Библиотеки Python (часть 1)

Для работы с приложениями

Одна из областей применения Python — разработка веб-приложений и десктопных программ. Библиотеки помогают сделать процесс проще.

Requests. Упрощает генерацию HTTP-запросов к другим сервисам, помогает писать их очень просто и быстро. Код получается лаконичным, а запрос легко настроить и отправить. Библиотека поддерживает множество функций и написана понятным языком.

HTTPX. Расширение для Requests. Оно поддерживает все функции библиотеки, помогает работать с HTTP и асинхронностью. HTTPX помогает отправлять и получать запросы, работать с клиент-серверными протоколами взаимодействия веб-сервера и приложения.

Асинхронное программирование — концепция программирования, при которой результат выполнения функции доступен спустя некоторое время в виде асинхронного (нарушающего стандартный порядок выполнения) вызова. Запуск длительных операций происходит без ожидания их завершения и не блокирует дальнейшее выполнение программы.

Функции

HTTPX основан на хорошо зарекомендовавшем себя удобстве использования requests и дает вам:

- Широко совместимый с запросами API .
- Стандартный синхронный интерфейс, но с поддержкой асинхронности, если вам это нужно .
- Поддержка HTTP/1.1 и HTTP/2 .
- Возможность делать запросы непосредственно к приложениям WSGI или приложениям ASGI .
- Везде строгие тайм-ауты.
- Полностью введите аннотации.
- 100% тестовое покрытие.

Плюс все стандартные функции requests...

- Международные домены и URL-адреса
- Keep-Alive и пул соединений
- Сеансы с сохраняемостью файлов cookie
- Проверка SSL в браузере

- Базовая/дайджест-аутентификация
- Элегантные файлы cookie ключ/значение
- Автоматическая декомпрессия
- Автоматическое декодирование контента
- Тело ответа Unicode
- Загрузка нескольких файлов
- Поддержка прокси-сервера HTTP(S)
- Тайм-ауты подключения
- Потоковые загрузки
- Поддержка .netrc
- Разделенные запросы

Retrying. Автоматизирует повторные вызовы. Если действие в коде, например запрос к внешнему источнику, не выполнилось и вернуло ошибку, с помощью Retrying можно настроить автоматические повторные попытки. Количество попыток и возможные изменения в запросах тоже настраиваются.

Самый простой вариант использования — повторная попытка ненадежной функции всякий раз, когда возникает исключение, пока не будет возвращено значение.

```
import random
from retrying import retry
@retry
def do_something_unreliable():
    if random.randint(0, 10) > 1:
        raise IOError("Broken sauce, everything is hosed!!!111one")
    else:
        return "Awesome sauce!"

print do_something_unreliable()
```

Функции

- Общий API декоратора
- Укажите условие остановки (т.е. ограничение по количеству попыток)
- Укажите условие ожидания (т. е. экспоненциальная отсрочка ожидания между попытками)
- Настройка повторных попыток для исключений
- Настроить повторную попытку для ожидаемого возвращаемого результата

Как вы видели выше, поведение по умолчанию — бесконечное повторение без ожидания.

```
@retry
def never_give_up_never_surrender():
    print "Retry forever ignoring Exceptions, don't wait between retries"
```

Давайте установим некоторые границы, например, количество попыток, прежде чем сдаться.

```
@retry(stop_max_attempt_number=7)

def stop_after_7_attempts():

    print "Stopping after 7 attempts"
```

У нас мало времени, так что давайте установим границы того, как долго мы должны повторять действия.

```
@retry(stop_max_delay=10000)

def stop_after_10_s():

    print "Stopping after 10 seconds"
```

Dramatiq vs Celery:

Мы часто сталкиваемся с асинхронными задачами в веб-разработке. Когда нам нужны такие операции как:

- Отправка электронных писем
- Отправка запросов к внешним API
- Долгие математические операции
- Сложные запросы к базе данных

Наше приложение тратит время на их выполнение и заставляет пользователя ждать, что может негативно сказаться на его пользовательском опыте. Для решения этой проблемы существуют асинхронные очереди выполнения задач.

Celery. Помогает правильно распределить множество задач в больших проектах, расставить приоритеты и выполнить их в оптимальной последовательности. Часто используется в backend-разработке, например с фреймворком Django.

Celery снижает нагрузку на производительность, выполняя часть функциональности в виде отложенных задач либо на том же сервере, что и другие задачи, либо на другом сервере. Чаще всего разработчики используют его для отправки электронных писем. Тем не менее, Celery может предложить гораздо больше. В этой статье я покажу вам некоторые основы Celery, а также пару лучших практик Python-Celery.

Основы Celery

Лучше создать экземпляр в отдельном файле, так как будет необходимо запустить Celery так же, как он работает с WSGI в Django. Например, если вы создадите два экземпляра Flask и Celery в одном файле в приложении Flask и запустите его, у вас будет два экземпляра, но вы будете использовать только один. То же самое, когда вы запускаете Celery.

Первое, что вам нужно, это экземпляр Celery. Мы называем это *приложением Celery* или просто *приложением* для краткости. Поскольку этот экземпляр используется в качестве точки входа для всего, что вы хотите делать в Celery, например, для создания задач и управления работниками, другие модули должны иметь возможность импортировать его.

В этом уроке мы храним все, что содержится в одном модуле, но для более крупных проектов вы хотите создать специальный модуль .

Давайте создадим файл `tasks.py`:

```
from celery import Celery

app = Celery('tasks', broker='pyamqp://guest@localhost//')

@app.task
def add(x, y):
    return x + y
```

Первым аргументом `Celery` является имя текущего модуля. Это нужно только для того, чтобы имена могли генерироваться автоматически, когда задачи определены в модуле *__main__* .

Второй аргумент — это аргумент ключевого слова брокера, указывающий URL-адрес брокера сообщений, который вы хотите использовать. Здесь мы используем RabbitMQ (тоже вариант по умолчанию).

Celery и его проблемы

Хотя самой популярной для языка программирования Python давно является Celery. Она предоставляет большие возможности работы с задачами, запуске их с определенным периодом и тонкой настройки.

До выхода Python 3.7 она была чуть ли не единственным способом реализовать асинхронную обработку задач. Но она, к сожалению, также известна своими проблемами с совместимостью с Python 3.7 и выше и отсутствием поддержки Windows с версии Celery 4.

И поэтому для тестирования своего приложения приходится разворачивать Celery в Docker или переходить на Linux, что для многих является нежелательным.

Современное решение - Dramatiq

Dramatiq - это простая в использовании и надежная библиотека распределенной обработки задач для Python 3, написанная в 2017 году для решения проблем Celery.

Какие же плюсы есть у Dramatiq?

- Активно разрабатывается и используется в производстве
- Отличная документация
- Автоматическая перезагрузка кода ваших задач
- Поддержка Redis и RabbitMQ
- Понятный исходный код, который позволяет сообществу развивать библиотеку

Flask. это небольшой и легкий веб-фреймворк, написанный на языке Python, предлагающий полезные инструменты и функции для облегчения процесса создания веб-приложений с использованием Python. Он обеспечивает гибкость и является более доступным фреймворком для новых разработчиков, так как позволяет создать веб-приложение быстро, используя только один файл Python. Flask — это расширяемая система, которая не обязывает использовать конкретную структуру директорий и не требует сложного шаблонного кода перед началом использования.

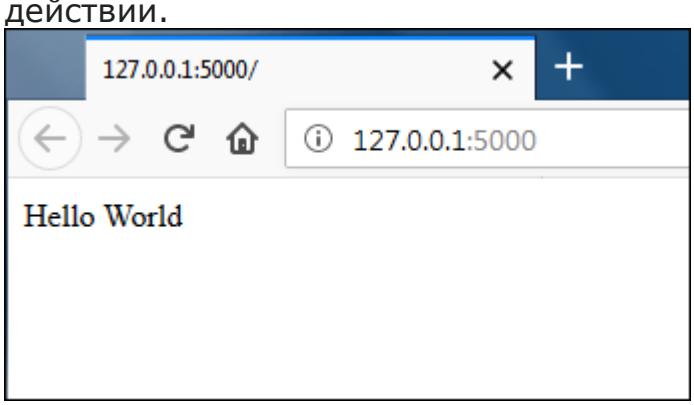
Создания простого приложения, которое выводит “Hello World”. Создаем новый файл `main.py` и вводим следующий код.

```
from flask import Flask

app = Flask(__name__)

@app.route('/')

def index():

    return 'Hello World'

if __name__ == "__main__":

    app.run()
```

Это приложение “Hello World”, созданное с помощью фреймворка Flask. Если код в `main.py` не понятен, это нормально. В следующих разделах все будет разбираться подробно. Чтобы запустить `main.py`, нужно ввести следующую команду в виртуальную среду Python.

```
(env) gvido@vm:~/flask_app$ python main.py

* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Запуск файла `main.py` запускает локальный сервер для разработки на порте 5000. Осталось открыть любимый браузер и зайти на `https://127.0.0.1:5000/`, чтобы увидеть приложение Hello World в действии.



Для логирования, обработки и форматирования данных

И при разработке, и при тестировании специалист должен иметь дело с большим количеством информации. «Сырые» данные нужно приводить к единому виду и очищать от лишних сведений, чтобы не вызывать ошибок и

получать более точные результаты. Уже обработанные – заносить в различные системы. А саму работу программы следует логировать, то есть записывать сведения о ее действиях. Логи нужно форматировать, выводить и сохранять в файл. Для всего перечисленного тоже есть свои библиотеки.

Rich. Позволяет форматировать текст, который Python выводит в консоль. Словосочетание Rich Text означает «отформатированный», «украшенный» текст. Можно сделать разноцветными сообщения в консоли, изменить в них начертание шрифта, выводить таблицы, пользоваться эмодзи. Это удобно, если нужны понятные и наглядные логи.

Если вы не используете такой инструмент, как Rich, вывод вашего кода на терминал может быть немного скучным и трудным для понимания. Если вы хотите сделать его понятнее и красивее, вы, вероятно, захотите использовать Rich — и вы попали в нужное место, чтобы узнать, как это сделать.

Как установить Rich

Вы можете установить Rich с помощью pip:

```
pip install Rich
```

Чтобы узнать, на что способен Rich, вы можете набрать в терминале следующую команду:

```
python -m rich
```

Теперь вы можете видеть, что мы можем делать довольно много вещей с Rich. Давайте попробуем несколько из них, чтобы увидеть, как они работают.

Как вывести Rich на Python

Rich имеет возможность выделять выходные данные в соответствии с типом данных. Мы импортируем альтернативную функцию `print` из библиотеки Rich, которая принимает те же аргументы, что и встроенная функция `print`.

Чтобы не путаться со встроенной функцией `print`, мы будем импортировать `print` из библиотеки `rich` как `rprint`.

```
from rich import print as rprint


nums_list = [1, 2, 3, 4]
rprint(nums_list)


nums_tuple = (1, 2, 3, 4)
rprint(nums_tuple)


nums_dict = {'nums_list': nums_list, 'nums_tuple': nums_tuple}
rprint(nums_dict)


bool_list = [True, False]
rprint(bool_list)
```

Вывод:

```
PS C:\Users\ashut\Desktop\Test\Rich-Tutorial> python rich-print.py
[1, 2, 3, 4]
(1, 2, 3, 4)
{'nums_list': [1, 2, 3, 4], 'nums_tuple': (1, 2, 3, 4)}
[True, False]
PS C:\Users\ashut\Desktop\Test\Rich-Tutorial>
```

Видите, как разные типы данных выделяются разными цветами? Это может нам сильно помочь при отладке.

Loguru. Инструмент для удобного и простого логирования данных. В Python есть встроенная библиотека logging, но многие разработчики считают ее неудобной из-за сложных конфигураций логов, неудобства настроек разного уровня логирования и ротации файлов логов. Поэтому они пишут логи через loguru. Библиотека имеет широкие настройки форматирования, удобна в работе и поддерживает множество функций, например архивирование файлов с логами.

Использование Lovuru очень просто, в логовре всего один объект: logger. Для удобства использования в системе использования используется регистратор, и он предварительно сконфигурирован, и начало выводится в STDERR по умолчанию (но они полностью настроены), а информация журнала печати настроен по умолчанию. Цвет. Цвет. Как показано ниже, использование Loguru действительно просто:

```
1. from loguru import logger
2. logger.debug("This's a log message")
```

Приведенное выше оператор регистрации по умолчанию отпечатывает выходное оператор на STDERR (Console), вывод выглядит следующим образом:

```
2020-08-07 15:55:39.206 | DEBUG    | __main__:<module>:8 - This's a log message
```

Видно, что loguru настроен с помощью формата выходного журнала, имеет время, уровень, имя модуля, номер строки и информацию о журналах. Вам не нужно вручную создавать регистратор. Используйте его напрямую, и вывод все еще окрашен, Это выглядит более дружелюбно. Поэтому нам не нужно никакого продвижения настроить, вы можете использовать его напрямую.

Dateparser. Инструмент находит и определяет даты в массиве данных. Он работает с разными форматами записи: и строгими, и «человекопонятными». Dateparser сможет найти дату и формата «25.06.1999», и формата «вчера» или «месяц назад». В основном библиотека используется при парсинге данных.

```
>>> import dateparser

>>> dateparser.parse('Fri, 12 Dec 2014 10:55:50')
datetime.datetime(2014, 12, 12, 10, 55, 50)

>>> dateparser.parse('1991-05-17')
datetime.datetime(1991, 5, 17, 0, 0)

>>> dateparser.parse('In two months')  # today is 1st Aug 2020
datetime.datetime(2020, 10, 1, 11, 12, 27, 764201)
```

Для отслеживания и анализа

Профилирование — это неотъемлемая часть любых работ по оптимизации кода или производительности программ. Любой опыт, любые знания в сфере оптимизации производительности, которые уже у вас есть, не принесут особой пользы в том случае, если вы не знаете о том, где их применить. В результате оказывается, что поиск узких мест приложений может помочь в деле решения проблем производительности, поможет сделать это быстро и приложив не слишком много усилий.

Далее мы обсудим инструменты и методы работы, которые способны обнаруживать и конкретизировать проблемы с производительностью кода, связанные и с ресурсами процессора, и с потреблением памяти. Здесь же мы поговорим о том, как реализовывать (почти безо всяких усилий) простые механизмы, позволяющие бороться с проблемами производительности. Эти механизмы используются в тех случаях, когда даже точно просчитанные изменения кода больше не позволяют улучшить ситуацию.

Идентификация узких мест

В деле оптимизации производительности программ лениться — это хорошо. Вместо того чтобы пытаться понять то, какая именно часть кодовой базы замедляет приложение, можно просто воспользоваться инструментами профилирования кода. Они позволят найти те места приложения, на которые стоит обратить внимание, такие, которые нуждаются в более глубоком исследовании.

Самый распространённый инструмент, который используют для этих целей Python-разработчики — это cProfile. Это — стандартный модуль, который способен измерять время выполнения функций.

Рассмотрим следующую функцию, которая возводит (медленно) e в степень X:

```
# some-code.py
from decimal import *
def exp(x):
    getcontext().prec += 2
    i, lasts, s, fact, num = 0, 0, 1, 1, 1
    while s != lasts:
        lasts = s
        i += 1
        fact *= i
        num *= x
        s += num / fact
    getcontext().prec -= 2
    return +s
exp(Decimal(3000))
```

Исследуем этот медленный код с помощью `cProfile`:

```
python -m cProfile -s cumulative some-code.py
         1052 function calls (1023 primitive calls) in 2.765 seconds
   Ordered by: cumulative timek
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
      5/1    0.000    0.000    2.765    2.765 {built-in method builtins.exec}
        1    0.000    0.000    2.765    2.765 some-code.py:1(<module>)
        1    2.764    2.764    2.764    2.764 some-code.py:3(exp)
      4/1    0.000    0.000    0.001    0.001 <frozen
importlib._bootstrap>:986(_find_and_load)
      4/1    0.000    0.000    0.001    0.001 <frozen
importlib._bootstrap>:956(_find_and_load_unlocked)
      4/1    0.000    0.000    0.001    0.001 <frozen
importlib._bootstrap>:650(_load_unlocked)
      3/1    0.000    0.000    0.001    0.001 <frozen
importlib._bootstrap_external>:842(exec_module)
      5/1    0.000    0.000    0.001    0.001 <frozen
importlib._bootstrap>:211(_call_with_frames_removed)
        1    0.000    0.000    0.001    0.001 decimal.py:2(<module>)
...
```

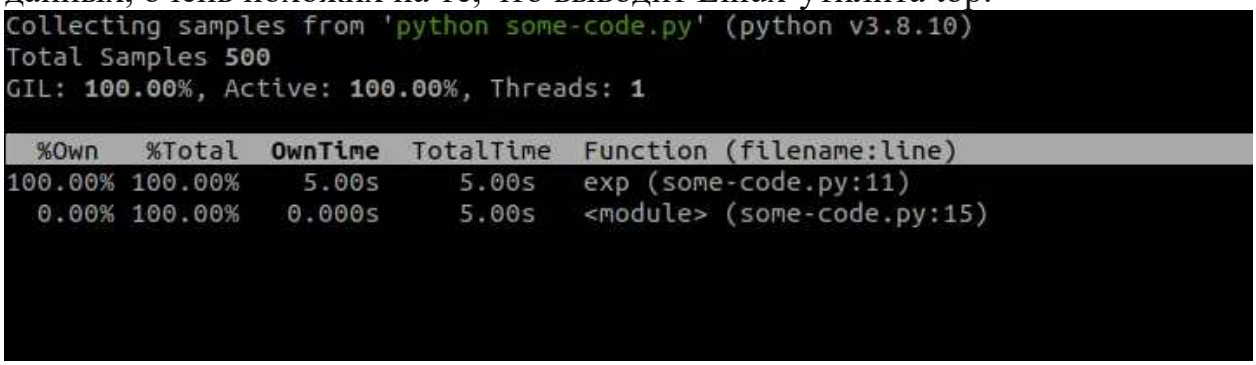
Тут мы воспользовались опцией `-s cumulative` для сортировки выходных данных по суммарному времени, затраченному на выполнение каждой из функций. Это упрощает поиск проблемных участков кода. Видно, что почти всё время (примерно 2,764 секунды) в ходе одного сеанса выполнения программы было потрачено в функции exp.

Профилирование подобного рода может принести пользу, но его, к сожалению, не всегда достаточно. cProfile снабжает нас информацией лишь о вызовах функций, но не об отдельных строках кода. Если вызвать какую-то особую функцию, вроде append, в разных местах, то сведения обо всех её вызовах будут собраны в одной строке отчёта cProfile. То же самое относится и к скриптам, вроде того, который мы исследовали выше. Он содержит единственную функцию, которая вызывается лишь один раз, в результате у cProfile оказывается не особенно много данных для формирования отчёта.

Иногда такая роскошь, как локальная отладка проблемного кода, программисту не доступна. Или бывает так, что нужно проанализировать проблему с производительностью, что называется, «на лету», когда она возникает в продакшн-окружении. В таких ситуациях можно воспользоваться пакетом py-spy. Это — профилировщик, способный исследовать программы, которые уже запущены. Например — приложения, работающие в продакшне, или на любой удалённой системе:

```
pip install py-spy
python some-code.py &
[1] 1129587
ps -A -o pid,cmd | grep python
...
1129587 python some-code.py
1130365 grep python
sudo env "PATH=$PATH" py-spy top --pid 1129587
```

В этом фрагменте кода мы сначала устанавливаем py-spy, а потом, в фоне, запускаем программу, которая выполняется длительное время. Это приводит к автоматическому показу идентификатора процесса (PID), но если мы его не знаем, можно, для его выяснения, воспользоваться командой ps. И, наконец, мы запускаем py-spy в режиме top, передавая ему PID. Это ведёт к выводу данных, очень похожих на те, что выводит Linux-утилита top.



Данные, выводимые py-spy в режиме top

Тут, правда, в нашем распоряжении оказывается не так много данных, так как скрипт представляет собой всего лишь одну функцию, выполняющуюся длительное время. Но в реальных случаях, вероятнее всего, подобный отчёт

будет содержать сведения о многих функциях, совместно использующих процессорное время. А это может помочь несколько прояснить ситуацию с существующими проблемами производительности программы.

Более глубокое исследование кода

Профилировщики, о которых мы только что говорили, должны помочь вам в деле обнаружения функций, которые вызывают проблемы, связанные с производительностью. Но если это не приведёт к обнаружению конкретных строк кода, которые надо доработать, это значит, что мы можем обратиться к профилировщикам, которые позволяют исследовать программы на более глубоком уровне.

Один из таких инструментов представлен пакетом `line_profiler`. Он, как можно судить по его названию, может использоваться для выяснения того, сколько времени уходит на выполнение каждой конкретной строки кода:

```
# https://github.com/pyutils/line_profiler
pip install line_profiler
kernprof -l -v some-code.py  # Это может занять некоторое время...
Wrote profile results to some-code.py.lprof
Timer unit: 1e-06 s
Total time: 13.0418 s
File: some-code.py
Function: exp at line 3
Line #      Hits         Time  Per Hit   % Time  Line Contents
     3                                           @profile
     4                                           def exp(x):
     5         1          4.0      4.0      0.0      getcontext().prec += 2
     6         1          0.0      0.0      0.0      i, lasts, s, fact, num = 0, 0,
1, 1, 1
     7      5818       4017.0      0.7      0.0      while s != lasts:
     8      5817       1569.0      0.3      0.0          lasts = s
     9      5817       1837.0      0.3      0.0          i += 1
    10      5817       6902.0      1.2      0.1          fact *= i
    11      5817       2604.0      0.4      0.0          num *= x
    12      5817   13024902.0   2239.1     99.9          s += num / fact
    13         1          5.0      5.0      0.0      getcontext().prec -= 2
    14         1          2.0      2.0      0.0      return +s
```

Библиотека `line_profiler` распространяется вместе с интерфейсом командной строки `kernprof` (названным так в честь Роберта Керна), который используется для организации эффективного анализа результатов тестовых прогонов программ. Передача нашего кода этой утилите приводит к созданию `.lprof`-файла со сведениями об анализе кода. В нашем распоряжении, кроме того, оказывается отчёт, выводимый на экран (при использовании опции `-v`), подобный показанному выше. Тут чётко видны места функции, на выполнение которых уходит больше всего времени. Это очень сильно помогает в деле поиска и исправления проблем с

производительностью. В выходных данных можно заметить декоратор @profile, добавленный к функции exp. Это — необходимое дополнение, которое позволяет line_profiler узнать о том, какую именно функцию в файле мы хотим изучить.

Но даже если построчно проанализировать функцию, первоисточник проблем с производительностью можно и не обнаружить. Например, такое бывает в том случае, если в конструкциях while или if используются условия, составленные из множества выражений. В подобных случаях имеет смысл переписать проблемные фрагменты, разбить одну строку кода на несколько. Это позволит получить более полные и понятные результаты анализа.

Pympler. Мониторит и анализирует память, которая используется при исполнении кода программ на Python. Инструмент находит ее избыточное потребление, утечки и другие баги. С помощью Pympler можно узнать все о размере и длительности процессов приложения на Python за время работы.

Бывают ситуации, когда нам нужно отслеживать использование памяти определенным типом объекта, и названная библиотека Python pympler может быть очень полезна для таких требований. У pympler есть список модулей, которые позволяют нам различными способами отслеживать использование памяти кодом Python.

- **asizeof** — этот модуль предоставляет нам различные методы измерения размера объектов.
- **classtracker** — этот модуль предоставляет нам методы для мониторинга использования памяти объектами, созданными определяемыми пользователем классами.
- **classtracker_stats** — этот модуль позволяет нам classtrackerпо-разному форматировать данные, полученные с помощью модуля.
- **tracker** — этот модуль позволяет нам отслеживать изменения в общей памяти с течением времени.
- **muppy** — этот модуль позволяет нам отслеживать использование памяти списком объектов с течением времени.
- **garbagegraph**— этот модуль позволяет нам анализировать объекты, которые создают эталонный цикл, поэтому его трудно собрать сборщиком мусора.
- **refbrowser** — этот модуль позволяет нам выполнять древовидное исследование рефереров объектов.
- **refgraph** — этот модуль предоставляет способы иллюстрировать объекты и их ссылки в виде ориентированных графов. Он может даже генерировать графические ориентированные графы.
- **summary** — этот модуль предоставляет функции для суммирования информации для списка объектов.

- **asizeof.asizeof()** Этот метод принимает в качестве входных данных один или несколько объектов и возвращает размер каждого объекта в байтах.

Ниже мы создали два списка, первый из которых представляет собой настоящий список со всеми элементами, а второй — генератор Python. Затем мы измерили размер обоих из них, используя asizeof()метод.

```
from pympler import asizeof
```

```
l1 = [i for i in range(10)]
l2 = range(10)

print("Size of List l1              : %d bytes"%asizeof.asizeof(l1))
print("Size of List l1              : %d bytes"%asizeof.asizeof(l2))
print("Size of List l1,l2 Combined  : %d bytes"%asizeof.asizeof(l1,l2))
print("Size of List l1 & l2         : %d bytes, %d bytes"%asizeof.asizesof(l1, l2))
```

```
Size of List l1              : 504 bytes
Size of List l1              : 48 bytes
Size of List l1,l2 Combined  : 552 bytes
Size of List l1 & l2         : 504 bytes, 48 bytes
```