Урок 1

# Полезные модули

Модули: subprocess, os, ipaddress, tabulate, pprint.

В данном разделе рассмотрим дополнительные модули, которые импортируются в файлы с программным кодом Python и позволяют реализовывать полезную функциональность.

# Модуль subprocess

## Особенности использования модуля subprocess

Модуль **subprocess** содержит функции и классы, обеспечивающие универсальный интерфейс для создания новых процессов, управления потоками ввода и вывода и обработки кодов возврата. Он объединяет многие функциональные возможности, присутствующие в модулях **os**, **popen2** и **commands**.

- **Popen(args, **parms)** — выполняет команду, запуская новый дочерний процесс, и возвращает объект класса **Popen**, представляющий его. Команда определяется в аргументе **args** либо как строка вида **"ls -l"**, либо как список строк, такой как **["ls", "-l"]**. В аргументе **parms** передается коллекция именованных аргументов, которые используют для управления характеристиками дочернего процесса. Перечень допустимых именованных аргументов:

| Именованный аргумент | Описание |
|---|---|
| bufsize | Определяет режим буферизации, где значение 0 соответствует ее отсутствию, 1 — выполняется построчная буферизация. При отрицательном значении используются системные настройки. Любое другое положительное значение определяет примерный размер буфера. По умолчанию используется значение 0. |
| close_fds | Если имеет значение **True**, перед запуском дочернего процесса все дескрипторы файлов, кроме 0, 1 и 2, закрываются. По умолчанию используется значение **False**. |
| creation_flags | Определяет флаги создания процесса в Windows. В настоящее время доступен только один флаг — **CREATE_NEW_CONSOLE**. По умолчанию используется значение 0. |
| cwd | Каталог, в котором будет запущена команда. Перед запуском текущим рабочим каталогом дочернего процесса назначается **cwd**. По умолчанию установлено значение **None**, которое соответствует использованию текущего рабочего каталога родительского процесса. |
| env | Словарь с переменными окружения для нового процесса. По умолчанию установлено значение **None**, которое соответствует использованию окружения родительского процесса. |
| executable | Определяет имя выполняемой программы. Используется редко, так как имя уже включено в **args**. Если в этом параметре определить имя командной оболочки, команда будет запущена в ней. По умолчанию используется значение **None**. |
| preexec_fn | Определяет функцию в дочернем процессе, которая должна быть вызвана перед запуском команды. У нее не должно быть входных аргументов. |

| | |
|---|---|
| shell | Если значение **True**, команда выполняется в командной оболочке UNIX с помощью функции **os.system()**. По умолчанию используется командная оболочка **/bin/sh**, но можно указать другую в параметре **executable**. По умолчанию используется значение **None**. |
| startupinfo | Определяет флаги запуска процесса в Windows. По умолчанию задано **None**. В число допустимых значений входят **STARTF_USESHOWWINDOW** и **STARTF_USESTDHANDLERS**. |
| stderr | Объект файла, который будет использоваться дочерним процессом как поток **stderr**. В этом параметре допускается передавать объект файла, созданный с помощью функции **open()**, целочисленный дескриптор файла или специальное значение **PIPE**. Оно указывает, что необходимо создать новый неименованный канал. По умолчанию используется значение **None**. |
| stdin | Объект файла, который будет использоваться дочерним процессом как поток **stdin**. В этом параметре допускается передавать те же значения, что и в параметре **stderr**. По умолчанию используется значение **None**. |
| stdout | Объект файла, который будет использоваться дочерним процессом как поток **stdout**. В этом параметре допускается передавать те же значения, что и в **stderr**. По умолчанию — None. |
| universal_newlines | Если имеет значение **True**, файлы, представляющие потоки **stdin**, **stdout** и **stderr**, открываются в текстовом режиме с поддержкой универсального символа перевода строки. |

- **call(args, **parms)** — у этой функции те же действия, что и **Popen()**, но она просто выполняет команду и возвращает код завершения (не возвращает объект класса **Popen**). Ее удобно использовать, когда требуется только выполнить команду и нет необходимости получать от нее вывод или управлять ею. Аргументы имеют тот же смысл, что и в функции **Popen()**;

- **check_call(args, **parms)** — то же, что и **call()**, но при получении ненулевого кода завершения вызывает исключение **CalledProcessError**. Код завершения сохраняется в атрибуте исключения **returncode**;

- **run(args, *, stdin=None, input=None, stdout=None, stderr=None, shell=False, timeout=None, check=False, encoding=None, errors=None)** — упрощенный способ создавать процессы. Запускает процесс, ждет его завершения, возвращает объект **CompletedProcess**. Добавлен в Python 3.5.

Объект **p** класса **Popen**, возвращаемый функцией **Popen()**, обладает методами и атрибутами, которые используют для взаимодействия с дочерним процессом.

- **p.communicate([input])** — передает данные **input** на стандартный ввод дочернего процесса и ожидает его завершения. При этом продолжает принимать от него данные, которые выводятся в его потоки стандартного вывода и стандартного вывода сообщений об ошибках. Возвращает кортеж (**stdout, stderr**), где поля **stdout** и **stderr** являются строками. Если не требуется передавать данные дочернему процессу, аргумент **input** можно установить в значение **None** (по умолчанию);

- **p.kill()** — принудительно завершает дочерний процесс. Для этого в UNIX посылается сигнал **SIGKILL**, а в Windows вызывается метод **p.terminate()**;

- **p.poll()** — проверяет, завершился ли дочерний процесс. Если да, возвращает код завершения. В противном случае — **None**;
- **p.send_signal(signal)** — посылает сигнал дочернему процессу. В аргументе **signal** передается номер сигнала, как определено в модуле **signal**. В Windows поддерживается единственный сигнал **SIGTERM**;
- **p.terminate()** — принудительно завершает дочерний процесс, посылая ему сигнал **SIGTERM** в UNIX или вызывая **Win32-API**-функцию **TerminateProcess** в Windows;
- **p.wait()** — ожидает завершения дочернего процесса и возвращает код завершения;
- **p.pid** — целочисленный идентификатор дочернего процесса;
- **p.returncode** — код завершения дочернего процесса. Значение **None** свидетельствует, что дочерний процесс еще не завершился. Отрицательное значение указывает, что он завершился в результате получения сигнала **(UNIX)**;
- **p.stdin, p.stdout, p.stderr** — эти три атрибута представляют открытые объекты файлов. Они соответствуют потокам ввода-вывода, открытым как неименованные каналы (например, установкой параметра **stdout** функции **Popen()** в значение **PIPE**). Эти объекты файлов позволяют подключаться к другим дочерним процессам. Атрибуты получают значение **None**, когда каналы не используются.

Рассмотрим несколько примеров создания дочерних процессов и взаимодействия с ними (для ОС UNIX) (**листинг 1**):

```
import subprocess

# Выполнить простую системную команду с помощью os.system()
ret = subprocess.call("ls -l", shell=True)

# Выполнить простую команду, игнорируя все, что она выводит
ret = subprocess.call("rm -f *.tmp", shell=True, stdout=open("/dev/null"))

# Выполнить системную команду, но сохранить ее вывод
p = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
out = p.stdout.read()

# Выполнить команду, передать ей входные данные и сохранить вывод
p = subprocess.Popen("wc", shell=True, stdin=subprocess.PIPE,
                           stdout=subprocess.PIPE, stderr=subprocess.PIPE)
s = b"Hello world!"
out, err = p.communicate(s) # Передать строку s дочернему процессу

# Создать два дочерних процесса и связать их каналом
p1 = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
p2 = subprocess.Popen("wc", shell=True, stdin=p1.stdout, stdout=subprocess.PIPE)
out = p2.stdout.read()
```

Следующий пример выполняет архивирование файлов (7-Zip) с указанным расширением (для Windows) (**листинг 2**):

```
import os
from subprocess import Popen, CREATE_NEW_CONSOLE

# Создаем кортеж расширений файлов, которые будут нужны
EXT = ('.py', '.PY')

# Создаем список файлов с нужным расширением в текущей директории
files = [f.name for f in os.scandir() if f.is_file() and f.name.endswith(EXT)]
print('Файлы для упаковки:', files)

# Для Windows флаг CREATE_NEW_CONSOLE укажет создать новую консоль для
процесса
packer = Popen(['7z','a','test.zip', *files],
               creationflags=CREATE_NEW_CONSOLE)
# Ждем завершения процесса, чтобы что-то делать дальше...
packer.wait()

print("Файлы упакованы, можно переименовывать")
# Переименовываем файл, созданный архиватором
os.rename('test.zip', 'backup.zip')
```

Поскольку объект класса **Popen** поддерживает протокол менеджера контекста, запуск и ожидание завершения процесса можно выполнить так:

```
with Popen(['7z','a','test.zip', *files],
           creationflags=CREATE_NEW_CONSOLE) as packer:
    print("Ждем упаковку...")
```

Простое создание процесса с ожиданием его завершения можно выполнить функцией **run**:

```
import subprocess

p = subprocess.run(['python', '-V'], stdout=subprocess.PIPE)
print(p.stdout)
```

## Примечания

Команды лучше передавать в виде списка строк, а не единственной строки: **["wc", "filename"]** вместо **"wc filename"**.

Многие системы позволяют использовать пробелы и другие неалфавитные символы в именах файлов (например, папка **Documents and Settings** в Windows). Если команда определяется в виде списка строк, проблем возникать не будет. Чтобы сформировать команду, где используются подобные имена файлов, надо экранировать специальные символы и пробелы.

В Windows каналы открываются в двоичном режиме. Текст, прочитанный из потока стандартного вывода дочернего процесса, будет содержать дополнительный символ возврата каретки на концах

строк (**"\r\n"** вместо **"\n"**). Если это может вызывать проблемы, функции **Popen()** следует передавать значение **True** в параметре **universal_newlines**.

Модуль **subprocess** нельзя использовать для управления дочерними процессами, которые предполагают возможность прямого доступа к терминалу или к устройству **TTY**. Наиболее типичные примеры таких программ — любые команды, ожидающие ввода пароля пользователя (**ssh**, **ftp**, **svn** и другие). Для управления ими рекомендуется искать сторонние модули, основанные на популярной утилите **Expect** (есть в системе UNIX).

# Модуль os

Служебный модуль для операций с файловой системой и окружением, управления процессами. Список его наиболее полезных команд:

1. **mkdir** — создание каталога (**листинг 3**):

```
import os
# Создания каталога в текущей директории
os.mkdir('test_dir')
```

В модуле реализована проверка на существование каталога с указанным именем в текущей директории. При попытке дублирования каталога генерируется ошибка:

```
FileExistsError: [WinError 183] Невозможно создать файл, так как он уже существует.
```

2. **path.exists**

Во избежание ошибки дублирования можно воспользоваться конструкцией проверки существования каталога:

```
if not os.path.exists('test_dir'):
    os.mkdir('test_dir')
```

3. **listdir**

Просмотр содержимого текущей директории:

```
In [1]: dir_struct = os.listdir('.')
In [2]: print(dir_struct)
```

Результат:

```
Out[2]: ['os.py', 'test_dir']
```

4. **path.isdir**

Проверка на объект-каталог. Пример использования команды — получение списка каталогов для текущей директории:

```
In [3]: dirs = [d for d in os.listdir('.') if os.path.isdir(d)]
In [4]: print(dirs)
```

Результат:

```
Out[4]: ['d_1', 'd_2', 'd_3']
```

5. **path.isfile**

Проверка на объект-файл. Пример — получение списка файлов для текущей директории:

```
In [5]: fls = [f for f in os.listdir('.') if os.path.isfile(f)]
In [6]: print(fls)
```

Результат:

```
Out[6]: ['f_1.txt', 'f_2.txt', 'f_3.txt', 'os.py']
```

6. **path.basename**

Определение базового имени пути:

```
In [7]: base_path = os.path.basename('c:\\system\\apps\\Python\\Python.app')
In [8]: print(base_path)
```

Результат:

```
Out[8]: 'Python.app'
```

7. **path.dirname**

Определение имени директории указанного пути:

```
In [9]: dir_path = os.path.dirname ('c:\\system\\apps\\Python\\Python.app')
In [10]: print(dir_path)
```

Результат:

```
Out[10]: 'c:\\system\\apps\\Python'
```

8. **path.split**

Разбивает путь к файлу на путь к родительской папке и название файла, возвращает кортеж соответствующих строк:

```
In [11]: dir_tuple = os.path.split('c:\\system\\apps\\Python\\Python.app')
In [12]: print(dir)
```

Результат:

```
Out[12]: 'c:\\system\\apps\\Python\\', 'Python.app'
```

# Модуль ipaddress

Предназначен для операций с IP-адресами. Он поддерживается в Python, начиная с версии 3.3.

## Функция ip_address()

Отвечает за создание IPv4-адреса. Простейший пример создания адреса (**листинг 4**):

```
In [13]: ipv4 = ipaddress.ip_address('192.168.0.1')
In [14]: print(ipv4)
```

Результат:

```
Out[14]: 192.168.0.1
```

У этого объекта-адреса есть набор методов и атрибутов, которые позволяют проводить с объектом операции. С помощью атрибутов семейства **_is** проверяется диапазон, к которому принадлежит адрес.

1. **is_loopback** — возвращает значение **True**, если обнаруживает loopback-адрес.
2. **is_multicast** — возвращает **True**, если обнаруживает multicast-адрес.
3. **is_reserved** — возвращает **True**, если обнаруживает **IETF**-зарезервированный адрес.
4. **is_private** — возвращает **True**, если адрес выделен для частных сетей.

Примеры использования атрибутов:

```
print(ipv4.is_loopback)
print(ipv4.is_multicast)
print(ipv4.is_reserved)
print(ipv4.is_private)
```

Результат:

```
False
False
False
True
```

С объектами-адресами можно выполнять операции:

1. Сравнение ip-адресов.
2. Конвертация ip-адреса в строковое представление.

3. Конвертация ip-адреса в целочисленное представление.

4. Изменение идентификатора узла в сети.

```
ip1 = ipaddress.ip_address('192.168.1.0')
ip2 = ipaddress.ip_address('192.168.1.255')
if ip2 > ip1:
    print(True)
print(str(ip1))
print(int(ip1))
print(ip1 + 5)
print(ip1 - 5)
```

Результат:

```
192.168.1.0
3232235776
192.168.1.5
192.168.0.251
```

## Функция ip_network()

Отвечает за создание объекта, описывающего сеть (**IPv4** или **IPv6**).

Как и для объекта-адреса, для объекта-сети предусмотрены атрибуты и методы для выполнения операции с объектом-сетью (**листинг 4)**.

1. **Атрибут получения широковещательного адреса для сети — broadcast_address**:

```
subnet = ipaddress.ip_network('80.0.1.0/28')
ba = subnet.broadcast_address
print(ba)
```

Результат:

```
80.0.1.15
```

2. **Просмотр всех хостов для объекта-сети — метод hosts()**:

```
print(list(subnet.hosts()))
```

Результат:

```
[IPv4Address('80.0.1.1'), IPv4Address('80.0.1.2'), IPv4Address('80.0.1.3'),
IPv4Address('80.0.1.4'), IPv4Address('80.0.1.5'), IPv4Address('80.0.1.6'),
IPv4Address('80.0.1.7'), IPv4Address('80.0.1.8'), IPv4Address('80.0.1.9'),
IPv4Address('80.0.1.10'), IPv4Address('80.0.1.11'), IPv4Address('80.0.1.12'),
IPv4Address('80.0.1.13'), IPv4Address('80.0.1.14')]
```

3. **Разбиение сети на подсети (по умолчанию — на две) — метод subnets()**:

```
print(list(subnet.subnets()))
```

Результат:

```
[IPv4Network('80.0.1.0/29'), IPv4Network('80.0.1.8/29')]
```

4. **Обращение к любому адресу в сети.** Объект-сеть в Python представляется в виде списка ip-адресов, к каждому из которых можно обратиться по индексу:

```
print(subnet[1])
```

Результат:

```
80.0.1.1
```

## Функция ip_interface()

Отвечает за создание объектов **IPv4Interface** или **IPv6Interface** (**листинг 4**):

```
ipv4_int = ipaddress.ip_interface('10.0.1.1/24')
```

Получение адреса, маски, сети интерфейса:

```
# Получение адреса
print(ipv4_int.ip)
# Получение маски
print(ipv4_int.netmask)
# Получение сети
print(ipv4_int.network)
```

Результат:

```
10.0.1.1
255.255.255.0
10.0.1.0/24
```

# Пример работы с модулем ipaddress

Программный код, выполняющий проверку типа адреса (адрес хоста или сети) (**листинг 5**):

```
import ipaddress
```

```
ip_1 = '10.0.1.1/24'
ip_2 = '10.0.1.0/24'

def ip_network_check(ip_addr):
    try:
        ipaddress.ip_network(ip_addr)
        return True
    except ValueError:
        return False
print(ip_network_check(ip_1))
print(ip_network_check(ip_2))
```

Результат:

```
False
True
```

# Модуль tabulate

## Введение в модуль table

Реализует возможности табличного отображения данных. Не включен в стандартную библиотеку Python и требует дополнительной установки с помощью команды:

```
pip install tabulate
```

В модуле реализована поддержка следующих типов табличных данных:

1. Список списков.
2. Список словарей, где ключи — имена столбцов.
3. Словарь с любыми объектами, поддерживающими итерирование, где ключи — имена столбцов.

Таблица генерируется с помощью функции **tabulate** этого модуля. Пример представления списка кортежей с помощью функции **tabulate** (**листинг 6**):

```
from tabulate import tabulate
tuples_list = [('Python', 'interpreted', '1991'),
               ('JAVA', 'compiled', '1995'),
               ('C', 'compiled', '1972')]
print(tabulate(tuples_list))
```

Результат:

```
------  -----------  ----
Python  interpreted  1991
JAVA    compiled     1995
C       compiled     1972
```

```
------  -----------  ----
```

Чтобы определить заголовки, необходимо указать параметр **headers** и передать в него их набор:

```
columns = ['programming language', 'type', 'year']

print(tabulate(tuples_list, headers=columns))
```

Результат:

```
programming language    type           year
----------------------  -----------  ------
Python                  interpreted    1991
JAVA                    compiled       1995
C                       compiled       1972
```

Также набор заголовков можно определить как первую строку набора данных и сделать соответствующее указание, запуская создание таблицы (**листинг 7**):

```
columns = ['programming language', 'type', 'year']

print(tabulate(tuples_list, headers='firstrow'))
```

Если данные — список словарей, значением параметра **headers** является оператор **keys** (**листинг 8**):

```
print(tabulate(dicts_list, headers='keys'))
```

Результат:

```
programming language      year  type
----------------------  ------  -----------
Python                    1991  interpreted
JAVA                      1995  compiled
C                         1972  compiled
```

## Стилизация таблиц

В модуле **tabulate** поддерживается несколько вариантов оформления табличного представления данных.

Grid-формат (**листинг 9**):

```
print(tabulate(dicts_list, headers='keys', tablefmt="grid"))
```

Результат:

```
+-------------+--------+-----------------------+
```

```
| type        |   year | programming language   |
+=============+========+========================+
| interpreted |   1991 | Python                 |
+-------------+--------+------------------------+
| compiled    |   1995 | JAVA                   |
+-------------+--------+------------------------+
| compiled    |   1972 | C                      |
+-------------+--------+------------------------+
```

Markdown-формат:

```
print(tabulate(dicts_list, headers='keys', tablefmt="pipe"))
```

Результат:

```
| programming language   |   year | type        |
|:-----------------------|-------:|:------------|
| Python                 |   1991 | interpreted |
| JAVA                   |   1995 | compiled    |
| C                      |   1972 | compiled    |
```

HTML-формат:

```
print(tabulate(dicts_list, headers='keys', tablefmt="html"))
```

Результат:

```
<table>
<thead>
<tr><th style="text-align: right;">  year</th><th>type       </th><th>programming
language  </th></tr>
</thead>
<tbody>
<tr><td style="text-align: right;">  1991</td><td>interpreted</td><td>Python
</td></tr>
<tr><td style="text-align: right;">  1995</td><td>compiled   </td><td>JAVA
</td></tr>
<tr><td style="text-align: right;">  1972</td><td>compiled   </td><td>C
</td></tr>
</tbody>
</table>
```

## Выравнивание столбцов

Свойства выравнивания определяются параметром **stralign** (**листинг 10**):

```
print(tabulate(dicts_list, headers='keys', tablefmt="pipe", stralign="center"))
```

Результат:

```
|  programming language  |   year |    type     |
|:----------------------:|-------:|:-----------:|
|         Python         |   1991 | interpreted |
|          JAVA          |   1995 |  compiled   |
|           C            |   1972 |  compiled   |
```

# Модуль pprint

Улучшает качество отображения Python-объектов, сохраняя структуру исходных данных. Входит в стандартную библиотеку Python и не требует дополнительной установки. Самый распространенный способ применения модуля — задействовать его функцию **pprint**.

Отображение словаря с вложенными словарями (**листинг 11**):

```
from pprint import pprint

dict_dicts = {'el_1': {'el_1.1': 'val_1.1', 'el_1.2': 'val_1.2', 'el_1.3':
'val_1.3'},
'el_2': {'el_2.1': 'val_2.1', 'el_2.2': 'val_2.2', 'el_2.3': 'val_2.3'},
'el_3': {'el_3.1': 'val_3.1', 'el_3.2': 'val_3.2', 'el_3.3': 'val_3.3'}}

pprint(dicts_list)
```

Результат (вместо вывода строки — упорядоченное представление вложенных словарей):

```
{'el_1': {'el_1.1': 'val_1.1', 'el_1.2': 'val_1.2', 'el_1.3': 'val_1.3'},
 'el_2': {'el_2.1': 'val_2.1', 'el_2.2': 'val_2.2', 'el_2.3': 'val_2.3'},
 'el_3': {'el_3.1': 'val_3.1', 'el_3.2': 'val_3.2', 'el_3.3': 'val_3.3'}}
```

Отображение строки:

```
str_pp = '\n programming language Python\n type interpreted\n year 1991\n license
free \n'
pprint(str_pp)
```

Результат:

```
('\n'
 ' programming language Python\n'
 ' type interpreted\n'
 ' year 1991\n'
 ' license free \n')
```

# Практическое задание

1. Написать функцию **host_ping()**, в которой с помощью утилиты **ping** будет проверяться доступность сетевых узлов. Аргументом функции является список, в котором каждый сетевой узел должен быть представлен именем хоста или ip-адресом. В функции необходимо перебирать ip-адреса и проверять их доступность с выводом соответствующего сообщения («Узел доступен», «Узел недоступен»). При этом ip-адрес сетевого узла должен создаваться с помощью функции **ip_address()**.

2. Написать функцию **host_range_ping()** для перебора ip-адресов из заданного диапазона. Меняться должен только последний октет каждого адреса. По результатам проверки должно выводиться соответствующее сообщение.

3. Написать функцию **host_range_ping_tab()**, возможности которой основаны на функции из примера 2. Но в данном случае результат должен быть итоговым по всем ip-адресам, представленным в табличном формате (использовать модуль **tabulate**). Таблица должна состоять из двух колонок и выглядеть примерно так:

| Reachable | Unreachable |
|---|---|
| 10.0.0.1 | 10.0.0.3 |
| 10.0.0.2 | 10.0.0.4 |

4. Продолжаем работать над проектом «Мессенджер»:

    a. Реализовать скрипт, запускающий два клиентских приложения: на чтение чата и на запись в него. Уместно использовать модуль **subprocess**);

    b. Реализовать скрипт, запускающий указанное количество клиентских приложений.

5. *В следующем уроке мы будем изучать дескрипторы и метаклассы. Но вы уже сейчас можете перевести часть кода из функционального стиля в объектно-ориентированный. Создайте классы «Клиент» и «Сервер», а используемые функции превратите в методы классов.

# Дополнительные материалы

1. ipaddress — работаем с IPv4/v6.
2. Python. Краткий обзор стандартной библиотеки.
3. Модуль subprocess — работаем с процессами.
4. Примеры использования модуля os в Python.

# Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:


1. Python 3 для сетевых инженеров.
2. David Beazley, Brian K. Jones. Python Cookbook. Third Edition (каталог «Дополнительные материалы»).
3. Лучано Ромальо. Python. К вершинам мастерства (каталог «Дополнительные материалы»).
4. Модуль os.path.
5. ipaddress — Ipv4/IPv6 manipulation library.

# Приложение

**Листинг 1**

```
# ======================== Потоки и многозадачность
============================
# ------------------ Обзор возможностей модуля subprocess (UNIX)
--------------

import subprocess

# Выполнить простую системную команду с помощью os.system()
ret = subprocess.call("ls -l", shell=True)

# Выполнить простую команду, игнорируя все, что она выводит
ret = subprocess.call("rm -f *.tmp", shell=True, stdout=open("/dev/null"))

# Выполнить системную команду, но сохранить ее вывод
p = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
out = p.stdout.read()

# Выполнить команду, передать ей входные данные и сохранить вывод
p = subprocess.Popen("wc", shell=True, stdin=subprocess.PIPE,
                           stdout=subprocess.PIPE, stderr=subprocess.PIPE)
out, err = p.communicate(s) # Передать строку s дочернему процессу

# Создать два дочерних процесса и связать их каналом
p1 = subprocess.Popen("ls -l", shell=True, stdout=subprocess.PIPE)
p2 = subprocess.Popen("wc", shell=True, stdin=p1.stdout, stdout=subprocess.PIPE)
out = p2.stdout.read()
```

**Листинг 2**

```
# ======================== Потоки и многозадачность
============================
# ------------------ Обзор возможностей модуля subprocess
---------------------

# Модуль subprocess содержит функции и классы, обеспечивающие
# универсальный интерфейс для выполнения таких задач, как создание новых
процессов,
# управление потоками ввода и вывода и обработка кодов возврата.

import os
from subprocess import run, Popen, CREATE_NEW_CONSOLE

# Popen(args, **parms)
# Выполняет команду, запуская новый дочерний процесс,
# и возвращает объект класса Popen, представляющий новый процесс.
```

```
# Команда определяется в аргументе args либо как строка, такая как 'ls -l',
# либо как список строк, такой как ['ls', '-l'].

# Создаем кортеж расширений файлов, которые будут нужны
EXT = ('.py', '.PY')

# Создаем список файлов с нужным расширением в текущей директории
files = [f.name for f in os.scandir() if f.is_file() and f.name.endswith(EXT)]
print('Файлы для упаковки:', files)

# Для создания процесса используем класс Popen
# Будет создан процесс архиватора 7-Zip
# Для Windows флаг CREATE_NEW_CONSOLE укажет создать новую консоль для
процесса

# packer = Popen(['7z','a','test.zip', *files],
#                 creationflags=CREATE_NEW_CONSOLE)
# Ждём завершения процесса, чтобы что-то делать дальше...
# packer.wait()

# Можно упростить, т.к. Popen поддерживает менеджер контекста:
with Popen(['7z','a','test.zip', *files],
           creationflags=CREATE_NEW_CONSOLE) as packer:
    print(packer.args)
    print("Ждём упаковку...")

print("Файлы упакованы, можно переименовывать")

# Переименовываем файл, созданный архиватором
os.rename('test.zip', 'backup.zip')

# В Python 3.5 добавлен упрощенный способ создания процессов - функция run.
# run запускает процесс, ждёт его завершения, возвращает объект
CompletedProcess.
# subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None,
#                shell=False, timeout=None, check=False, encoding=None,
errors=None)
#py_proc = run(['python', '-V'])
#print(py_proc)
```

**Листинг 3**

```
import os

# Создания каталога в текущей директории
os.mkdir('test_dir')

# Проверка существования
if not os.path.exists('test_dir'):
    os.mkdir('test_dir')
```

```
# Проверка содержимого каталога
dir_struct = os.listdir('.')
print(dir_struct)

# Проверка на объект-каталог
dirs = [ d for d in os.listdir('.') if os.path.isdir(d)]
print(dirs)

# Проверка на объект-файл
fls = [ f for f in os.listdir('.') if os.path.isfile(f)]
print(fls)

# Определение базового типа пути
base_path = os.path.basename('c:\\system\\apps\\Python\\Python.app')
print(base_path)

# Определение имени директории пути path
dir_path = os.path.dirname ('c:\\system\\apps\\Python\\Python.app')
print(dir_path)

# Разбиение пути к файлу
dir_tuple = os.path.split('c:\\system\\apps\\Python\\Python.app')
print(dir)
```

**Листинг 4**

```
import ipaddress

# Создание IPv4-адреса - функция ip_address()
ipv4 = ipaddress.ip_address('192.168.0.1')
print(dir(ipv4))

# Проверка диапазона, к которому принадлежит адрес - атрибуты is_loopback,
# is_multicast, is_reserved, is_private
print(ipv4.is_loopback)
print(ipv4.is_multicast)
print(ipv4.is_reserved)
print(ipv4.is_private)

# Операции с объектом адреса
ip1 = ipaddress.ip_address('192.168.1.0')
ip2 = ipaddress.ip_address('192.168.1.255')

if ip2 > ip1:
    print(True)

print(str(ip1))
print(int(ip1))
print(ip1 + 5)
print(ip1 - 5)
```

```
# Создание объекта, описывающего сеть - функция ip_network()
subnet = ipaddress.ip_network('80.0.1.0/28')
ba = subnet.broadcast_address
print(ba)

# Просмотр всех хостов для объекта-сети
print(list(subnet.hosts()))

# Разбиение сети на подсети
print(list(subnet.subnets()))

# Обращение к любому адресу в сети
print(subnet[1])

# Создание интерфейса
ipv4_int = ipaddress.ip_interface('10.0.1.1/24')

# Получение адреса
print(ipv4_int.ip)
# Получение маски
print(ipv4_int.netmask)
# Получение сети
print(ipv4_int.network)
```

**Листинг 5**

```
import ipaddress

ip_1 = '10.0.1.1/24'
ip_2 = '10.0.1.0/24'

def ip_network_check(ip_addr):
    try:
        ipaddress.ip_network(ip_addr)
        return True
    except ValueError:
        return False

print(ip_network_check(ip_1))
print(ip_network_check(ip_2))
```

**Листинг 6**

```
# Табличное представление списка словарей
from tabulate import tabulate
tuples_list = [('Python', 'interpreted', '1991'),
               ('JAVA', 'compiled', '1995'),
               ('C', 'compiled', '1972')]
```

```
columns = ['programming language', 'type', 'year']
# Указание заголовков в параметре headers
print(tabulate(tuples_list, headers = columns))
```

**Листинг 7**

```
# Табличное представление списка словарей
from tabulate import tabulate
tuples_list = [ ('programming language', 'type', 'year'),
                ('Python', 'interpreted', '1991'),
                ('JAVA', 'compiled', '1995'),
                ('C', 'compiled', '1972')]

columns = ['programming language', 'type', 'year']
# Указание первой строки таблицы как набора заголовков
print(tabulate(tuples_list, headers = 'firstrow'))
```

**Листинг 8**

```
# Табличное представление списка словарей
from tabulate import tabulate
dicts_list = [{'programming language': 'Python',
               'type': 'interpreted',
               'year': '1991'},
              {'programming language': 'JAVA',
               'type': 'compiled',
               'year': '1995'},
              {'programming language': 'C',
               'type': 'compiled',
               'year': '1972'}]

# Табличное представление списка словарей
print(tabulate(dicts_list, headers = 'keys'))
```

**Листинг 9**

```
# Табличное представление списка словарей
from tabulate import tabulate
dicts_list = [{'programming language': 'Python',
               'type': 'interpreted',
               'year': '1991'},
              {'programming language': 'JAVA',
               'type': 'compiled',
               'year': '1995'},
              {'programming language': 'C',
               'type': 'compiled',
               'year': '1972'}]
```

```
# grid-формат
print(tabulate(dicts_list, headers='keys', tablefmt="grid"))

# markdown-формат
print(tabulate(dicts_list, headers='keys', tablefmt="pipe"))

# html-формат
print(tabulate(dicts_list, headers='keys', tablefmt="html"))
```

**Листинг 10**

```
# Табличное представление списка словарей
from tabulate import tabulate
dicts_list = [{'programming language': 'Python',
               'type': 'interpreted',
               'year': '1991'},
              {'programming language': 'JAVA',
               'type': 'compiled',
               'year': '1995'},
              {'programming language': 'C',
               'type': 'compiled',
               'year': '1972'}]

# Выравнивание по центру
print(tabulate(dicts_list, headers='keys', tablefmt="pipe", stralign="center"))
```

**Листинг 11**

```
from pprint import pprint

# Отображение словаря с вложенными словарями

dict_dicts = {'el_1': {'el_1.1': 'val_1.1', 'el_1.2': 'val_1.2', 'el_1.3':
'val_1.3'},
'el_2': {'el_2.1': 'val_2.1', 'el_2.2': 'val_2.2', 'el_2.3': 'val_2.3'},
'el_3': {'el_3.1': 'val_3.1', 'el_3.2': 'val_3.2', 'el_3.3': 'val_3.3'}}

pprint(dict_dicts)

# Отображение строки
str_pp = '\n programming language Python\n type interpreted\n year 1991\n
license free \n'
pprint(str_pp)
```