ҚАЗАҚСТАН РЕСПУБЛИКАСЫНЫҢ БІЛІМ ЖӘНЕ ҒЫЛЫМ МИНИСТРЛІГІ

СӘТБАЕВ УНИВЕРСИТЕТІ

ИАиИТ институты

Программная инженерия кафедрасы



ЛАБОРАТОРИЯЛЫҚ ЖҰМЫС

Тақырыбы: **«Жетілдірілген рекурсия және функционалдық үлгілер»**

|  |  |  |  |
| --- | --- | --- | --- |
| № | Жұмысты орындау сапасы | Баға диапазоны | Орындалған  % |
| 1 | Орындалған жоқ | 0% |  |
| 2 | Орындалды | 0-50% |  |
| 3 | Материялдық өзіндік жүйелендіру | 0-10% |  |
| 4 | Талап етілген көлемде және көрсетілген мерзімде орындау | 0-5% |  |
| 5 | Қосымша ғылыми әдебиеттерді пайдалану | 0-5% |  |
| 6 | Орындаған тапсырманың ерекшелігі | 0-10% |  |
| 7 | СӨЖ-ді қорғау | 0-20% |  |
|  | Қорытынды: | 0-100% |  |

Оқытушы: Қасенхан А.М.

Студент: Әбибулла Б.М.

Мамандығы: Computer Science

Тобы: вт 12.10-14.05

**4-зертхана:** Жетілдірілген рекурсия және функционалдық үлгілер

**Мақсат:** Python тіліндегі озық рекурсивті әдістер мен функционалдық үлгілерді терең меңгеру және қолдану. Зертхананың мақсаты рекурсивті кодты жазу дағдыларын жақсарту ғана емес, сонымен қатар бағдарламалардың оқылуын, тиімділігін және масштабталуын жақсартатын әртүрлі функционалдық үлгілерді түсіну және қолдану болып табылады.

**Тапсырма:**

10. Сұрыптауға арналған «Стратегия» функционалдық үлгісі - Аргумент ретінде сұрыптау стратегиясын (мысалы, көпіршікті сұрыптау, біріктіру сұрыптау) қабылдайтын функцияны жасаңыз.

def sort(arr, strategy):

if strategy == "bubble":

return sorted(arr)

elif strategy == "merge":

return sorted(arr, key=lambda x: x)

else:

print("белгісіз сұрыптау стратегиясы")

return arr

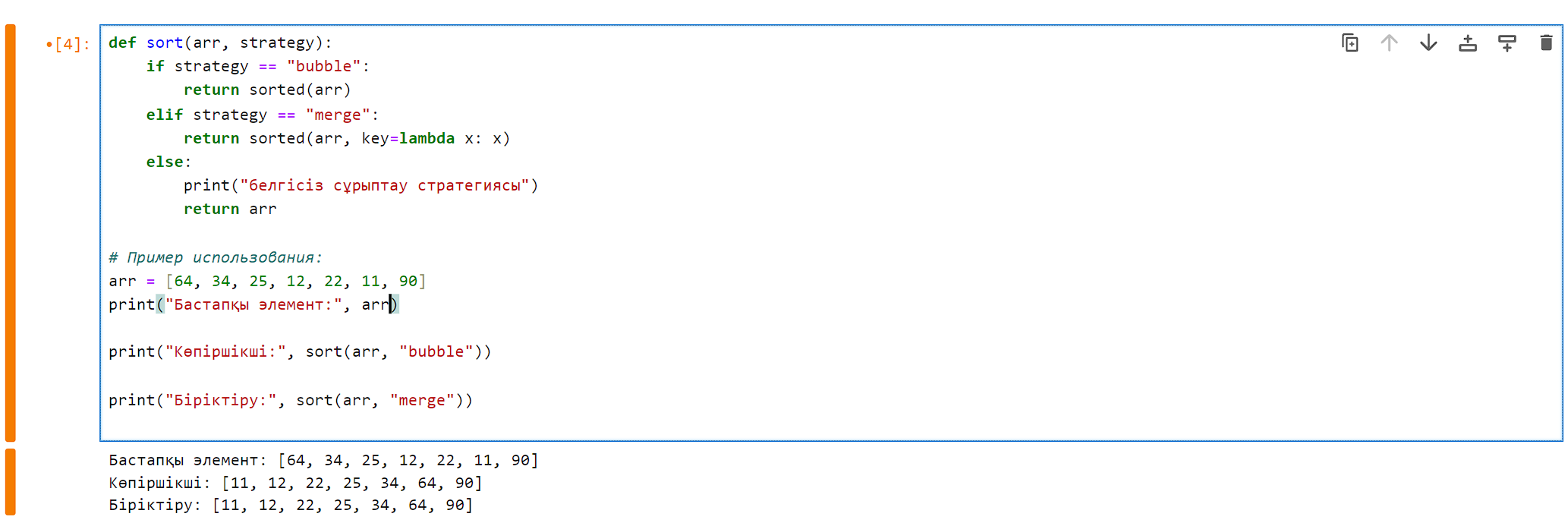
# Пример использования:

arr = [64, 34, 25, 12, 22, 11, 90]

print("Бастапқы элемент:", arr)

print("Көпіршікші:", sort(arr, "bubble"))

print("Біріктіру:", sort(arr, "merge"))



**Дайындық сұрақтары:**

**1тапс. Жетілдірілген рекурсияда қолданылатын негізгі принциптер мен әдістер қандай?**

Жетілдірілген рекурсия рекурсияны тиімдірек қолдануға және күрделі есептерді шешуге мүмкіндік беретін бірқатар принциптер мен әдістерді қамтиды. Жетілдірілген рекурсияның кейбір негізгі принциптері мен әдістеріне мыналар жатады:

**Есте сақтау(мемоизация):** бұл бұрыннан есептелген ішкі мәселелердің нәтижелері кейінірек пайдалану үшін жадта сақталатын оңтайландыру әдісі. Бұл қайталанатын есептеулерді болдырмайды және рекурсивті алгоритмдердің орындалуын айтарлықтай жылдамдатады.

Есте сақтау (Кэш) бағдарламалық жасақтаманы әзірлеуде, бағдарламалауда, қайталанатын есептеулерді болдырмау үшін функцияларды орындау нәтижелерін сақтауда кэшті пайдаланудың мысалы болып табылады.

Есте сақтау - функцияны сол аргументтермен қайта шақырған кезде қайта есептеулерді болдырмау үшін функцияны орындау нәтижелері кэштелетін (есте сақталатын) оңтайландыру әдісі. Рекурсивті функцияларда есте сақтау бірдей кіріс мәндері үшін нәтижелерді қайталап есептеуді болдырмау үшін әсіресе пайдалы болуы мүмкін.

Есте сақтау - қайталанатын есептеулерді болдырмау үшін бұрыннан қарастырылған кіріс мәндері үшін функцияны орындау нәтижелерін сақтаудан тұратын оңтайландыру әдісі. Рекурсивті функцияларда есте сақтауды бұрын есептелген нәтижелерді сақтау үшін сөздікті пайдалану арқылы жүзеге асыруға болады.

Есептеулер нәтижелерін кэштеу үшін сөздікті пайдалану арқылы рекурсивті функцияда есте сақтауды жүзеге асыруға болады. Мұны қалай жасауға болатынының мысалы:

***def fibonacci(n, memo={}):***

***if n in memo:***

***return memo[n]***

***if n <= 1:***

***return n***

***memo[n] = fibonacci(n-1, memo) + fibonacci(n-2, memo)***

***return memo[n]***

***print(fibonacci(10)) # Жауабы: 55***

Бұл мысалда Фибоначчи функциясының есептеулерінің нәтижелерін кэштеу үшін memo сөздігі пайдаланылады. Функция белгілі n мәнімен шақырылғанда, ол алдымен сол мәннің кэште бар-жоғын тексереді. Егер ол бар болса, функция кэштен мәнді алып қайтарады. Олай болмаса, ол мәнді әдеттегідей рекурсивті есептейді және қайтармас бұрын оны кэшке қосады. Бұл бірдей кіріс мәндері үшін қайталанатын есептеулерді болдырмайды.

**Динамикалық бағдарламалау:** Бұл есептерді ішкі мәселелерге бөлу арқылы шешу әдісі, олардың шешімі кейінірек пайдалану үшін сақталады.

**Құйрық рекурсиясы:** Бұл рекурсивті шақыру функциядағы соңғы операция болатын рекурсияның ерекше түрі.

Кейбір бағдарламалау тілдері қоңыраулар стекінің толып кетуін болдырмау үшін рекурсияны автоматты түрде оңтайландырады.

Рекурсия - бұл өзін-өзі пайдалану. Рекурсивті шақыру – функцияның өзіне тікелей немесе жанама шақыруы. Рекурсивті функциялар итерациялық процесті пайдаланатын функциялар болып табылады. Әрбір рекурсивті шақыру үшін қоңыраулар стегі функцияның параметрлері мен айнымалы мәндері сияқты осы шақырумен байланысты барлық ақпаратты жазады.

Құйрық рекурсиясы - рекурсивті шақыру функция кодының соңында орналасқан қарапайым рекурсия.

Құйрық рекурсиясының бір алғы шарты бар - соңғы жол өзін шақыруы керек.

***def factorial\_tail\_recursive(n, accumulator=1):***

***if n == 0:***

***return accumulator***

***else:***

***return factorial\_tail\_recursive(n - 1, n \* accumulator)***

***result = factorial\_tail\_recursive(5)***

***print(result) # Жауабы: 120***

Бұл мысал n факториалын құйрық рекурсиясын пайдаланып есептейтін factorial\_tail\_recursive функциясын ұсынады.

Бұл функцияның қалай жұмыс істейтінін қарастырайық:

Негізгі жағдай: n 0 болса, функция аккумуляторды қайтарады. Бұл рекурсияны аяқтайтын негізгі жағдай.

Рекурсивті жағдай: n 0 болмаса, функция n - 1 және n \* аккумулятор аргументтері арқылы өзін шақырады. Мұндағы n\*аккумулятор – n-тің ағымдағы мәнін есепке алатын аккумулятордың жаңа мәні. Осылайша, әрбір рекурсивті шақыру рекурсияның «құйрығын» өңдейді, мұнда аккумулятор сандарды көбейту нәтижелерін жинақтайды.

Негізгі жағдайға жеткенде (n 0 болғанда), рекурсия аяқталады және нәтиже ретінде аккумулятордың ағымдағы мәні қайтарылады.

***factorial\_tail\_recursive(5) шақыруының мысалы 5 факториалын келесідей есептейді:***

***factorial\_tail\_recursive(5, 1)***

***factorial\_tail\_recursive(4, 5 \* 1)***

***factorial\_tail\_recursive(3, 4 \* 5 \* 1)***

***factorial\_tail\_recursive(2, 3 \* 4 \* 5 \* 1)***

***factorial\_tail\_recursive(1, 2 \* 3 \* 4 \* 5 \* 1)***

***factorial\_tail\_recursive(0, 1 \* 2 \* 3 \* 4 \* 5 \* 1)***

Негізгі жағдайға жеткенде (n 0 болғанда), соңғы рекурсивті шақыру аккумулятордың ағымдағы мәнін қайтарады, ол 5 факториалы, яғни 120.

*В данной функции отличие от обычной рекурсии заключается в том, что операция накопления результата (промежуточных вычислений) выполняется во втором аргументе функции accumulator. Этот аргумент передается при каждом рекурсивном вызове и используется для накопления промежуточных результатов вычислений.*

*В обычной рекурсии результаты каждого рекурсивного вызова просто накапливаются в стеке вызовов до того момента, пока не будет достигнут базовый случай. После этого они начинают раскручиваться в обратном порядке, причем промежуточные результаты вычислений не используются напрямую.*

*В хвостовой рекурсии, как в данной функции, промежуточные результаты вычислений непосредственно передаются и накапливаются в аргументе accumulator, а не в стеке вызовов. После достижения базового случая рекурсия завершается, и результат можно вернуть непосредственно из последнего вызова, минуя раскрутку стека вызовов. Это позволяет эффективнее использовать ресурсы и избегать переполнения стека при работе с большими значениями аргумента n.*

*В рекурсивной функции обычно используется стек вызовов для отслеживания последовательности вызовов функции. Каждый раз, когда функция вызывает саму себя, новый вызов помещается в вершину стека вызовов, а когда базовый случай достигнут, вызовы начинают разворачиваться из вершины стека в обратном порядке.*

**Итеративті декомпозиция:** бұл аралық нәтижелерді сақтау үшін деректер стегін немесе басқа құрылымдарды пайдалана отырып, рекурсивті алгоритм итеративтіге түрленетін әдіс.

Итеративті декомпозицияда рекурсивті шақырулар циклдармен ауыстырылады. Бұл өнімділікті жақсартуға және жадты үнемдеуге әкелуі мүмкін, әсіресе рекурсивті қоңыраулар терең кірістірілген жағдайларда.

Итеративті декомпозицияның мысалы ретінде рекурсивті санның факториалын табуды итеративті түрге түрлендіру:

***def factorial\_iterative(n):***

***result = 1***

***for i in range(1, n + 1):***

***result \*= i***

***return result***

***factorial\_iterative(5)***

**2тапс. Стек толып кетуін болдырмау үшін рекурсивті функцияларды қалай оңтайландыруға болады?**

Стек толып кетуін болдырмау үшін рекурсивті функцияларды оңтайландыруға рекурсивті шақыруды **итеративті шақыруға** айналдыру арқылы қол жеткізуге болады. Мұны рекурсивті функцияны циклге түрлендіру арқылы жасауға болады. Мысалы, факториалды есептеу функциясын қарастырайық:

***def factorial\_recursive(n):***

***if n == 0:***

***return 1***

***else:***

***return n \* factorial\_recursive(n - 1)***

***factorial\_recursive(5)***

Стектің толып кетуін болдырмау үшін бұл функцияны келесідей қайта жазуға болады:

***def factorial\_iterative(n):***

***result = 1***

***for i in range(1, n + 1):***

***result \*= i***

***return result***

***factorial\_iterative(5)***

**Құйрық рекурсиясы:** Бұл рекурсивті шақыру функциясындағы соңғы операция болатын рекурсияның ерекше жағдайы.

***def factorial\_tail\_recursive(n, accumulator=1):***

***if n == 0:***

***return accumulator***

***else:***

***return factorial\_tail\_recursive(n - 1, accumulator \* n)***

***factorial\_tail\_recursive(5)***

**Мемоизация:** Этот подход заключается в сохранении результатов уже вычисленных вызовов функции для последующего использования. Таким образом, функция сначала проверяет, был ли уже вычислен результат для данного входа, и использует сохраненное значение, если оно доступно, вместо выполнения повторного вычисления.

**Есте сақтау(мемоизация):** бұл бұрыннан есептелген ішкі мәселелердің нәтижелері кейінірек пайдалану үшін жадта сақталатын оңтайландыру әдісі. Бұл қайталанатын есептеулерді болдырмайды және рекурсивті алгоритмдердің орындалуын айтарлықтай жылдамдатады.

Есте сақтау (Кэш) бағдарламалық жасақтаманы әзірлеуде, бағдарламалауда, қайталанатын есептеулерді болдырмау үшін функцияларды орындау нәтижелерін сақтауда кэшті пайдаланудың мысалы болып табылады.

Есте сақтау - функцияны сол аргументтермен қайта шақырған кезде қайта есептеулерді болдырмау үшін функцияны орындау нәтижелері кэштелетін (есте сақталатын) оңтайландыру әдісі. Рекурсивті функцияларда есте сақтау бірдей кіріс мәндері үшін нәтижелерді қайталап есептеуді болдырмау үшін әсіресе пайдалы болуы мүмкін.

Есте сақтау - қайталанатын есептеулерді болдырмау үшін бұрыннан қарастырылған кіріс мәндері үшін функцияны орындау нәтижелерін сақтаудан тұратын оңтайландыру әдісі. Рекурсивті функцияларда есте сақтауды бұрын есептелген нәтижелерді сақтау үшін сөздікті пайдалану арқылы жүзеге асыруға болады.

***memo = {}***

***def fibonacci\_memoized(n):***

***if n in memo:***

***return memo[n]***

***if n <= 1:***

***return n***

***memo[n] = fibonacci\_memoized(n - 1) + fibonacci\_memoized(n - 2)***

***return memo[n]***

***fibonacci\_memoized(5)***

Бұл әдістер рекурсияны пайдаланған кезде стектің толып кетуін болдырмауға көмектеседі.

**3тапс. Функционалдық декоратор үлгісі дегеніміз не және ол Python тілінде қалай қолданылады?**

Python тіліндегі декораторлар - басқа функциялардың немесе әдістердің әрекетін өзгертуге мүмкіндік беретін жоғары дәрежелі функциялар. Олар аргумент ретінде функцияны қабылдайды және оның кодын өзгертпестен бастапқы функцияға кейбір функцияларды қоса отырып, жаңа функцияны қайтарады.

• Декоратор @ белгісімен және одан кейін декоратордың атымен белгіленеді және функция анықтамасының үстінде қойылады.

Функционалдық декоратор үлгісі - бұл бар нысандарға немесе функцияларға оларды өзгертпестен жаңа мінез-құлық немесе функционалдылықты қосуға мүмкіндік беретін бағдарламалаудағы тұжырымдама. Python тілінде декораторлар басқа функцияны аргумент ретінде қабылдайтын және әдетте берілген функцияға кейбір қосымша әрекетті қосатын жаңа функцияны қайтаратын функциялар ретінде жүзеге асырылады.

*Декораторы в Python — это функции высшего порядка, которые позволяют модифицировать поведение других функций или методов. Они принимают функцию в качестве аргумента и возвращают новую функцию, добавляя некоторую функциональность к исходной функции без изменения её кода.*

*• Декоратор обозначается символом @, за которым следует имя декоратора и размещается над определением функции.*

*Функциональный паттерн декоратора - это концепция в программировании, которая позволяет добавлять новое поведение или функциональность существующим объектам или функциям без их изменения. В Python декораторы реализуются как функции, которые принимают другую функцию в качестве аргумента и возвращают новую функцию, которая обычно добавляет какое-то дополнительное поведение к переданной функции.*

Python-да декораторларды пайдаланудың мысалы:

***def my\_decorator(func):***

***def wrapper():***

***print("Что-то происходит до вызова функции.")***

***func()***

***print("Что-то происходит после вызова функции.")***

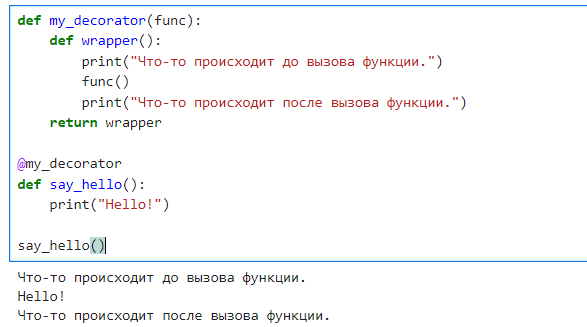
***return wrapper***

***@my\_decorator***

***def say\_hello():***

***print("Hello!")***

***say\_hello()***

******

Бұл мысалда **my\_decorator** — функцияны аргумент ретінде қабылдайтын, func шақырылғанға дейін және кейін қосымша әрекетті қосатын және оны қайтаратын жаңа wrapper функциясын жасайтын декоратор. Say\_hello функциясын жарияламас бұрын **@** белгісі бар декораторды пайдалану арқылы say\_hello функциясы декораторға оралады және say\_hello() шақырылғанда, алдымен қосымша әрекет орындалады, содан кейін say\_hello() функциясының өзі шақырылады.

Декораторлар Python-да кеширование, логирование, рұқсаттарды тексеру, ерекше жағдайларды өңдеу және т.б. сияқты әртүрлі функционалдылықты жүзеге асыру үшін кеңінен қолданылады.

**4тапс. Python-да жалқау бағалаудың артықшылықтары қандай және олар қалай жүзеге асырылады?**

**Жалқау бағалау** - нәтиже нақты қажет болғанша есептеу кейінге қалдырылатын тәсіл. Python тілінде жалқау бағалау көбінесе генераторлар, итераторлар және жоғары ретті функциялар (мысалы, map(), filter(), zip() функциялары және генератор өрнектері) арқылы жүзеге асырылады.

Python тіліндегі жалқау бағалаудың артықшылықтары мыналарды қамтиды:

***Ресурстарды үнемдеу:*** Жалқау бағалау қажетсіз есептеулерді болдырмайды, бұл жад пен процессорды тұтынуды азайтады.

***Өнімділікті арттыру:*** Нәтиже алу үшін қажетті операцияларды орындауды кейінге қалдыру арқылы бағдарлама тиімдірек жұмыс істей алады.

***Үлкен деректерді өңдеу:*** Жалқау бағалау әсіресе деректердің үлкен көлемімен жұмыс істегенде пайдалы, өйткені ол барлық деректер жинағын бірден жадқа жүктеуді болдырмайды.

Python-да жалқау бағалауды іске асыру мысалдары:

**Генераторлар:**

***data = [1, 2, 3, 4, 5]***

***# Пример использования функции map для ленивого вычисления квадратов чисел***

***squared = map(lambda x: x \*\* 2, data)***

***for num in squared:***

***if num > 9:***

***break # Прервать цикл, если квадрат числа больше 9***

***print(num)***

Мен итератор болып табылатын squared нысанын жасаймын.

Дегенмен, map() сандардың квадраттарын бірден есептемейді. Оның орнына ол келесі элементке сұрауды күтеді.

Мен for num in squared функциясын іске қосқан кезде: цикл, map() сандардың квадраттарын есептей бастайды және оларды қажетінше қайтарады.

Егер мен for циклін үзсем, есептеу тоқтайды және map() енді келесі элементтерді есептемейді.

Бұл жалқау бағалаудың мәні: мәндер нақты қажет болғанда ғана есептеледі, бұл ресурстарды үнемдеуге және тиімділікті арттыруға көмектеседі.

**Жоғары ретті функциялар:**

***def generate\_numbers():***

***for i in range(10):***

***yield i***

***numbers = generate\_numbers()***

***for num in numbers:***

***if num >= 5:***

***break # Прервать итерацию, если значение больше или равно 5***

***print(num)***

Ұсынылған кодта жалқау бағалау генераторды пайдалану арқылы жүзеге асырылады. generate\_numbers() функциясы мәнді қайтаратын және келесі элемент шақырылғанша функцияны уақытша тоқтататын yield құрылымын пайдаланады.

Бұл жағдайда 5 мәніне жеткенде, итерация break операторымен үзіледі, яғни барлық сандар өңделмейді. Осылайша, есептеулер жалқау болады, өйткені олар тек үзу шарты орындалғанша орындалады. Бұл деректердің үлкен көлемімен жұмыс істеу кезінде ресурстарды тиімді пайдалануға мүмкіндік беретін деректерді өңдеуге жалқау көзқарастың мысалы.

Әрбір элемент жад пен ресурстарды үнемдей отырып, for циклінде сұралғанда ғана жасалады.

**5тапс. Стратегияның функционалдық үлгісін енгізу және пайдалану жолын түсіндіре аласыз ба?**

Стратегия үлгісі (Паттерн "Стратегия" (Strategy pattern) негізгі классты өзгертпей, жағдайға немесе енгізілген деректерге байланысты бағдарламаның әрекетін өзгертуге мүмкіндік береді.

Стратегия үлгісі - алгоритмдер тобын анықтауға, олардың әрқайсысын инкапсуляциялауға және оларды өзара алмастыруға мүмкіндік беретін мінез-құлық дизайны үлгісі. Бұл бағдарламаны орындау кезінде қажетті алгоритмді таңдауға мүмкіндік береді.

Ол класстар мен методтардың орнына әртүрлі стратегияларды көрсететін функцияларды пайдаланады.

***def strategy\_add(a, b):***

***return a + b***

***def strategy\_subtract(a, b):***

***return a - b***

***def strategy\_multiply(a, b):***

***return a \* b***

***# Стратегияны қолданатын контекст***

***def apply\_strategy(a, b, strategy):***

***return strategy(a, b)***

***# Стратегияларды қолдану***

***result\_add = apply\_strategy(5, 3, strategy\_add)***

***result\_subtract = apply\_strategy(5, 3, strategy\_subtract)***

***result\_multiply = apply\_strategy(5, 3, strategy\_multiply)***

***print("Қосу нәтижесі:", result\_add)***

***print("Азайту нәтижесі:", result\_subtract)***

***print("Көбейту нәтижесі:", result\_multiply)***

Бұл мысалда бізде үш түрлі стратегия функциясы бар (strategy\_add, strategy\_subtract, strategy\_multiply) олардың әрқайсысы белгілі бір математикалық операцияны орындайды. Содан кейін бізде екі санды және стратегия функциясын аргумент ретінде қабылдайтын және сол стратегияны сол сандарға қолданатын apply\_strategy функциясы бар.

Біз apply\_strategy әртүрлі стратегиялармен шақырғанда, таңдалған стратегияға байланысты әртүрлі нәтижелерді аламыз. Ол аргументтер ретінде екі санды және стратегия функциясын алады, содан кейін сол сандарды аргумент ретінде жіберіп, сол функцияны шақырады. Осылайша, бұл мысалда стратегия белгілі бір операцияны (қосу, алу немесе көбейту) орындау жолын анықтайды және контекст сол операцияны орындау үшін таңдалған стратегияны пайдаланады. Бұл контексте стратегияларды оңай ауыстыруға мүмкіндік береді, бұл кодыңыздағы икемділік пен модульдікке мүмкіндік береді. Әртүрлі стратегияларды пайдалана отырып, біз контекстіміздің әрекетін оның өзін өзгертпей-ақ оңай өзгерте аламыз, бұл кодтың икемділігі мен модульділігін қамтамасыз етеді.

**6тапс. Рекурсиямен жұмыс істегенде қандай қиындықтар туындауы мүмкін және оларды қалай шешуге болады?**

Рекурсиямен жұмыс істегенде әртүрлі қиындықтар туындауы мүмкін, мысалы:

***Қоңыраулар стекінің толып кетуі:*** Егер рекурсивті функция өзін тым көп шақырса немесе рекурсия аяқталмаса, ол қоңыраулар стегінің толып кетуіне және бағдарламаның бұзылуына әкелуі мүмкін.

***Қате жазылған рекурсиядан шығу шарты:*** Егер рекурсиядан шығу шарты қате жазылса, функция шексіз шақырылады, бұл да шақыру стекінің толып кетуіне әкеледі.

***Жадты тұтыну:*** рекурсивті функцияға әрбір шақыру стекте функцияның жаңа экземплярын жасайды. Бұл үлкен кіріс деректерімен немесе терең рекурсиялармен жұмыс істегенде үлкен жадты пайдалануға және стектің толып кетуіне әкелуі мүмкін.

***Баяу өнімділік:*** Рекурсияны пайдалану функция шақыруларының қосымша шығындарына байланысты баяу өнімділікке әкелуі мүмкін. Кейбір жағдайларда итерацияны пайдалану тиімдірек болуы мүмкін.

***Түзету қиындығы****:* рекурсивті функцияларды жасырын логикасына байланысты жөндеу қиын болуы мүмкін. Негізгі жағдайда немесе рекурсия қадамындағы қате қате нәтижелерге немесе шексіз рекурсияға алып келуі мүмкін, бұл жөндеуді қиындатады.

Бұл мәселелерді шешу үшін келесі әдістерді қолдануға болады:

***Рекурсиядан шығу шартын дұрыс жазыңыз:*** Рекурсиядан шығу шарты дұрыс жазылғанын және оның рекурсивті функцияңызда ең соңында жеткенін тексеріңіз.

***Рекурсивті функцияны оңтайландыру:*** Қоңыраулар санын немесе ресурстарды пайдалануды азайту үшін рекурсивті функцияны оңтайландырып көріңіз. Мысалы, сіз құйрықты рекурсияны пайдалана аласыз.

***Құйрық рекурсиясын*** (Хвостовая рекурсия) пайдаланыңыз: Құйрық рекурсиясы - рекурсивті шақыру функциядағы соңғы әрекет болатын рекурсияның ерекше жағдайы. Кейбір бағдарламалау тілдері қоңыраулар стекінің толып кетуін болдырмау үшін рекурсияны автоматты түрде оңтайландырады.

***def factorial\_tail\_recursive(n, accumulator=1):***

***if n == 0:***

***return accumulator***

***else:***

***return factorial\_tail\_recursive(n - 1, n \* accumulator)***

***result = factorial\_tail\_recursive(5)***

***print(result) # Жауабы: 120***

Бұл мысал n факториалын құйрық рекурсиясын пайдаланып есептейтін factorial\_tail\_recursive функциясын ұсынады.

Бұл функцияның қалай жұмыс істейтінін қарастырайық:

Негізгі жағдай: n 0 болса, функция аккумуляторды қайтарады. Бұл рекурсияны аяқтайтын негізгі жағдай.

Рекурсивті жағдай: n 0 болмаса, функция n - 1 және n \* аккумулятор аргументтері арқылы өзін шақырады. Мұндағы n\*аккумулятор – n-тің ағымдағы мәнін есепке алатын аккумулятордың жаңа мәні. Осылайша, әрбір рекурсивті шақыру рекурсияның «құйрығын» өңдейді, мұнда аккумулятор сандарды көбейту нәтижелерін жинақтайды.

Негізгі жағдайға жеткенде (n 0 болғанда), рекурсия аяқталады және нәтиже ретінде аккумулятордың ағымдағы мәні қайтарылады.

factorial\_tail\_recursive(5) шақыруының мысалы 5 факториалын келесідей есептейді:

factorial\_tail\_recursive(5, 1)

factorial\_tail\_recursive(4, 5 \* 1)

factorial\_tail\_recursive(3, 4 \* 5 \* 1)

factorial\_tail\_recursive(2, 3 \* 4 \* 5 \* 1)

factorial\_tail\_recursive(1, 2 \* 3 \* 4 \* 5 \* 1)

factorial\_tail\_recursive(0, 1 \* 2 \* 3 \* 4 \* 5 \* 1)

Негізгі жағдайға жеткенде (n 0 болғанда), соңғы рекурсивті шақыру аккумулятордың ағымдағы мәнін қайтарады, ол 5 факториалы, яғни 120.

*В данной функции отличие от обычной рекурсии заключается в том, что операция накопления результата (промежуточных вычислений) выполняется во втором аргументе функции accumulator. Этот аргумент передается при каждом рекурсивном вызове и используется для накопления промежуточных результатов вычислений.*

*В обычной рекурсии результаты каждого рекурсивного вызова просто накапливаются в стеке вызовов до того момента, пока не будет достигнут базовый случай. После этого они начинают раскручиваться в обратном порядке, причем промежуточные результаты вычислений не используются напрямую.*

*В хвостовой рекурсии, как в данной функции, промежуточные результаты вычислений непосредственно передаются и накапливаются в аргументе accumulator, а не в стеке вызовов. После достижения базового случая рекурсия завершается, и результат можно вернуть непосредственно из последнего вызова, минуя раскрутку стека вызовов. Это позволяет эффективнее использовать ресурсы и избегать переполнения стека при работе с большими значениями аргумента n.*

*В рекурсивной функции обычно используется стек вызовов для отслеживания последовательности вызовов функции. Каждый раз, когда функция вызывает саму себя, новый вызов помещается в вершину стека вызовов, а когда базовый случай достигнут, вызовы начинают разворачиваться из вершины стека в обратном порядке.*

***Итеративті тәсілді пайдалану:*** Кейбір жағдайларда, әсіресе рекурсия қажет болмаса, рекурсивті емес, қайталанатын тәсілді қолдануға болады, қайталанатын тапсырмаларды орындау үшін рекурсияның орнына циклдарды пайдалануға болады. Бұл өнімділікті жақсартады және қоңыраулар стекінің толып кетуін болдырмайды.

***Рекурсия тереңдігінің шегі:*** Мүмкін болса, қоңыраулар стекінің толып кетуін болдырмау үшін максималды рекурсия тереңдігін орнатыңыз немесе рекурсивті қоңыраулар санына шектеулер орнатыңыз.

Рекурсиялық есептерді шешудің жалпы тәсілі оның дұрыс және тиімді жұмыс істейтініне көз жеткізу үшін **кодты мұқият жобалау және тексеру** болып табылады.

**7тапс. Бағдарламалауда Observer үлгісін қолдануға мысал келтіріңіз.**

Observer үлгісі бір нысанның күйі өзгерген кезде оған тәуелді барлық нысандар автоматты түрде хабарланатын және жаңартылатындай нысандар арасында бірдің-көпке("один ко многим") тәуелділігін орнату үшін қолданылады.

Сөйлесу қолданбасында хабарландыру жүйесін енгізу үшін Observer үлгісін пайдалану. Бізде қарапайым сөйлесу қолданбасы(чат-приложение) бар деп елестетіп көрейік, онда пайдаланушылар хабарламалар жібере алады және басқа пайдаланушылар жаңа хабарламалар туралы хабардар болуы керек.

***# Интерфейс для наблюдателя***

***class Observer:***

***def update(self, message):***

***pass***

***#*** ***Бұл хабарландыру алынған кезде шақырылатын жаңарту әдісі бар Observer интерфейсін анықтайды.***

***# Конкретный наблюдатель (пользователь)***

***class User(Observer):***

***def \_\_init\_\_(self, name):***

***self.name = name***

***def update(self, message):***

***print(f"{self.name} получил сообщение: {message}")***

***#*** ***Бұл User класы белгілі бір бақылаушыны (чат пайдаланушысын) білдіреді. Ол Observer интерфейсінің мұрагері және консольге хабарлама басып шығаратын жаңарту әдісін жүзеге асырады.***

***# Конкретный субъект (чат)***

***class Chat:***

***def \_\_init\_\_(self):***

***self.observers = []***

***def attach(self, observer):***

***self.observers.append(observer)***

***def detach(self, observer):***

***self.observers.remove(observer)***

***def notify(self, message):***

***for observer in self.observers:***

***observer.update(message)***

***def send\_message(self, message):***

***print(f"Отправлено сообщение: {message}")***

***self.notify(message)***

***#Класс Chat представляет субъект (чат), за который могут наблюдать пользователи. Он имеет методы attach и detach для подписки и отписки пользователей от чата, метод notify для уведомления всех наблюдателей о новом сообщении, и метод send\_message, который отправляет сообщение и вызывает метод notify.***

***# Создание объектов наблюдателей (пользователей)***

***user1 = User("Петя")***

***user2 = User("Вася")***

***# Создание объекта субъекта (чата)***

***chat = Chat()***

***# Подписываем пользователей на чат***

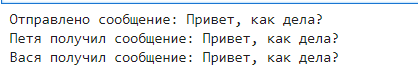
***chat.attach(user1)***

***chat.attach(user2)***

***# Пользователь отправляет сообщение, которое уведомляется всем пользователям***

***chat.send\_message("Привет, как дела?")***

***#Этот блок кода создает двух пользователей user1 и user2, а затем создает объект чата chat. После этого оба пользователя подписываются на чат с помощью метода attach, и отправляется сообщение с помощью метода send\_message, что приводит к уведомлению обоих пользователей.***

******

Сонымен, біз пайдаланушылар жаңа хабарлар туралы хабарландыру ала алатын Observer үлгісін пайдаланып чат қолданбасында қарапайым хабарландыру жүйесін жасадық.

**8тапс. Рекурсия көмегімен қандай есептер тиімді шешіледі?**

Бір типті ішкі мәселелерге бөлуге болатын есептерді шешу үшін рекурсия жиі қолданылады. Рекурсия көмегімен ең тиімді шешілетін есептердің бірнеше түрі:

**Ағаштар мен графтарді айналып өту:** рекурсия ағаштар мен графтарді тереңдік немесе ені бойынша өту үшін тиімді қолданылады. Бұл пайдалы болуы мүмкін, мысалы, тереңдікке іздеуде(поиск в глубину), ағаштың биіктігін есептеуде, екі шыңның арасындағы жолды табуда және т.б.

**Әртүрлі ағаш мәселелері:** максимум/мин мәнді табу, екілік іздеу ағашын (BST) құру, тепе-теңдікті тексеру, түйіндер/жапырақтардың санын есептеу және т.б. сияқты ағашқа қатысты көптеген мәселелерді рекурсия арқылы тиімді шешуге болады.

**Динамикалық бағдарламалау есептерін шешу: Динамикалық** бағдарламалау арқылы шешуге болатын кейбір есептерді рекурсивті түрде де шешуге болады. Мысалы, Фибоначчи сандарын табу мәселесі.

**Тереңдікке іздеу(поиск в глубину**): рекурсия көбінесе графтарда немесе ағаштарда тереңдікке іздеу үшін пайдаланылады.

**Әртүрлі математикалық есептер:** Мысалы, санның факториалын есептеу, Фибоначчи сандарын есептеу, т.б.

Рекурсия ең жақсы шешім болып табылатын есептердің бірі - Фибоначчи сандарын есептеу.

***Фибоначчи сандары*** әрбір сан алдыңғы екі санның қосындысына тең болатын тізбекті құрайды.

F(n)=F(n−1)+F(n−2)

бұнда F(0)=0 және F(1)=1.

Фибоначчи сандарын есептеуге арналған рекурсивті шешім келесідей:

def fibonacci(n):

if n <= 1:

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

result = fibonacci(6)

print(result) #Жауабы: 8

0 1 1 2 3 5 8

Бұл шешімде негізгі жағдайлар F(0)=0 және F(1)=1 болып табылады, олар әрі қарай рекурсивті шақыруларсыз тікелей қайтарылады. Рекурсия қадамы F(n-1) және F(n-2) үшін функцияны рекурсивті шақыру және олардың қосындысын қайтару болып табылады.

Бұл рекурсивті шешім кодтың анықтығы мен қысқалығы тұрғысынан жақсы.

***Санның факториалын есептеу.***

Факториал числа n (обозначается как n!) определяется как произведение всех положительных целых чисел от 1 до n. Формально:

n!=n×(n−1)×(n−2)×…×2×1

Факториалды есептеу үшін рекурсивті функцияның мысалы:

def factorial(n):

if n == 0:

return 1

else:

return n \* factorial(n - 1)

result = factorial(5)

print(result) #120

1-ден n-ге дейінгі сандардың қосындысын есептеуге арналған рекурсивті функцияның мысалы:

def sum(n):

if n == 1:

return 1

else:

return n + sum(n - 1)

print(sum(5)) # Жауабы: 15 (1 + 2 + 3 + 4 + 5)

Бұл функция 1-ден n-ге дейінгі сандардың қосындысын есептейді. Егер n 1 болса, функция 1 мәнін қайтарады (негізгі регистр). Әйтпесе, функция 1-ден n-1-ге дейінгі сандардың қосындысын есептеу үшін өзін шақырады және сол қосындыға n мәнін қосады.

Дегенмен, рекурсияны дұрыс пайдаланбау стектің толып кетуіне әкелуі мүмкін.

**9тапс. Рекурсия программаның өнімділігіне және жадты пайдалануға қалай әсер етеді?**

**Рекурсивті тәсіл** - бұл негізгі жағдайға жеткенше функция (өзгеретін) параметрлермен өзін шақыратын мәселені шешу әдісі, содан кейін нәтиже қайтарылады.

Рекурсияның **артықшылықтары:**

***Кодтың қарапайымдылығы және анықтығы:*** Рекурсивті шешімдер итерациялық шешімдерге қарағанда қарапайым және таза болуы мүмкін. Рекурсивті тәсілді түсіну және жазу оңайырақ болуы мүмкін, әсіресе тапсырма ішкі тапсырмаларға бөлінген болса.

***Күрделі есептерді шешуге жақсы:*** Рекурсия ағаш деректер құрылымдарын айналып өту, генерация мен комбинацияларды құру және математикалық және логикалық есептеулерді қамтитын есептерді шешу сияқты күрделі есептерді шешуге арналған қуатты құрал бола алады.

Рекурсия часто используется для обхода древовидных структур данных, таких как деревья и графы. Обход дерева может быть выполнен с помощью двух основных подходов: обход в глубину (Depth-First Search, DFS) и обход в ширину (Breadth-First Search, BFS).

***Оқылуы:*** Рекурсивті код ықшамырақ(компактный) және оқуға оңай, әсіресе факториалды есептеу немесе ағаш өтуі сияқты рекурсияның көмегімен шешілетін мәселелер үшін.

***Ықшам, аз код:*** Кейбір жағдайларда рекурсивті шешімдер итерациялық шешімдерге қарағанда ықшам болуы мүмкін және кодты аз талап етеді.

Рекурсияның **кемшіліктері:**

***Жадты тұтыну:*** рекурсивті функцияға әрбір шақыру стекте функцияның жаңа экземплярын жасайды. Бұл үлкен кіріс деректерімен немесе терең рекурсиялармен жұмыс істегенде үлкен жадты пайдалануға және стектің толып кетуіне әкелуі мүмкін.

***Баяу өнімділік:*** Рекурсияны пайдалану функция шақыруларының қосымша шығындарына байланысты баяу өнімділікке әкелуі мүмкін. Кейбір жағдайларда итерацияны пайдалану тиімдірек болуы мүмкін.

***Түзету қиындығы***: рекурсивті функцияларды жасырын логикасына байланысты жөндеу қиын болуы мүмкін. Негізгі жағдайда немесе рекурсия қадамындағы қате қате нәтижелерге немесе шексіз рекурсияға алып келуі мүмкін, бұл жөндеуді қиындатады.

***Қоңыраулар тереңдігі(Глубина вызовов):*** Рекурсивті тәсіл қоңыраулар стекінің максималды тереңдігімен шектелуі мүмкін, бұл терең рекурсия жағдайында стек толып кетуіне әкелуі мүмкін.

**10тапс. Функционалдық үлгілер бағдарламалық жасақтама дизайны мен архитектурасын қалай жақсарта алатынын түсіндіріңіз.**

Функционалдық үлгілер түсінігі функционалдық бағдарламалаумен тығыз байланысты және функционалдық бағдарламалау стиліндегі жалпы есептерді шешуге көмектесетін дизайн үлгілерін білдіреді. Міне, кейбір негізгі функционалдық үлгілер:

**Декораторлар** - Python тіліндегі тағы бір қуатты функционалды үлгі. Декораторлар функциялардың немесе әдістердің әрекетін оларды басқа функцияларға орау арқылы өзгертуге мүмкіндік береді.

**Стратегия үлгісі** (Паттерн "Стратегия" (Strategy Pattern): - алгоритмдер жиынын өз сыныптарына орналастыратын және оларды бір-бірімен алмастыруға болатын мінез-құлық үлгісі. Бұл клиент кодына қажетті алгоритмді жылдам таңдауға мүмкіндік береді. Стратегия Python кодында жиі пайдаланылады, әсіресе бағдарлама жұмыс істеп тұрған кезде алгоритмді өзгерту қажет болған жағдайда.

**Бақылаушы үлгісі** (Паттерн "Наблюдатель" (Observer Pattern): Бұл үлгі бір нысанның күйі өзгерген кезде, оған тәуелді барлық нысандар автоматты түрде хабардар етілетін және жаңартылатындай етіп объектілер арасындағы «бірден көпке» тәуелділікті анықтауға мүмкіндік береді. Нысандарға басқа нысандарда болып жатқан оқиғаларды бақылауға және оларға әрекет етуге мүмкіндік беретін механизм жасайды.

**Map-Reduce:** Бұл үлгі деректер жинағының (map) әрбір элементіне функцияны қолдануды және одан кейін нәтижелерді біріктіруді (reduce) қамтиды. Ол деректерді өңдеу және параллельді бағдарламалау үшін кеңінен қолданылады.

**Жоғары ретті функциялар:** Бұл үлгі функцияларды басқа функцияларға дәлел ретінде пайдалануды немесе оларды басқа функциялардан қайтаруды қамтиды. Ол әрекетті функция ретінде беруге мүмкіндік береді.

**Жалқау бағалау (ленивое вычисление):** Бұл үлгі мәнді нақты пайдаланғанша есептеуді кейінге қалдыруға мүмкіндік береді. Бұл шексіз деректер ағындарымен тиімді жұмыс істеу және ресурстарды пайдалануды оңтайландыру үшін пайдалы.

**Рекурсия:** Бұл үлгі өзін шақыру үшін функцияны пайдалануды қамтиды. Ол тізімдер мен ағаштар сияқты деректер құрылымдарын айналып өту үшін кеңінен қолданылады.

**Lambda функциялары**: тек бір өрнекті қамтитын анонимді функциялар. Олар map, filter, sorted және т.б. сияқты жоғары ретті функцияларға аргумент ретінде пайдалы.

**Есте сақтау:** бірдей аргументтер үшін қайталанатын есептеулерді болдырмау үшін функция нәтижелері есте сақталатын (кэштелген) оңтайландыру әдісі.

*Фабричный метод (Factory Method): Этот паттерн позволяет создавать объекты без явного указания их классов. Вместо этого используется метод, который создает экземпляры классов.*

*Абстрактная фабрика (Abstract Factory): Этот паттерн предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.*

*Одиночка (Singleton): Одиночка - это паттерн, который гарантирует, что класс имеет только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.*

*Строитель (Builder): Паттерн, который позволяет создавать сложные объекты пошагово. Он позволяет использовать один и тот же код построения для создания различных представлений объектов.*

*Прототип (Prototype): Прототип - это паттерн, который позволяет создавать новые объекты на основе существующих экземпляров путем копирования.*

*Адаптер (Adapter): Этот паттерн позволяет объектам с несовместимыми интерфейсами работать вместе. Он используется для обертывания одного интерфейса в другой, совместимый с клиентом.*

*Компоновщик (Composite): Паттерн, который позволяет сгруппировать объекты в древовидные структуры для представления их в виде единого объекта.*

*Мост (Bridge): Этот паттерн разделяет абстракцию от ее реализации таким образом, чтобы они могли изменяться независимо друг от друга.*

Функционалдық үлгілер жалпы мәселелерді шешуге құрылымдық және икемді тәсілдерді ұсыну арқылы бағдарламалық жасақтама дизайны мен архитектурасын айтарлықтай жақсарта алады. Міне, мұның бірнеше жолы бар:

**Модульдік және кодты қайта пайдалану:** Функционалды үлгілер бағдарламаны бағдарламаның әртүрлі бөліктерінде қайта пайдалануға болатын шағын, тәуелсіз функцияларға бөлуді болдырады. Бұл кодты түсінуді және өзгертуді жеңілдетеді, оны икемді және модульді етеді.

**Параллельді және бөлінген бағдарламалауды қолдау:** Функционалдық үлгілер параллель және бөлінген жүйелерді құруды жеңілдетеді.

**Функция құрамы(композиция):** Функция үлгілері оңай біріктірілетін және қайта пайдалануға болатын шағын, тәуелсіз функциялардан бағдарлама құруға мүмкіндік береді.

**Кодтың мәнерлілігі мен түсініктілігінің жоғарылауы:** Функционалдық үлгілерді пайдалану оқуға және түсінуге оңай кодты жасауға көмектеседі. Жоғары ретті функциялар, функция құрамы және функционалдық үлгілер күрделі ұғымдар мен операцияларды қарапайым, түсінікті етіп көрсетуге мүмкіндік береді.

**Код күрделілігін азайту:** Функционалдық үлгілер код күрделілігін азайтуға көмектеседі, себебі олар тапсырманы кішірек, басқарылатын бөліктерге бөлуге мүмкіндік береді.

Конечная цель данной функции generateParenthesis(n) - сгенерировать все правильные комбинации скобок длины 2\*n.

**def generateParenthesis(n):**

Это объявление функции generateParenthesis, которая принимает один аргумент n, представляющий количество пар скобок.

**def generate\_helper(open\_count, close\_count, current):**

Это внутренняя функция generate\_helper, которая используется для рекурсивного генерирования комбинаций скобок. Она принимает три аргумента: open\_count - количество оставшихся открывающих скобок, close\_count - количество оставшихся закрывающих скобок, current - текущая строка скобок.

**if open\_count == 0 and close\_count == 0:**

**result.append(current)**

**return**

Это базовый случай рекурсии. Если количество открывающих и закрывающих скобок равно 0, это означает, что мы завершили построение одной комбинации скобок. Мы добавляем текущую строку скобок в список result и завершаем выполнение функции.

**if open\_count > 0:**

**generate\_helper(open\_count - 1, close\_count, current + '(')**

Это условие рекурсии для добавления открывающей скобки. Если у нас остались открывающие скобки (open\_count > 0), мы вызываем generate\_helper с уменьшенным на 1 количеством открывающих скобок, неизменным количеством закрывающих скобок и текущей строкой, к которой мы добавляем открывающую скобку.

**if open\_count < close\_count:**

**generate\_helper(open\_count, close\_count - 1, current + ')')**

Это условие рекурсии для добавления закрывающей скобки. Если количество открывающих скобок меньше количества закрывающих скобок (open\_count < close\_count), мы вызываем generate\_helper с неизменным количеством открывающих скобок, уменьшенным на 1 количеством закрывающих скобок и текущей строкой, к которой мы добавляем закрывающую скобку.

**result = []**

**generate\_helper(n, n, '')**

Мы создаем пустой список result и вызываем generate\_helper с начальным количеством открывающих и закрывающих скобок, равным n, и пустой строкой '' в качестве начальной строки.

**return result**

Мы возвращаем список result, содержащий все сгенерированные комбинации скобок.