Лабораторная работа 2.9. Рекурсия в языке Python

Цель работы: *приобретение навыков по работе с рекурсивными функциями при написании программ с помощью языка программирования Python версии 3.x.*

Ход работы

Сущность рекурсии

Функция может содержать вызов других функций. В том числе процедура может вызвать саму себя. Никакого парадокса здесь нет – компьютер лишь последовательно выполняет встретившиеся ему в программе команды и, если встречается вызов процедуры, просто начинает выполнять эту функцию. Без разницы, какая функция дала команду это делать.

Пример рекурсивной функции:

```
def rec(n):
    if n > 0:
        rec(n - 1)

    print(n)
```

Рассмотрим, что произойдет, если в основной программе поставить вызов, например, вида `rec(3)`. Ниже представлена блок-схема, показывающая последовательность выполнения операторов.

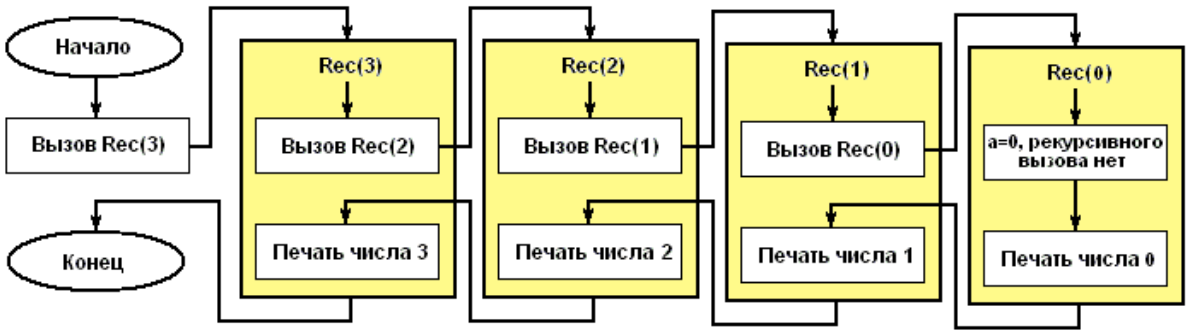

Рисунок 1. Блок схема работы рекурсивной процедуры.

Функция `rec` вызывается с параметром `n = 3`. В ней содержится вызов функции `rec` с параметром `n = 2`. Предыдущий вызов еще не завершился, поэтому можете представить себе, что создается еще одна функция и до окончания ее работы первая свою работу не заканчивает. Процесс вызова заканчивается, когда параметр `n = 0`. В этот момент одновременно выполняются 4 экземпляра функции. Количество одновременно выполняемых функций называют *глубиной рекурсии*.

Пример 1. Если бы мы хотели узнать сумму чисел от 1 до n, где n — натуральное число, то могли бы посчитать вручную 1 + 2 + 3 + 4 + ... + (несколько часов спустя) + n. А можно просто написать цикл `for`:

```
n = 0
for i in range (1, n+1):
    n += i
```

Или использовать рекурсию:

```
def recursion(n):
    if n == 1:
        return 1

    return n + recursion(n - 1)
```

У рекурсии есть несколько преимуществ в сравнении с первыми двумя методами. Рекурсия занимает меньше времени, чем выписывание 1 + 2 + 3 на сумму от 1 до 3, рекурсия может работать в обратную сторону:

Вызов функций: (4 -> 4 + 3 -> 4 + 3 + 2 -> 4 + 3 + 2 + 1 -> 10)

Принимая во внимание, что цикл `for` работает строго вперед: (1 -> 1 + 2 -> 1 + 2 + 3 -> 1 + 2 + 3 + 4 -> 10). Иногда рекурсивное решение проще, чем итеративное решение. Это очевидно при реализации обращения связанного списка.

Как и когда происходит рекурсия

Рекурсия появляется когда вызов функции повторно вызывает ту же функцию до завершения первоначального вызова функции. Например, рассмотрим известное математическое выражение $x!$ (т. е. факториал). Факториал определяется для всех неотрицательных целых чисел следующим образом:

Если число равно 0, то будет 1. В противном случае ответом будет то, что число умножается на факториал на единицу меньше этого числа.

В Python наивная реализация факториала может быть определена как функция следующим образом:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)
```

Иногда функции рекурсии трудно понять, поэтому давайте рассмотрим поэтапно. Рассмотрим выражение `factorial(3)`. Эта и все остальные вызовы функций создают новую среду. Среда представляет собой таблицу, которая сопоставляет идентификаторы (например, `n`, `factorial`, `print` и т. д.) с их соответствующими значениями. В любой момент времени вы можете получить доступ к текущей среде с помощью `locals()`. В первом вызове функции единственная локальная переменная, которая определяется `n = 3` . Поэтому `locals()` будет показывать `{"n": 3}`. Так как `n == 3`, возвращаемое значение становится `n * factorial(n - 1)`.

На следующем этапе ситуация может немного запутаться. Глядя на наше новое выражение, мы уже знаем, что такое `n`. Однако мы еще не знаем, что такое `factorial(n - 1)`. Во-первых, `n - 1` принимает значение `2` . Затем `2` передаётся `factorial` как значение для `n`. Поскольку это новый вызов функции, создаётся вторая среда для хранения нового `n`. Пусть A — первое окружение, а B — второе окружение. A всё ещё существует и равен `{"n": 3}` , однако B (что

равно `{"n": 2}`) является текущей средой. Если посмотреть на тело функции, возвращаемое значение, опять же, `n * factorial(n - 1)`. Не определяя это выражение, заменим его на исходное выражение `return`. Делая это, мы мысленно отбрасываем `B`, поэтому не забудьте заменить `n` соответственно (т.е. ссылки на B `n` заменены на `n - 1)` который использует `A` `n`). Теперь исходное обратное выражение становится `n * ((n - 1) * factorial((n - 1) - 1))`. Подумайте, почему так?

Теперь давайте определим `factorial((n - 1) - 1))`. Так как A `n == 3`, мы пропускаем 1 через `factorial`. Поэтому мы создаем новую среду C, которая равна `{"n": 1}`. Мы снова возвращаем значение `n * factorial(n - 1)`. Итак, заменим исходный `factorial((n - 1) - 1))` выражения return аналогично тому, как раньше мы скорректировали исходное выражение return. Исходное выражение теперь `n * ((n - 1) * ((n - 2) * factorial((n - 2) - 1)))`.

Почти закончили. Теперь нам нужно оценить `factorial((n - 2) - 1)`. На этот раз мы пропустим через `0`. Следовательно, должно получиться `1`. Теперь давайте проведём нашу последнюю замену. Исходное выражение return теперь `n * ((n - 1) * ((n - 2) * 1))`. Напомню, что исходное выражение возврата оценивается под A , выражение становится `3 * ((3 - 1) * ((3 - 2) * 1))`. Здесь получается 6. Чтобы убедиться, что это правильный ответ, вспомните, что `3! == 3 * 2 * 1 == 6`. Прежде чем читать дальше, убедитесь, что вы полностью понимаете концепцию среды и то, как они применяются к рекурсии.

Утверждение, `if n == 0: return 1`, называется базовым случаем. Потому что это не рекурсия. Базовый случай необходим, без него вы столкнетесь с бесконечной рекурсией. С учетом сказанного, если у вас есть хотя бы один базовый случай, у вас может быть столько случаев, сколько вы хотите. Например, можно записать факториал таким образом:

```
def factorial(n):
    if n == 0:
        return 1
    elif n == 1:
        return 1
    else:
        return n * factorial(n - 1)
```

У вас может также быть несколько случаев рекурсии, но мы не будем вдаваться в подробности, потому что это редкий случай, и его трудно мысленно обрабатывать.

Вы также можете иметь «параллельные» рекурсивные вызовы функций. Например, рассмотрим последовательность Фибоначчи, которая определяется следующим образом:

- Если число равно 0, то ответ равен 0.
- Если число равно 1, то ответ равен 1.

В противном случае ответ представляет собой сумму двух предыдущих чисел Фибоначчи.

Мы можем определить это следующим образом:

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Не будем разбирать эту функцию также тщательно, как и с `factorial(3)`, но окончательное значение возврата `fib(5)` эквивалентно следующему выражению:

```
(
  fib((n - 2) - 2)
  +
  (
    fib(((n - 2) - 1) - 2)
    +
    fib(((n - 2) - 1) - 1)
  )
)
+
(
  (
    fib(((n - 1) - 2) - 2)
    +
    fib(((n - 1) - 2) - 1)
  )
  +
  (
    fib(((n - 1) - 1) - 2)
    +
    (
      fib((((n - 1) - 1) - 1) - 2)
      +
      fib((((n - 1) - 1) - 1) - 1)
    )
  )
)
```

Решением `(1 + (0 + 1)) + ((0 + 1) + (1 + (0 + 1)))` будет `5`.

Теперь давайте рассмотрим еще несколько терминов:

Хвостовой вызов — это просто вызов рекурсивной функции, который является последней операцией и должна быть выполнена перед возвратом значения. Чтобы было понятно, `return foo(n - 1)` — это хвост вызова, но return `foo(n - 1) + 1` не является (поскольку операция сложения будет последней операцией).

Оптимизация хвостового вызова (TCO) — это способ автоматического сокращения рекурсии в рекурсивных функциях.

Устранение хвостового вызова (TCE) - это сокращение хвостового вызова до выражения, которое может быть оценено без рекурсии. TCE — это тип TCO.

Оптимизация хвоста вызова может пригодиться по нескольким причинам:

1. Интерпретатор может снизить объём памяти, занятый средами. Поскольку ни у кого нет неограниченной памяти, чрезмерные рекурсивные вызовы функций приведут к переполнению стека.
2. Интерпретатор может уменьшить количество переключателей кадров стека.

В Python нет TCO по нескольким причинам, поэтому для обхода этого ограничения можно использовать другие методы. Выбор используемого метода зависит от варианта использования. Интуитивно понятно, что `factorial` и `fib` можно относительно легко преобразовать в итеративный код следующим образом:

```
def factorial(n):
    product = 1
```

```
    while n > 1:
        product *= n
        n -= 1
    return product


def fib(n):
    a, b = 0, 1
    while n > 0:
        a, b = b, a + b
        n -= 1
    return a
```

Обычно это самый эффективный способ устранения рекурсии вручную, но для более сложных функций может быть трудно.

Другим полезным инструментом является декоратор `lru_cache`, который можно использовать для уменьшения количества лишних вычислений.

```
from functools import lru_cache

. . .

@lru_cache
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n - 2) + fib(n - 1)
```

Теперь вы знаете, как избежать рекурсии в Python, но когда её нужно использовать? Ответ «не часто». Все рекурсивные функции могут быть реализованы итеративно. Это просто вопрос, как именно сделать. Однако есть редкие случаи, когда можно использовать рекурсию. Рекурсия распространена в Python, когда ожидаемые вводы не вызовут значительного количества вызовов рекурсивных функций.

Хотя приведённый выше пример последовательности Фибоначчи, хорошо показывает, как применять декоратор `lru_cache`, при этом он имеет неэффективное время работы, из-за того, что выполняет 2 рекурсивных вызова. Количество вызовов функции растет экспоненциально от `n`.

В таком случае лучше использовать линейную рекурсию:

```
def fib(n):
    if n <= 1:
        return (n, 0)
    else:
        (a, b) = fib(n - 1)
        return (a + b, a)
```

Но у этого примера есть проблема в возвращении пары чисел. Это показывает, что не всегда стоит использовать рекурсию.

Увеличение максимальной глубины рекурсии

Существует предел глубины возможной рекурсии, который зависит от реализации Python. Когда предел достигнут, возникает исключение `RuntimeError`:

```
RuntimeError: Maximum Recursion Depth Exceeded
```

Пример программы, которая может вызвать такую ошибку:

```
def cursing(depth):
    try:
        cursing(depth + 1) # actually, re-cursing
    except RuntimeError as RE:
        print('I recursed {} times!'.format(depth))


if __name__ == '__main__':
    cursing(0)
# Out: I recursed 1083 times!
```

Можно изменить предел глубины рекурсии с помощью вызова:

```
sys.setrecursionlimit(limit)
```

Чтобы проверить текущие параметры лимита, нужно запустить:

```
sys.getrecursionlimit()
```

Если запустить тот же метод выше с новым пределом, мы получим

```
sys.setrecursionlimit(2000)
cursing(0)  # Out: I recursed 1997 times!
```

В Python 3.5 ошибка стала называться RecursionError, которая является производной от RuntimeError.

Хвостовая рекурсия

Хвостовая рекурсия — частный случай рекурсии, при котором любой рекурсивный вызов является последней операцией перед возвратом из функции. Подобный вид рекурсии примечателен тем, что может быть легко заменён на итерацию путём формальной и гарантированно корректной перестройки кода функции. **Оптимизация хвостовой рекурсии** путём преобразования её в плоскую итерацию реализована во многих оптимизирующих компиляторах. В некоторых функциональных языках программирования спецификация гарантирует обязательную оптимизацию хвостовой рекурсии.

Типовой механизм реализации вызова функции основан на сохранении адреса возврата, параметров и локальных переменных функции в стеке и выглядит следующим образом:

1. В точке вызова в стек помещаются параметры, передаваемые функции, и адрес возврата.
2. Вызываемая функция в ходе работы размещает в стеке собственные локальные переменные.

3. По завершении вычислений функция очищает стек от своих локальных переменных, записывает результат (обычно — в один из регистров процессора).
4. Команда возврата из функции считывает из стека адрес возврата и выполняет переход по этому адресу. Либо непосредственно перед, либо сразу после возврата из функции стек очищается от параметров.

Таким образом, при каждом рекурсивном вызове функции создаётся новый набор её параметров и локальных переменных, который вместе с адресом возврата размещается в стеке, что ограничивает максимальную глубину рекурсии объёмом стека. В чисто функциональных или декларативных (типа Пролога) языках, где рекурсия является единственным возможным способом организации повторяющихся вычислений, это ограничение становится крайне существенным, поскольку, фактически, превращается в ограничение на число итераций в любых циклических вычислениях, при превышении которого будет происходить *переполнение стека*.

Нетрудно видеть, что необходимость расширения стека при рекурсивных вызовах диктуется требованием восстановления состояния вызывающего экземпляра функции (то есть её параметров, локальных данных и адреса возврата) после возврата из рекурсивного вызова. Но если рекурсивный вызов является *последней* операцией перед выходом из вызывающей функции и результатом вызывающей функции должен стать результат, который вернёт рекурсивный вызов, сохранение контекста уже не имеет значения — ни параметры, ни локальные переменные уже использоваться не будут, а адрес возврата уже находится в стеке. Поэтому в такой ситуации вместо полноценного рекурсивного вызова функции можно просто заменить значения параметров в стеке и передать управление на точку входа. До тех пор, пока исполнение будет идти по этой рекурсивной ветви, будет, фактически, выполняться обычный цикл. Когда рекурсия завершится (то есть исполнение пройдёт по терминальной ветви и достигнет команды возврата из функции) возврат произойдёт сразу в исходную точку, откуда произошёл вызов рекурсивной функции. Таким образом, при любой глубине рекурсии стек переполнен не будет.

Вот пример обратного отсчета, написанного с использованием хвостовой рекурсии:

```
def countdown(n):
    if n == 0:
        print("Blastoff!")
    else:
        print(n)
        countdown(n-1)
```

Любое вычисление, которое может быть выполнено с использованием итерации, также может быть выполнено с использованием рекурсии. Вот версия `find_max` (поиск максимального значения в последовательном контейнере, например списке или кортеже), написанная с использованием хвостовой рекурсии:

```
def find_max(seq, max_so_far):
    if not seq:
        return max_so_far
    if max_so_far < seq[0]:
        return find_max(seq[1:], seq[0])
    else:
        return find_max(seq[1:], max_so_far)
```

Хвостовую рекурсию лучше не использовать, поскольку компилятор Python не обрабатывает оптимизацию для хвостовых рекурсивных вызовов. В таких случаях рекурсивное решение использует больше системных ресурсов, чем итеративное.

По умолчанию рекурсивный стек Python не превышает 1000 кадров. Это ограничение можно изменить, установив `sys.setrecursionlimit(15000)` который быстрее, однако этот метод потребляет больше памяти. Вместо этого мы также можем решить проблему рекурсии хвоста, используя интроспекцию стека.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-

# Эта программа показывает работу декоратора, который производит оптимизацию
# хвостового вызова. Он делает это, вызывая исключение, если оно является его
# прародителем, и перехватывает исключения, чтобы вызвать стек.

import sys


class TailRecurseException:
    def __init__(self, args, kwargs):
        self.args = args
        self.kwargs = kwargs


def tail_call_optimized(g):
"""
Эта программа показывает работу декоратора, который производит оптимизацию
хвостового вызова. Он делает это, вызывая исключение, если оно является его
прародителем, и перехватывает исключения, чтобы подделать оптимизацию хвоста.

Эта функция не работает, если функция декоратора не использует хвостовой вызов.
"""

    def func(*args, **kwargs):
        f = sys._getframe()
        if f.f_back and f.f_back.f_back and f.f_back.f_back.f_code == f.f_code:
            raise TailRecurseException(args, kwargs)
        else:
            while True:
                try:
                    return g(*args, **kwargs)
                except TailRecurseException, e:
                    args = e.args
                    kwargs = e.kwargs

    func.__doc__ = g.__doc__
    return func
```

Чтобы оптимизировать рекурсивные функции, мы можем использовать декоратор `@tail_call_optimized` для вызова нашей функции. Вот несколько примеров общей рекурсии с использованием декоратора, описанного выше:

```
@tail_call_optimized
def factorial(n, acc=1):
    "calculate a factorial"
    if n == 0:
        return acc

    return factorial(n-1, n*acc)


if __name__ == '__main__':
    print(factorial(10000))
# выводит большое число,
# но не доходит до лимита рекурсии
```

```
@tail_call_optimized
def fib(i, current = 0, next = 1):
    if i == 0:
        return current
    else:
        return fib(i - 1, next, current + next)


if __name__ == '__main__':
    print(fib(10000))
# также выводит большое число,
# но не доходит до лимита рекурсии
```

Аппаратура и материалы

1. Компьютерный класс общего назначения с конфигурацией ПК не хуже рекомендованной для ОС Windows 10 с подключением к глобальной сети Интернет.
2. Операционная система Windows 10.
3. Система контроля версий Git.
4. Браузер для доступа к web-сервису GitHub, рекомендован к использованию Google Chrome.
5. Дистрибутив языка программирования Python, включающий набор популярных библиотек Anaconda.
6. Интегрированная среда разработки PyCharm Community Edition.

Указания по технике безопасности

При работе на ЭВМ без разрешения руководителя занятия запрещается:

- подавать (снимать) напряжение на ПЭВМ и электрические розетки с распределительного щита;
- включать и выключать блоки питания ПЭВМ и мониторы;
- извлекать ПЭВМ из защитного кожуха;
- устранять неисправности, возникшие в ходе выполнения лабораторной работы.

Методика и порядок выполнения работы

1. Изучить теоретический материал работы.
2. Создать общедоступный репозиторий на GitHub, в котором будет использована лицензия MIT и язык программирования Python.

3. Выполните клонирование созданного репозитория.
4. Дополните файл `.gitignore` необходимыми правилами для работы с IDE PyCharm.
5. Организуйте свой репозиторий в соответствие с моделью ветвления git-flow.
6. Создайте проект PyCharm в папке репозитория.
7. Самостоятельно изучите работу со стандартным пакетом Python `timeit`. Оцените с помощью этого модуля скорость работы итеративной и рекурсивной версий функций `factorial` и `fib`. Во сколько раз измениться скорость работы рекурсивных версий функций `factorial` и `fib` при использовании декоратора `lru_cache`? Приведите в отчет и обоснуйте полученные результаты.
8. Самостоятельно проработайте пример с оптимизацией хвостовых вызовов в Python. С помощью пакета `timeit` оцените скорость работы функций `factorial` и `fib` с использованием интроспекции стека и без использования интроспекции стека. Приведите полученные результаты в отчет.
9. Выполните индивидуальные задания. Приведите в отчете скриншоты работы программ решения индивидуального задания.
10. Зафиксируйте сделанные изменения в репозитории.
11. Добавьте отчет по лабораторной работе в *формате PDF* в папку *doc* репозитория. Зафиксируйте изменения.
12. Выполните слияние ветки для разработки с веткой *master* / *main*.
13. Отправьте сделанные изменения на сервер GitHub.
14. Отправьте адрес репозитория GitHub на электронный адрес преподавателя.

Индивидуальное задание

Составить программу с использованием рекурсивных функций для решения задачи. Номер варианта определяется по согласованию с преподавателем.

1. Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в строке. При правильной расстановке выполняются условия:

- количество открывающих и закрывающих скобок равно.
- внутри любой пары открывающая – соответствующая закрывающая скобка, скобки расставлены правильно.

Примеры неправильной расстановки:)(, ())(, ())(() и т. п.

2. В строке могут присутствовать скобки как круглые, так и квадратные скобки. Каждой открывающей скобке соответствует закрывающая того же типа (круглой – круглая, квадратной- квадратная). Напишите рекурсивную функцию, проверяющую правильность расстановки скобок в этом случае.

Пример неправильной расстановки: ([ )].

3. Число правильных скобочных структур длины 6 равно 5: ()()(), (())(), ()(()), ((())), (()()). Напишите рекурсивную программу генерации всех правильных скобочных структур длины $2n$.

Указание: Правильная скобочная структура минимальной длины «()». Структуры большей длины получаются из структур меньшей длины, двумя способами:

- если меньшую структуру взять в скобки,
- если две меньших структуры записать последовательно.

4. Создайте рекурсивную функцию, печатающую все возможные перестановки для целых чисел от 1 до N.
5. Создайте рекурсивную функцию, печатающую все подмножества множества $\{1, 2, \ldots, N\}$.

6. Создайте процедуру, печатающую все возможные представления натурального числа N в виде суммы других натуральных чисел.
7. Создайте функцию, подсчитывающую сумму элементов массива по следующему алгоритму: массив делится пополам, подсчитываются и складываются суммы элементов в каждой половине. Сумма элементов в половине массива подсчитывается по тому же алгоритму, то есть снова путем деления пополам. Деления происходят, пока в получившихся кусках массива не окажется по одному элементу и вычисление суммы, соответственно, не станет тривиальным.
8. Напечатать в обратном порядке последовательность чисел, признаком конца которой является 0.
9. Даны целые числа m и n, где $0 \leq m \leq n$, вычислить, используя рекурсию, число сочетаний C_n^m по формуле: $C_n^0 = C_n^n = 1$, $C_n^m = C_{n-1}^m + C_{n-1}^{m-1}$ при $0 \leq m \leq n$. Воспользовавшись формулой

$$C_n^m = \frac{n!}{m!(m-n)!} \tag{1}$$

можно проверить правильность результата.

10. Опишите рекурсивную функцию, которая по заданным вещественному x и целому n вычисляет величину x^n согласно формуле:

$$x^n = \begin{cases} 1, & n = 0, \\ 1/x^{|n|}, & n < 0, \\ x \cdot (x^{n-1}), & n > 0. \end{cases} \tag{2}$$

11. Задан список положительных чисел, признаком конца которых служит отрицательное число. Используя рекурсию, подсчитать количество чисел и их сумму.
12. Дан список X из n вещественных чисел. Найти минимальный элемент списка,, используя вспомогательную рекурсивную функцию, находящую минимум среди последних элементов списка X, начиная с n-го.
13. Напишите программу вычисления функции Аккермана для всех неотрицательных целых аргументов m и n:

$$A(m,n) = \begin{cases} A(0,n) = n+1 \\ A(m,0) = A(m-1,1), & m \\ A(m,n) = A(m-1, A(m,n-1)), & m,n > 0. \end{cases} \tag{3}$$

14. Напишите рекурсивную функцию, которая вычисляет $y = \sqrt[k]{x}$ по следующей формуле:

$$y_0 = 1; y_{n+1} = y_n + \frac{x/y_n^{k-1} - y_n}{k}, \tag{4}$$

$n = 0, 1, 2, \ldots$. За ответ принять приближение, для которого выполняется $|y_n - y_{n+1}| < \varepsilon$, где $\varepsilon = 0,0001$.

Содержание отчета и его форма

Отчет по лабораторной работе оформляется электронно в формате PDF, должен содержать ответы на контрольные вопросы, ссылку на репозиторий с которым выполнялась работа, скриншоты IDE PyCharm, скриншоты результатов работы программ.

Вопросы для защиты работы

1. Для чего нужна рекурсия?
2. Что называется базой рекурсии?

3. Самостоятельно изучите что является стеком программы. Как используется стек программы при вызове функций?
4. Как получить текущее значение максимальной глубины рекурсии в языке Python?
5. Что произойдет если число рекурсивных вызовов превысит максимальную глубину рекурсии в языке Python?
6. Как изменить максимальную глубину рекурсии в языке Python?
7. Каково назначение декоратора `lru_cache`?
8. Что такое хвостовая рекурсия? Как проводится оптимизация хвостовых вызовов?