
print("to end subprocess press Enter")
input()
```

```
main process - step 1:
 [4, 5, 6, 0, -1, -2, -3]
subprocess:
 [8, 10, 12, 0, -2, -4, -6]
to end subprocess press Enter

main process - step 2:
 [9, 11, 13, 1, -1, -3, -5]
to end main process press Enter
```

Останні показані результати виконання потоків в консольному режимі, всі виведення виконують у вікно консолі.

**Асинхронне виконання процесів без очікування**

Дочірній процес можна створити і керувати ним за допомогою класу Popen:

```
class subprocess.Popen(args, . . . )
```

При роботі у Windows клас використовує Windows `CreateProcess()` функцію. Перший параметр `args` визначають подібно як у `subprocess.run`, це може бути список з імені програми та її параметрів, або єдиний текстовий рядок запуску програми у Windows.

Зі списку параметрів цього класу, можливо, буде корисним параметр *cwd=None*. Якщо *cwd* не дорівнює *None*, перед виконанням дочірнього процесу функція Popen перемикає робочу директорію на *cwd*. Параметр *cwd* має бути рядком str, в якому записаний path-like object.

Задачі, для яких можна виконати поділ на паралельні асинхронні процеси, мають врахувати, що виконання процесів відбувається одночасно. При цьому швидкість виконання кожного процесу невідома, отже немає підстав до використання спільних несинхронізованих об’єктів процесів.

Нижче подано тексти модельних сценаріїв основного і дочірнього процесу. Моделі не виконують жодної роботи, лише демонструють спосіб запуску і паралельність виконання, для чого введені паузи виконання. Для коректного спостереження виконання основний процес (модуль L19a_4.py) потрібно запускати не в середовищі IDLE, а через провідник Windows (варіант виконання 2, описаний на початку лекції).

```
# модуль основного процесу:  файл L19a_4.py
#  Основні і дочірні процеси: асинхронне виконання дочірнього
import subprocess
import time
print("main process - started")  # початок основного процесу
# . . . . .
# запускаємо паралельний дочірній процес
import os  # краще обчислювати повних шлях до модуля дочірнього процесу
dirname = os.path.abspath('.')  # шлях від кореня до поточної папки
filename = os.path.join(dirname, 'L19a_5.py')
parprog = subprocess.Popen(["python", filename])
# після запуску основний процес продовжує виконання
time.sleep(2)  # затримати виконання на 2 секунди
print("main process - continue")  # продовження основного процесу
# . . . . .
# перевірити стан дочірнього процесу
print( "subprocess finished ? ", parprog.poll() )
print("main process - finished")
input()  # пауза для перегляду результатів
pass
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
# модуль дочірнього асинхронного процесу: файл  L19a_5.py
import time
print("subprocess started")
# . . . . .
time.sleep(3)  # затримати виконання на 3 секунди
print("subprocess finished")
pass
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
```

```
main process - started
subprocess started
main process - continue
subprocess finished ?  None
main process - finished
subprocess finished
```

Останні показані результати виконання процесів.