GeekBrains

Лекция 1. Введение и основные типы парадигм

Парадигмы программирования и языки парадигм

Оглавление

Введение

Вступление к курсу в целом

Добрый день, коллеги! Сегодняшняя лекция открывает курс “Парадигмы программирования и языки парадигм”.

Меня зовут Александр Левин, я исследователь данных. В основном я занимаюсь обучением моделей машинного обучения на Python и оптимизацией логистики с помощью искусственного интеллекта, и большУю часть времени я занимаюсь разработкой.

Возможно вы уже слышали например такие словосочетания как:

- Объектно-ориентированное программирование (или “ООП”)
- Процедурное программирование
- или Декларативное программирование

В рамках данного курса мы разберём каждый из этих и некоторые другие термины более детально, с помощью определений, на примере реальных кейсов из жизни и задач разработки. Данный курс предназначен для тех, кто уже знаком с базовыми аспектами разработки и хочет расширять кругозор в этой области.

Вообще считается, что разработчику для своего развития стоит расширять именно множество знакомых ему парадигм программирования, так как на каких бы языках вы не программировали внутри одной парадигмы, принципы в них остаются примерно одни и те же. Наилучшим образом данную мысль иллюстрирует пример множества естественных языков: носителю русского языка не составит труда понять белорусскую речь, но при встрече с китайским или например арабским языком, он почувствует растерянность, причём не в отдельных предложениях, а в целом, так как данные языки принадлежат альтернативным языковым группам, которые основаны на разных принципах, разных языковых оборотах, фонетике, алфавите и так далее. Поэтому одной из главных целей данного курса является знакомство слушателей с альтернативными подходами, и помочь научиться выбирать наиболее подходящий в зависимости от имеющейся задачи.

Теперь обсудим план на сегодня.

Вступление к лекции

В сегодняшней лекции мы ответим на вопрос что такое парадигмы программирования и зачем они нужны. Погрузимся в историю возникновения данного термина. Напоследок разберёмся в основных парадигмах программирования на конкретных примерах, а также начнём заглядывать в императивные парадигмы, после чего подведём итоги лекции.

И предлагаю начать со знакомства с определением термина “парадигмы программирования”.

Термины, используемые в лекции

Парадигма – это множество идей, методов, подходов к решению или исследованию конкретных научных задач.

Программирование – процесс написание программного кода.

Парадигма программирования – это набор правил, концепций, идей и методов, определяющих способ написания программ.

Мультипарадигменный язык программирования – язык программирования поддерживающий одновременную разработку в нескольких парадигмах.

Чистый язык программирования – язык программирования поддерживающий только одну парадигму.

Смешанный стиль программирования - стиль разработки с использованием более одной парадигмы.

Императивное программирование – это парадигма программирования, которой свойственно написание кода в стиле последовательности команд (императивов).

Декларативное программирование – это парадигма программирования, в которой разработчик описывает желаемый результат, а не процесс его получения.

Псевдокод – неформальное текстовое описание алгоритма.

Машинный код – инструкции, которые могут быть исполнены процессором напрямую. Как правило оформленные в двоичный код.

Что такое “парадигма программирования”

Парадигма

Само слово парадигма с английского переводится как “пример”, “паттерн”, “образец”. В английский оно пришло из греческого в XV веке, от глагола “показывать”.

В философии науки данный термин стал особо популярен в середине XX века, когда философ Томас Кун опубликовал книгу “Структура научных революций” в 1962

году. В ней он исследовал как устроен процесс научных открытий на больших исторических периодах. И в результате он представляет теорию о “сменах научных парадигм”, в которой описывает скачкообразный характер научного прогресса в целом.

Парадигма в Куновском смысле - это широко общепризнанный научный результат (или множество результатов), который на какое-то время описывает модельные задачи, решения которых будут исследоваться в данный исторический период, а также подходы и научные взгляды этого контекста. Проще говоря, парадигма - это множество идей, методов, подходов к решению или исследованию конкретных научных задач. То есть парадигма описывает не то, “что” мы делаем, а то “как” и “с помощью каких инструментов” мы можем подойти к решению задачи. Ещё можно сказать, что это “язык, которым мы рассуждаем при попытке выработать решение”.

Рассмотрим примеры применения различных “парадигм”. Уверен, что вы слышали об известнейшей библейской истории про Давида и Голиафа, повествующей о том как Давид (невысокого роста юноша без воинского опыта и доспехов) смог победить в поединке один на один гигантского воина Голиафа одетого в броню. Ветхий Завет повествует, о наличии у Давида пращи (то есть оружия дальнего боя), тогда как Голиаф имел в распоряжении только копье. Сразу после начала поединка, Давид достал камень и отправил Голиафу в лоб с помощью пращи и Голиаф сразу же был повержен. Если говорить терминами данного курса, то Давид использовал неожиданную для оппонента парадигму: парадигму дальнего боя. Данный пример потрясающе демонстрирует основную мысль данной лекции, которая заключается в том, что способов достичь какой-либо цели может существовать гораздо больше, чем один. Давид знал, что ему не одолеть Голиафа в традиционном ближнем бою, поэтому он воспользовался альтернативным оружием, с которым к тому же он уже умел обращаться. Сейчас спустя время, мы можем довольно уверенно сказать, что Давид использовал достаточно эффективную для его ситуации парадигму. Но в реальности это не всегда бывает так очевидно. И хотя программистам, конечно, как правило, ни с кем в смертельный поединок вступать не нужно, но применять различные парадигмы и подходы (а иногда и новые придумывать) - постоянно.

Рассмотрим ещё один пример. Предположим, в Интернете вы наткнулись на ОЧЕНЬ интересную IT-конференцию, которая пройдёт завтра в вашем городе. Чтобы туда попасть, нужно было купить пропуск, но их раскупили ещё месяц назад. Сейчас пропуска у вас нет, но вы всё равно очень хотите туда попасть. Какие у вас могут быть варианты, чтобы это сделать?

- В теории, можно попробовать поискать пропуск на конференцию у перекупщиков, которые продают пропуска с наценкой непосредственно перед началом мероприятия
- Можно попробовать найти людей, которые купили пропуск, но в последний момент решили не идти
- Ещё вы можете попробовать устроиться на это мероприятие волонтёром
- Также, возможно, у них есть онлайн-трансляция и чтобы послушать материал - покупка пропуска не обязательна
- Конечно, вы можете просто приехать туда и попробовать пройти без пропуска

Ну и в конце концов, можно вообще никуда не ехать, а просто дождаться когда будет следующая конференция, особенно если она проводится часто.
Все перечисленные варианты, можно назвать парадигмами. Конечно, не в научном смысле, а в бытовом, но всё же - каждый из них описывает набор инструментов, методов и идей, которые могут быть использованы для достижения конкретного результата. То же разделение есть и в программировании, но к счастью, в компьютерных науках, каждая парадигма гораздо более чётко определена.

Можно также вспомнить известную фразу “work smarter, not harder”, которую можно перевести как “работай разумнее, а не усерднее”. В данной фразе термин “более разумно” можно интерпретировать как “другим способом”. Поэтому, данная фраза, по сути, является призывом для каждого отдельного человека к поиску наиболее эффективной или подходящей для него парадигмы рабочего процесса.

Парадигмы программирования

Похоже, что мы готовы перейти к парадигмам программирования. Вспомним, что программирование по определению - это процесс написание программного кода, т.е. создания программ. Для написания программного кода как правило мы используем языки программирования. И как мы совсем скоро узнаем, каждый язык связан с одной или несколькими парадигмами программирования.

Теперь попробуем самостоятельно соединить два термина (“парадигма” и “программирование”) воедино и получим примерно следующее определение:
Парадигмы программирования - это множества актуальных достижений и идей в области программирования, в рамках которых ведётся практическая и исследовательская деятельность, то есть собственно разработка программ.
Если немного упростить “актуальные достижения и идеи в области программирования”, то можно получить более наглядное определение:

Парадигма программирования — это набор правил, концепций, идей и методов, определяющих способ написания программ. Таким образом, как и в случае

с научной парадигмой, парадигма программирования - это некий **стиль** написания программ.

🔥 Важно отметить, что парадигма - это множество именно правил и идей, а не их реализация. Реализацией этих правил и идей являются языки программирования. То есть с помощью парадигмы можно спроектировать решение, но не реализовать его окончательно.

Есть ещё немного другое определение:

Парадигмы программирования - это способ классифицировать языки программирования по их свойствам. Языки при этом могут быть классифицированы как в одну, так и в несколько парадигм. Из этого определения следует, что каждый язык программирования поддерживает стиль одной или более парадигм. И это правда. Например язык Python является **мультипарадигменным** языком и поддерживает как объектно-ориентированное программирование, так и процедурное, и ещё несколько прочих. SQL - является частью декларативной парадигмы. А язык Prolog принадлежит исключительно логической парадигме, и, таким образом, кстати является крайне редким представителем так называемых “**чистых**” языков программирования, то есть таких, которые реализуют идеи одной парадигмы.

В свою очередь, **мультипарадигменность** - это очень удобное свойство для языков, на которых (как например на Python) пишут разные типы программ. На этом языке вы можете написать и плеер для видео-хостинга, и построить сложную прогностическую модель основанную на данных (с которыми вы здесь же и поработаете), и порисовать в 3D или 2D, и игру разработать, или всё что захотите. Разработчики в процессе использования таких языков имеют возможность переключаться с одной парадигмы на другую по ситуации, не меняя при этом язык программирования. Поэтому, одна из задач этого курса продемонстрировать слушателям процесс подбора наиболее подходящей парадигмы под конкретную задачу.

💡 Вообще, термин “парадигма программирования” появился в 1978 году. Роберт Флойд, профессор Стэнфордского университета, выпустил статью-лекцию с названием: “The Paradigms of Programming”, в которой он, основываясь на парадигмах Томаса Куна, описывает их именно в контексте компьютерных наук. Лекция очень интересная и короткая, и если вы интересуетесь историей программирования или научного прогресса в целом - очень рекомендую вам её прочитать. В ней он предъявляет парадигмы программирования и объясняет как нехватка парадигм является основной причиной, так называемой депрессии разработки (“software depression”) того времени и как создание новых парадигм поможет этой индустрии с ней справиться (что в итоге и произошло).

Перейдём к тому чем такое разделение вообще может быть полезно обычному разработчику.

Зачем нужна классификация парадигм

Как мы уже знаем, парадигма программирования - это множество правил и концепций и методов, которые мы используем в процессе разработки и актуальные на сегодняшний день. А на наглядных примерах, мы увидели, что задачи в целом бывают очень разными и могут иметь больше одного подхода для генерации “хорошего” решения. И в каждом конкретном случае каждый подход будет иметь свои достоинства и недостатки. Поэтому, разработчик который имеет в репертуаре более одной парадигмы имеет больше шансов выполнить свою задачу быстрее, эффективнее и элегантнее. Сама же классификация просто помогает ориентироваться в парадигмах, для того, чтобы вам было из чего выбирать при решении задач.

Поскольку парадигмы программирования в одном из смыслов - это такая верхнеуровневая классификация языков программирования, то зная парадигму можно ещё и легко выбрать сам язык, на котором вы будете писать. Также удобно планировать своё развитие по расширению арсенала языков, действуя по рекомендации Роберта Флойда - расширяя количество известных вам парадигм. То есть если вы уже например знаете Java, Python и C++, то для развития вас как профессионала возможно имеет больше смысла либо попробовать новые парадигмы внутри этих же языков, либо такие языки как SQL, Haskell, UML, а меньше же смысла будет иметь в этом случае изучение C# или Objective C (если вы конечно не учите в них новую для себя парадигму). Так что общая рекомендация по развитию от Роберта - пробовать разные парадигмы.

С другой стороны, оставаясь в рамках одной парадигмы, изучение новых языков будет намного проще, чем при выход за её пределы. Например, после Си гораздо проще изучать императивную часть синтаксиса C++ или Jav-ы, чем пробовать SQL, Haskell или ООП синтаксис в том же C++.

Разумеется также очень важно учитывать и прикладную область, в которой вы собираетесь применять тот или иной язык. Безусловно, между прикладной областью и используемыми в ней парадигмами есть сильная корреляция. Например, в работе с данными или большими вычислениями часто применяются декларативные подходы. В веб-разработке, мобильной разработке и например в разработке игр - совершенно никуда без ООП, так как там всё держится на объектах и их атрибутах. В автоматизации рутинных операций или разработке API - процедурное и структурное программирование. Если пишете что-то более сложное или малоизученное, например, проект по робототехнике или искусственному интеллекту - скорее всего в процессе вы будете использовать все парадигмы популярные на данный момент одновременно: например, ООП для симуляций, процедурную и структурную для основной логики и функциональную для общения с базой данных.

💡 Использование сразу нескольких парадигм одним человеком - это вполне нормальное явление и происходит из-за того, что сложный продукт как правило состоит из разных частей, которые разработаны с использованием разных инструментов. Поэтому, если в вашей рабочей команде не очень много человек, скажем, меньше десяти, то хорошо разделить задачи по инструментам - скорее всего не получится. Это зачастую свойственно технологическим стартапам, особенно в начале пути. Поэтому в такой ситуации, будьте готовы думать на разных парадигмах 😀.

Надеюсь, к этому моменту я смог вас убедить, что в данном курсе речь пойдёт о чем-то что может оказаться крайне полезным для вас. Поэтому, перейдём к непосредственно самим парадигмам. Начнём мы с основного деления парадигм на императивное и декларативное программирование.

Деление на императивное и декларативное программирование

Императивное программирование

Что такое императивное программирование? Всё очень просто и интуитивно ожидаемо: по-английски “imperative” переводится как “повелительное наклонение”. А императивное программирование - это парадигма программирования, которой свойственно написание кода в стиле последовательности команд (императивов). Основной акцент тут делается на том, что в программе описывается именно “как” делать и в какой последовательности, а сама программа не знает что должно в итоге получится, так как она безукоризненно следует вашим инструкциям.

Например, давайте напишем программу для прохождения этого курса в императивном стиле на русском языке. Она бы, скорее всего, выглядела следующим образом:

1. Зайди на платформу Geek Brains
2. Для каждого урока:
    a. Посмотри лекцию
    b. Поучаствуй в семинаре
    c. Сделай домашнее задание
    d. Подведи итоги урока
3. Сдай финальный проект (или экзамен)
4. Получи сертификат о прохождения курса

Скорее всего, если вы недавно начали изучать тему программирования и IT в целом, то вы уже знакомы именно с этой парадигмой, т.к. именно на ней в основе своей держатся такие языки как Python, Java, C, C#, C++, Rust и даже R.

Декларативное программирование

С противоположного края - у нас декларативная парадигма. Declare с английского переводится как “объявлять”, a declarative как “повествовательный” или, если буквально, - “объявительный”. Опять же, название парадигмы довольно точно отражает её суть. Декларативное программирование - это парадигма программирования, в которой разработчик **описывает желаемый результат**, а не **процесс его получения**. Рассмотрим такую программу на нашем примере прохождения этого курса, написав программу в декларативном стиле:

1. Пройди курс “Парадигмы языков программирования и языки парадигм”
2. End

Потрясающе, правда? Очень удобно и самое главное, что вам совсем не обязательно думать о том сколько там уроков, нужно ли проходиться по ним циклом, и как они устроены внутри. Точнее, автору данной программы не нужно об этом думать, а вам как раз придётся, чтобы пройти данный курс 😀. То есть акцент в декларативном программировании делается на “объявлении” желаемого результата. Важно заметить, что естественно программист пишущий в декларативной парадигме не просто заявляет о своём намерении получить результат, он ещё и знает **какой результат** он попросить **может** и **как именно** его попросить. В этом и заключается искусство программирования на декларативной парадигме. Поэтому, если вы точно знаете, что при написании программы у вас есть возможность попросить результат типа “Пройди курс”, то программа, которую мы написали выше - вполне работоспособна. Но мы ещё углубимся в это в следующих лекциях. А пока что предлагаю вернуться к примерам.

Пример программы в императивном и декларативном стиле

Давайте рассмотрим ещё один простой пример, чтобы закрепить то что мы узнали. Предположим, наша задача - пройтись по массиву **arr** и получить сумму всех его элементов. Напишем алгоритм естественным языком. Тогда, императивная программа выглядела бы следующим образом:

1. Объявим переменную “**summ**” равной нулю
2. Перебираем элементы из массива **arr**:
    a. Для каждого элемента из массива прибавляем этот элемент к переменной “summ”
3. Возвращаем переменную “summ”

В данном примере мы снова подробно описали что именно нужно делать.

💡 Кстати, такое человекочитаемое описание алгоритма называется **псевдокод**. Само собой, программа такие команды скорее всего не поймёт.

Давайте теперь рассмотрим декларативную программу:

1. Используя функцию “сумма массива” - вычислить сумму элементов массива для массива arr
2. Profit!

Магия, не иначе! Уверен, что сейчас вам очень нравится данный подход. И это не просто так. Дело в том, что нам декларативный подход по форме ближе к нашему мыслительному процессу. Но заметьте, я заранее знал, что существует функция, называемая “сумма массива”, и что чтобы её применить, я должен на вход ей подать массив.

💡 Имя функции, её аргументы и тип возвращаемой переменной в программировании называется **сигнатурой функции**.

То есть я попросил результат, но я уже заранее знал какую функцию использовать для того, чтобы этот результат получить и какая у неё сигнатура.

Рассуждение
Дальше у опытного слушателя почти наверняка возникает вопрос: хорошо, а если я функцию “сумма массива” написал самостоятельно до этого, а потом уже в рамках нашей программы просто вызвал её из своего внешнего модуля - моя программа перестанет быть от этого императивной или нет? Ответом на данный вопрос будет “скорее нет, она останется императивной”. Строго говоря, если вы реализовали функцию “сумма массива” самостоятельно, то вы в процессе её написания вероятне всего использовали **императивную парадигму** (а иначе, если она уже реализована, то зачем бы вы это делали?). Тогда ваша внешняя программа, вызывающая эту, можно сказать, “готовую” функцию, как бы написана в **декларативной форме**. То есть получается, что вся программа целиком как бы и декларативная и императивная одновременно. Но на самом деле, декларативной программу делает та часть кода, к пошаговому исполнению которого у вас нету доступа, то есть она происходит под капотом у языка программирования. В данном же примере, вы можете залезть под этот “капот” и изменить принцип работы вашей функции, поэтому программа будет **императивной**, просто с использованием вашей самописной процедуры.

И наоборот: если вы используете функцию “суммы” встроенную в сам язык, то тогда ваша программа будет декларативной.

Очень важно добавить, что парадигма (или парадигмы) именно языка программирования НЕ изменилась! Язык остаётся прежним, несмотря на то, что при решении задач, вы можете смешивать различные парадигмы в одной программе. А вообще в современных языках программирования высокого уровня (таких как например, Python, Java, JavaScript, C++) у разработчика как правило есть возможность писать в **смешанном стиле**, выбирая в каждом конкретном случае то, что он сочтёт наиболее подходящим. Сейчас возможно это кажется непонятным, но это нормально и далее мы разберём как раз такой пример с возможностью использования разных парадигм в рамках одного и того же языка.

И наконец, мы приходим к важному вопросу: насколько императивность и декларативность является четким разделением? Насколько детально должен быть

описан алгоритм, чтобы считаться **императивным**? Или наоборот, насколько верхнеуровеный конечный результат нужно запросить, чтобы программа считалась **декларативной**?

Чтобы ответить на этот вопрос, давайте рассмотрим нашу программу по нахождению суммы в императивном стиле. Мы вроде бы как всё довольно *подробно* описали, но всё равно через *удобный* для нас верхнеуровневый интерфейс. Что я имею ввиду: даже если предположить что вместо псевдокода мы написали это на каком-то реальном языке программирования, при исполнении программы в компьютере всё равно в конечном счёте исполняются в виде **машинного кода**, то есть длинных последовательностей из нулей и единиц. Именно такие инструкции может прочитать и исполнить обычный процессор. При этом, превращением наших с вами трудов в машинный код обычно занимается компилятор, который скорее всего был установлен в комплекте с самим языком программирования и умеет читать именно этот язык. Когда вы запускаете программу, в результате в процессор на исполнение передаётся именно бинарный файл содержащий машинный код, то есть последовательность из нулей и единиц. Процесс формирования бинарника может очень сильно отличаться от языка к языку, т.к. у каждого языка свой компилятор (а иногда - интерпретатор), но это сейчас не главное.

Главное тут то, что при использовании какого-либо из **современных** языков программирования, ваша программа неизбежно столкнётся с процессом превращения(трансляции) в **машинный код**. Выходит, что относительно самого низкого уровня (машинного кода), любой язык программирования является как бы более декларативным, т.к. относительно “нулей и единиц”, результат - это как раз и есть какая-нибудь команда, например “прибавь 2 к X и сохрани результат в Y”. Поэтому, в данной классификации нам придётся опираться на **конвенции**. А конвенция в данном случае заключается в том, что называть “x+2” декларативным кодом относительно машинного - “не принято”. Но “принято”, в общих чертах, следующее: если вы описываете математические операции или пишете команды на уровне конкретных переменных (то есть имеете доступ к памяти) - вы скорее всего пишете **императивную** программу, а если оперируете готовым функционалом написанным до вас, и его исполнение происходит где-то в “черном ящике”, то скорее всего, **декларативную**. Иными словами: чем ближе то, что вы пишете при разработке к конечному результату, который вы хотите получить - тем вероятнее, что стиль вашего языка является **декларативным**.

Теперь я предлагаю перейти к более содержательным примерам написанным на синтаксисе реальных языков программирования.

Пример двух парадигм на Python

Давайте начнём с нашего примера суммирования всех элементов массива на языке Python. Сперва разберём код в императивном стиле

```
1   arr = [1,2,3,4,5,6,7,8,9,10] # объявляем массив arr
2   summ = 0                     # объявляем переменную summ равно нулю
3   for el in arr:               # перебираем элементы массива arr
4       summ += el               # на каждом шаге увеличиваем summ на arr
5   print(summ)                  # выводим summ на экран
```

В первой строчке мы объявляем массив **arr** равным каким-то числам. Во второй, объявляем переменную **summ** равной нулю, в ней мы в будущем будем хранить сумму всех элементов. В третьей строчке объявляется цикл, проходящийся по массиву **arr**. И очередной элемент на каждом шаге называется **el**. В четвёртой строчке мы увеличиваем нашу переменную **summ** на число **el**. И в пятой, наконец, выводим результат на экран. Именно цикл в третьей строчке и накопление суммы пошагово делает нашу программу *императивной*. Теперь давайте напишем *декларативную* программу:

```
1   arr = [1,2,3,4,5,6,7,8,9,10] # объявляем массив arr
2   sum(arr)                     # получаем значение функции для аргумента arr
```

Как я уже говорил, Python - язык высокого уровня, поэтому мы можем ожидать от него, что большое количество работы умные люди уже сделали за нас. И в случае с суммой массива - это действительно так. В Python есть готовая функция sum(), которая на вход принимает итерируемый объект (то есть такой, по которому можно пройтись циклом), а возвращает сумму всех его элементов.

💡 Итерируемый объект в Python (iterable) - это объект, элементы которого возможно перебрать по одному с помощью цикла. То есть тот, по которому можно итерироваться. Типичные примеры iterable: list (список), set (множество), tuple (кортеж), dict (словарь).

Результат исполнения двух приведенных выше скриптов - совершенно одинаковый (и равен 55). Но я конечно рекомендую не верить мне на слово, а установить Python на свой компьютер и убедиться в этом самостоятельно 🙂. Единственное ощутимое отличие между двумя парадигмами в данных реализациях, с которым вы можете столкнуться, - это время исполнения программы: второй скрипт отработает примерно в 2.5 раза быстрее (даже если не выводить результат на экран). И конечно же время разработки второго кусочка будет заметно меньше.

💡 Кстати, если у вас уже есть Питон, и вы исполняете скрипты с помощью ячеек (например - в формате ipynb), то возможно вам упростит жизнь

magic-функция **timeit**, которая автоматически измеряет время исполнения одной ячейки. Для того, чтобы вызвать timeit, необходимо написать в первой строке ячейки **%%timeit** (слитно без пробелов, маленькими буквами) и запустить данную ячейку.

Пример с reduce в Python

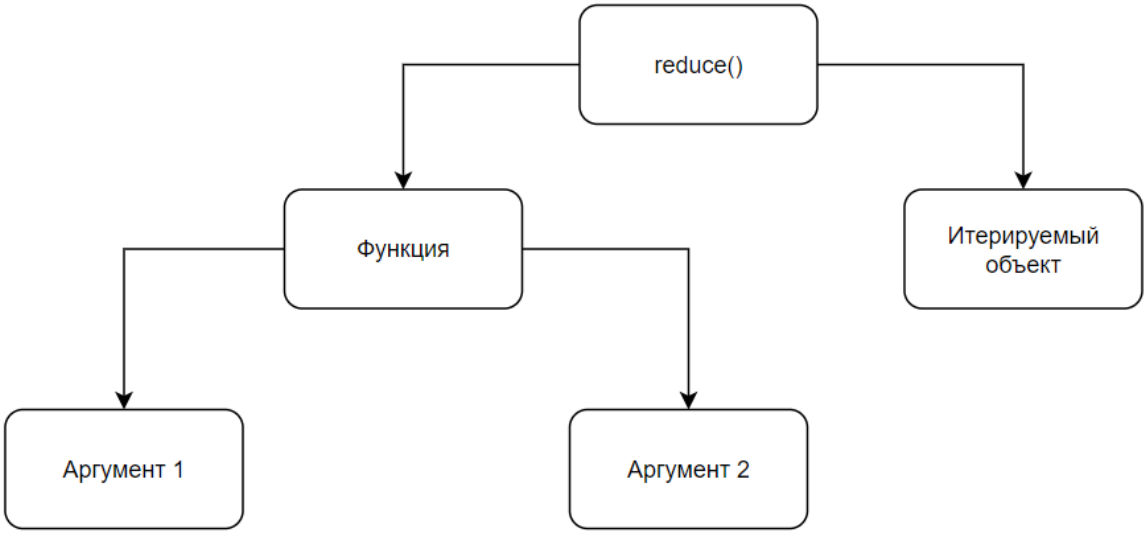
А что если бы готовой функции sum в Питоне не было, а мы всё равно хотели бы написать программу в декларативном стиле (сейчас будет мини-лекция по встроенным методам Питона)? Тогда к примеру нам помог бы метод reduce:

```
1   from functools import reduce        # импортируем reduce
2
3   arr = [1,2,3,4,5,6,7,8,9,10]        # объявляем массив arr
4   res = reduce(lambda x, y: x+y, arr) # вычисляем reduce
5   print(res)
```

💡 Ключевое слово **lambda** - в Питоне обозначает “анонимную функцию” и в нашем случае читается следующим образом: функция lambda двух аргументов x и y возвращает значение равное x+y. Список arr, который следует за “x+y” в вызове метода reduce в данном случае не имеет отношения к самой лямбде, а является именно вторым аргументом в вызове метода reduce.

Вероятно, вы ещё не сталкивались с данным методом, поэтому давайте вкратце разберём как он работает. Начнём с описания его *сигнатуры*. Reduce в языке Python - это встроенный метод, который принимает на вход два аргумента: **функцию** (в нашем случае - это функция *суммы* двух элементов) и **итерируемый объект** (в нашем случае - список arr), а возвращает **число**. Всё. Это сигнатура метода reduce.

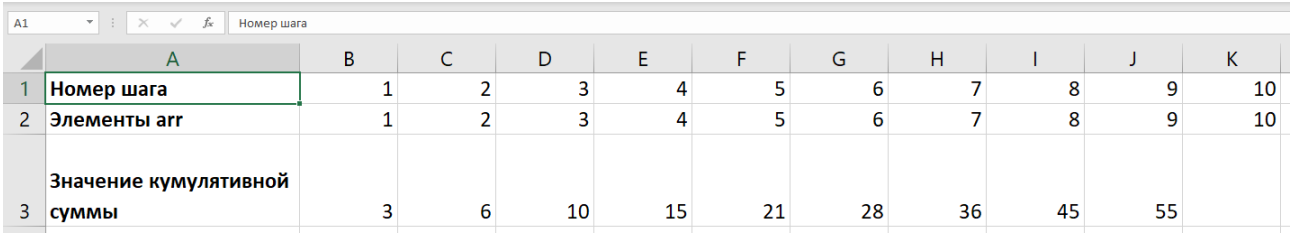
🔥 Здесь важно прояснить один тонкий момент: как я уже сказал, метод reduce принимает на вход два аргумента, первый из которых - это функция, принимающая на вход два аргумента. Слова “два аргумента” повторяются дважды и могут мешать интуитивному восприятию метода reduce. Но на самом деле ничего сложно тут нет. Есть метод reduce - у него есть сигнатура, в соответствии с которой он требует два аргумента, первый из которых - это функция. Причём эта функция тоже должна принимать два аргумента, иначе сам метод reduce - не сработает. Иллюстрация данной конструкции приведена ниже.



Теперь перейдём к тому, что происходит внутри метода. Возможно, вы уже догадались, что внутри будет присутствовать какой-то цикл, и это действительно так. Будем проходиться по итерируемому объекту, рассматривая по два элемента из списка на каждом шаге и начинём с первых двух. Для этих первых двух элементов вычислим значение с помощью переданной нами функции (в нашем случае - это сумма этих двух элементов) и сохраним это значение. На следующем шаге мы возьмём сохранённое значение с первого шага, и элемент следующий за вторым, после чего сложим эти два. В итоге получится, что мы на каждом шаге просто прибавляем к уже накопленной сумме следующее число, соответственно в итоге мы получим именно сумму всех чисел внутри списка.

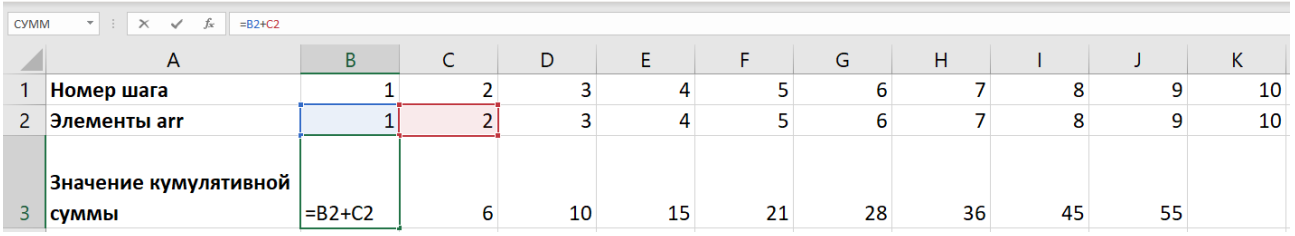
💡 Кстати, вычисленные значения суммы на каждом шаге называются **кумулятивной** суммой, то есть “накопленной” за определённое количество шагов. Это часто используется в статистике и анализе данных, например, при расчёте каких-нибудь показателей на временных рядах.
💡 Количество элементов, которое суммируется на каждом шаге - называется “размером окна”.

Для наглядности рассмотрим данный пример в Excel-е. Вот предположим у нас есть номера шагов в первой строке (всего их 10), на каждом шаге мы сдвигаемся на один элемент в право (элементы во второй строке), и для каждого шага у нас есть вычисленная накопленная сумма (третья строка). Теперь посмотрим как она считается.



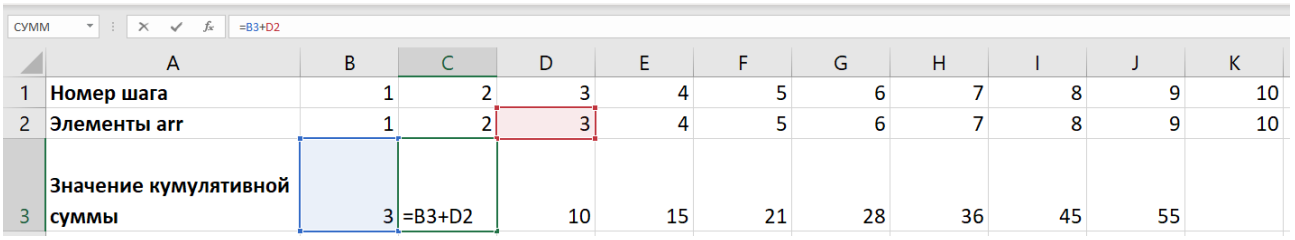
	A	B	C	D	E	F	G	H	I	J	K
1	**Номер шага**	1	2	3	4	5	6	7	8	9	10
2	**Элементы arr**	1	2	3	4	5	6	7	8	9	10
3	**Значение кумулятивной суммы**	3	6	10	15	21	28	36	45	55	

На первом шаге мы просто суммируем первые два элемента из списка. В нашем случае - “один” плюс “два”.



	A	B	C	D	E	F	G	H	I	J	K
1	**Номер шага**	1	2	3	4	5	6	7	8	9	10
2	**Элементы arr**	1	2	3	4	5	6	7	8	9	10
3	**Значение кумулятивной суммы**	=B2+C2	6	10	15	21	28	36	45	55	

А на следующих шагах мы суммируем ранее вычисленное значение со следующим элементом. В нашем случае, вычисленное значение - это “три”, а следующий элемент - это тоже “три”, соответственно в ячейке C3 будет стоять результат вычисления суммы для “три” и “три”, то есть - “шесть”.



	A	B	C	D	E	F	G	H	I	J	K
1	**Номер шага**	1	2	3	4	5	6	7	8	9	10
2	**Элементы arr**	1	2	3	4	5	6	7	8	9	10
3	**Значение кумулятивной суммы**	3	=B3+D2	10	15	21	28	36	45	55	

В результате, можно заключить, что если бы встроенного в Python метода sum не существовало, можно было бы использовать метод reduce, в который в свою очередь мы можем передать любую функцию двух аргументов какую захотим и в том числе реализовать простую “сумму элементов списка”. Такие методы существуют во многих языках программирования и по своей сути являются синтаксисом декларативной парадигмы.

Пример декларативного стиля на SQL

Предлагаю рассмотреть ещё один пример на языке запросов SQL. Если вы сталкиваетесь с базами данных и SQL впервые, то не переживайте, для вас в этом месте главное понять основную суть декларативной парадигмы. А так, в первый раз - мало кто этот кошмар понимает, к нему нужно просто привыкнуть. И на самом деле он является крайне эффективным и удобным языком.

Предположим, что имеется реляционная база данных содержащая таблицу users. Таблица users содержит следующие столбцы:

- age - возраст пользователя, тип integer
- registration_year - год регистрации пользователя, тип date

- is_subscribed - наличие подписки у пользователя, тип boolean

Предположим, нам поручили важную задачу: написать запрос, который бы посчитал количество пользователей старше 30 лет с имеющейся подпиской в разрезе года регистрации. То есть в итоге мы должны получить таблицу следующего вида:

registration_year	users_count
2021	x
2020	y
2019	z
..	

Для получения такой таблицы мы должны написать запрос на языке SQL и исполнить его на нашей базе данных. Если вы умеете писать запросы на SQL, то я рекомендую вам поставить на паузу и попробовать написать его самостоятельно.

Теперь давайте внимательно рассмотрим сам запрос, которым можно решить данную задачу. Что же тут происходит:

```
select registration_year, count(*) as users_count from users
where age > 30
  and is_subscribed = TRUE
group by registration_year
```

В первой строчке select говорит “я хочу таблицу, состоящую из двух столбцов: registration_year и users_count”. Далее from просит “собирать выборку из таблцы users”. После этого where говорит “отфильтруй только такие строки, в которых в поле age значения больше 30, а в поле is_subscribed значение равно TRUE. А в конце мы просим то, что получилось сгруппировать по значениям из поля registration_year с помощью команды group by. Причём count(*) в самом начале будет считать количество пользователей именно в каждом разрезе, то есть “количество пользоваталей для каждого значения registration_year”. Функция count здесь применяется именно к той таблице, которая получится после группировки, а не к исходной таблице users. Кстати, в этом случае говорят, что count - это агрегирующая функция для group by.

💡 Агрегирующии функции - это функции, которые получают на вход случайную величину (то есть вектор или массив) и возвращают число. Как

правило они используются при работе с данными и таблицами, когда нужно вычислить какие-нибудь характеристики. Например, среднее, сумму, минимум, максимум или количество записей.

💡 В данном примере агрегирующая функция применяется к результату group by, поэтому на вход ей будет подаваться не просто массив, а массив в каких-то разрезах (результат выполнения **группировки**) и для каждого из разрезов функция вернёт своё значение.

Но вернёмся к парадигмам. Вы наверняка заметили, особенно если раньше с запросами не сталкивались, что исполнение программы устроено слишком уж контринтуитивно: почему сначала мы говорим “выбрать”, потом говорим “откуда”, только потом “какие именно значения оставить в выборке”, а потом ещё и добавляем страшную группировку, которая в свою очередь использует агрегирующую функцию, которую мы написали в самом НАЧАЛЕ. Какая жесть! НО. Вы возможно даже не заметили, как написав эти страшные четыре строки вы избежали где-то десяти-пятнадцати не менее страшных, но в императивном стиле. Спросите себя, можете ли вы ответить на следующие вопросы:

- Как именно будет исполняться запрос?
- В какой последовательности будут перебираться записи из таблицы users?
- Будет ли прочитана вся таблица, а затем удалены все столбцы кроме registration_year или же всё кроме registration_year вообще не будет прочитано?
- Какие переменные будут создаваться по ходу исполнения запроса?
- Что происходило раньше:
    - группировка или фильтрация с помощью where?
    - группировка или создание столбца users_count?
- Как будут выглядеть утилизация оперативной памяти и ядер процессора на сервере базы данных в каждый момент времени во времени исполнения запроса?

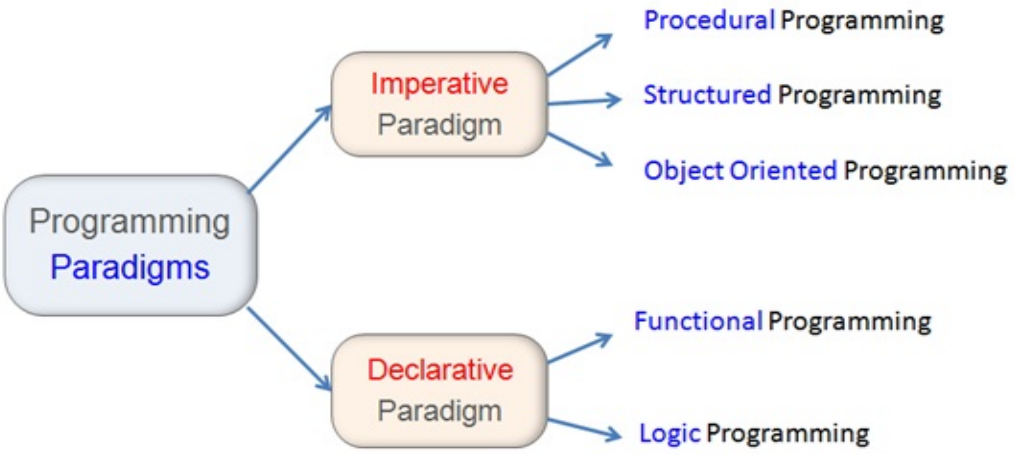
Если вы не архитектор баз данных, то скорее всего вы ответили нет, даже если знали как написать сам запрос. В данном случае на все эти вопросы за вас отвечала база данных, читая ваши SQL скрипты. SQL - это представитель декларативной парадигмы. В данном примере мы наглядно продемонстрировали главное отличие декларативной парадигмы, а именно: как можно “попросить результат” не зная при этом алгоритма для его достижения.

На самом деле, помимо этих двух парадигм, существует ещё множество других, менее популярных. Например, одной из групп парадигм наряду с декларативными и императивными является **метапрограммирование** - это создание таких программ,

которые на выходе создают другие программы. Но сейчас её мы изучать не будем, а вместо этого начнём погружаться глубже в декларативные и императивные парадигмы. Давайте рассмотрим из чего состоят данные группы парадигм.

Перечисление основных парадигм

Основные подтипы декларативного программирования - это логическое и функциональное программирование. Императивное программирование в свою очередь делится на процедурную, объектно-ориентированную (ООП) и структурную парадигмы. Конечно, существуют и другие, но в рамках данного курса, мы по большей части будем оставаться в этих основных пяти, так как на сегодняшний день они являются представителями наиболее популярных подходов и покрывают бОльшую часть широко используемых языков программирования.



Перейдём к первой парадигме - структурному программированию

Структурное программирование

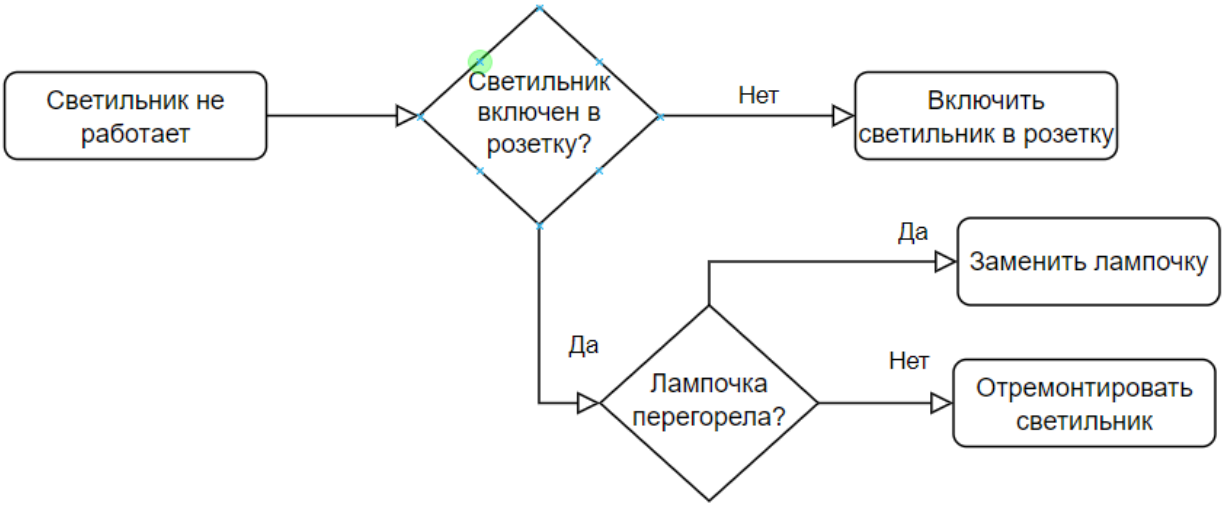
Структурное программирование - это тип императивного программирования, который основан на последовательном исполнении “блоков” или, правильнее сказать, на структурированном потоке управления (англ - “structured control flow”). Поток управления - это порядок выполнения инструкций программы. Вы наверняка уже знакомы с блок-схемами. С блок-схем как правило начинается изучение алгоритмов. Блок-схема - это графическая модель описания алгоритмов. А для нас - очень наглядный пример структурного программирования, потому что программа в структурном программировании - это последовательность исполняемых “блоков”. Блоки бывают трёх типов: **утверждение**, **ветвление** и **цикл**. **Утверждение** (или, ещё можно встретить “последовательность” - буквальный перевод с английского “sequence”) - это просто однократное выполнение следующей написанной

операции. **Ветвление** - это однократное выполнение любого количества операций внутри ветвления после проверки так называемого **условия**, при проверке которого возвращается булевое значение (True или False). И наконец, **цикл** - это многократное исполнение любого количества операций внутри цикла до тех пор, пока выполняется зарнее определённое **условие**. Основное отличие от базовой императивной парадигмы и других её современников - отсутствие оператора GOTO, который можно перевести как “исполни утверждение в заданной строке”. Но об этом мы более подробно ещё поговорим на лекции про структурное программирование

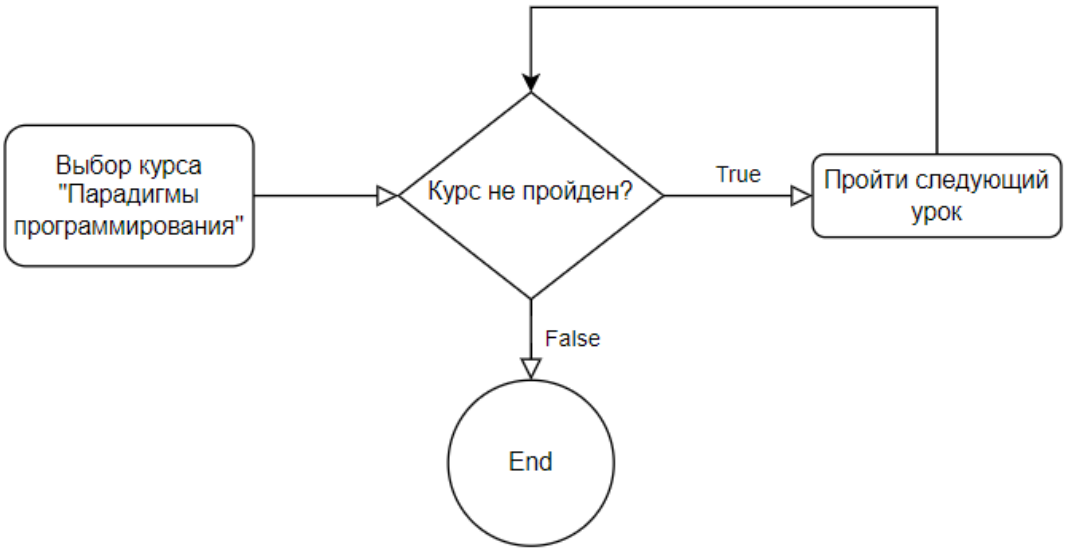
Перейдём к примерам структурного программирования

Примеры программ в структурной парадгиме

Разберём следующий пример.



Предположим, у вас не работает светильник. Это будет первым утверждением. Из которого мы приходим в первое ветвление, в котором проверяется условие “светильник включен в розетку”. Тут всё просто: если данное условие ложно, то мы переходим к утверждению “включить светильник в розетку”, а если истинно - к следующему условию. В следующем условии как мы видим проверяется “перегорела ли лампочка”, и если это истинно - меняем лампочку, а если ложно - несём светильник в ремонт. Как мы видим, в данном простом примере нет циклов, но есть последовательности и ветвления. Теперь разберём другой пример.



И снова у нас в примере прохождение курса по “Парадигмам программирования”. Какой замечательный пример! Мы уже писали нечто подобное, когда говорили об императивных парадигмах в целом. В данном примере присутствует цикл с условием. Как правило в программировании он назвается **while** и отрабатывает следующим образом: до тех пора пока переданное условие возвращает True, исполняется тело цикла. Соответственно, выход из цикла произойдёт в тот момет, когда условие вернёт False. В псевдокоде данный пример выглядит следующим образом:

```
выбрать курс
while курс пройден equals False do:
    пройти следующий урок
выход
```

Теперь давайте разберём более подробно ещё одну базовую парадигму: процедурное программирование.

Процедурное программирование

С этой парадигмы вероятнее всего начинался ваш путь в мир разработки. Это наверное самая популярная парадигма. Всё очень просто. Процедурное программирование - это вид императивного программирования, которое заключается в процессе написания последовательности команд и процедур. **Процедура** - это последовательность команд, которая исполняет команды **по очереди**, как обычная императивная программа. То есть вы своими руками пишите рецепт, а программа готовит суп (при этом, естественно, не зная что получит на выходе). Но при этом каждый шаг рецепта может быть составным. Таким образом,

большая часть кода как правило состоит именно из процедур и/или модулей (которые в свою очередь состоят из процедур). При этом уровень абстракции используемый в процедурной парадигме - минимален (конечно, относительно того языка, который вы используете). У процедур обычно есть доступ к **памяти** компьютера, на котором они исполняются, и следовательно там же существуют **переменные**, в которых можно что-то хранить, а потом изменять это что-то по мере исполнения программы.

Процедурное программирование включает в себя те же самые копцепции структурного программирования, такие как: ветвления и циклы. Здесь тоже все исполняется последовательно и чаще всего без GOTO. Но главное отличие от структурного заключается именно в акценте на **процедурах**, на которые может быть разделён ваш код. То есть ваш код в больше степени будет выглядеть как вызов конкретных процедур, чем сплошное полотно низкоуровневых манипуляций с переменными.

К процедурным языкам относятся, например, Си, Ассемблер и Фортран. На самом деле сам термин часто можно встретить в сининомичной форме с императивной парадигмой, но процедурное программирование всё-таки отличается от своих императивных соседей: объектно-ориенитрованной и структурной парадигм.

Давайте посмотрим как выглядит процедурный код.

Примеры процедурного кода

Пример с решением квадратных уравнений

Рассмотрим тривиальную задачу: поиск корней квадратного уравнения. Не волнуйтесь, решать их вручную нам точно не придётся. Вместо этого, мы напишем программу, которая будет вычислять их за нас. Естественно, это велосипед который уже много раз реализован, но нам в данном случае он очень подходит чтобы потренироваться. Перейдём к задаче. На вход подаётся три числа: a, b и c - которые являются соответствующими коэффициентами квадратного уравнения вида:

$$ax^2 + bx + c = 0$$

При этом, гарантируется, что a != 0, b != 0 c != 0. Необходимо написать программу, возвращая корни, если они есть или надпись “No real solution”, если корней в действительных числах - нет. Если вы уже начали программировать на любом языке поддерживающем процедурную парадигму (например Java, Python, C++, C# или что-то ещё), то я рекомендую вам прямо сейчас поставить эту лекцию на паузу и попробовать решить данную задачу самостоятельно. Если эта задача кажется вам

сложноватой или пока что что-то не понятно, то продолжайте смотреть, так как сейчас мы будем разбирать решение на примере языка Python –пауза 3 секунды–.

Поиск решения можно разложить на следующие шаги:

1. вычисление дискриминанта
2. проверка его знака
3. и наконец вычисление иксов (то есть - корней)

Формула дискриминанта: $(b^2 - 4ac)$. Формула корней: $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Напишем программу на языке Python для решения данной задачи. Основа любой процедурной программы - это функция “main” (главная функция). В ней описан самый верхний уровень того что происходит в вашем коде. И поскольку функция main использует всё что есть в вашем коде, то традиционно (и этот пример - не исключение) она размещается в самом конце вашего main скрипта. В нашем случае функция main содержит три строки: получение входных переменных, расчёт и сохранение solution-а (то есть “решения”) и вывод solution-а на экран.

💡 Здесь, я называю объявление переменных a, b и c именно “получением”, так как в реальной жизни вы, скорее всего, получали бы эти переменные откуда-то извне, например, с клавиатуры пользователя, при исполнении скрипта или из базы данных. А в данном случае, я захардкодил переменные прямо в скрипте для простоты, поэтому этот скрипт решает только одно конкретное квадратное уравнение, а не любое. А чтобы получить любое, нужно менять входные данные прямо в скрипте, что не очень безопасно для кода. В индустрии это называется “**хардкод**” и так делать без острейшей необходимости конечно же не стоит.

```
21  # %%
22  ### Example: procedural programming - roots of square equations
23  def calculate_x(a, b):
24      return -b / (2*a)
25
26  def calculate_x1x2(a, b, D):
27      x1 = (-b + D**0.5) / (2*a)
28      x2 = (-b - D**0.5) / (2*a)
29      return x1, x2
30
31  def calculate_D(a, b, c):
32      return b**2 - 4*a*c
33
34  def solve_for_x(a, b, c):
35      D = calculate_D(a, b, c) # расчёт дискриминанта
36      if D > 0:                # проверка знака дискриминанта
37          return calculate_x1x2(a, b, D) # вычисление корней
38      elif D == 0:             # проверка знака дискриминанта
39          return calculate_x(a, b) # вычисление корня
40      else:                    # проверка знака дискриминанта
41          return "No real solutions" # возвращаем "нет решений"
42
43  if __name__ == "__main__":
44      a, b, c = 6, -17, 12         # 6^2 - 17x + 12
45
46      solution = solve_for_x(a,b,c)
47
48      print(solution)
49
```

Продолжаем разбираться в решении квадратных уравнений. Процедура solve_for_x - это функция, которая принимает на вход три коэффициента a, b, c и возвращает решение (в каком-то виде). Она объявлена в 34 строке и вызывается из main-а в 46-ой. Теперь заглянем в тело этой функции. Первым делом вычисляется дискриминант D с помощью другой функции calculate_D, которая тоже принимает a, b и c и возвращает значение дискриминанта. Сама функция calculate_D - объявлена выше, в 31 строке. Далее в функции solve_for_x проверяется знак полуенного дискриминанта D: если он положительный, то вызывается функция calculate_x1x2, если равен нулю, то вызывается calculate_x, а в ином случае возвращается строка с текстом “No real solutions”. Данный пример является супер-наглядным для иллюстрации процедурной парадигмы. Какие у неё отличительные особенности:

1. Решение разбито на части, и эти части “запакованы” в процедуры
2. Части частей - тоже могут быть запакованы в процедуры

3. Промежуточные результаты мы храним в переменных, а значит используем память
4. Разработка программы производилась “сверху-вниз”, то есть с верхнеуровнего прототипа, где мы пошагово описали на какие этапы делится процесс получения решения, а затем спускаясь ниже и ниже, разработали вычисление конкретных дискриминанта и иксов

Данный случай отлично укладывается в процедурную парадигму именно потому, что получение решения очень хорошо разбивается на последовательны этапы. При этом, нам не нужно запоминать какие-то сложные отношения между переменными (как в ООП), а достаточно просто сохранять их значения и идти дальше в следующий этап. К тому же разработка “сверху-вниз” очень хорошо реализуется в данной задаче, так как сразу понятно что на низком уровне будут происходить какие-то конкретные расчёты, а на высоком результаты будут просто передаваться между этапами. Соответственно, удобно разработать сначала то как результаты вычислений будут передаваться между этапами, а затем уже и сами вычисления внутри.

Теперь предлагаю напоследок рассмотреть преимущества и недостатки процедурного программирования.

Преимущества и недостатки процедурного программирования

К основным преимуществам процедурной парадигмы можно отнести:

1) возможность повторного переиспользования уже написанного кода, так как он упакован в красивые процедуры, которые можно сохранить, а потом вызвать откуда угодно
2) универсальность, поскольку заранее почти никаких процедур верхнего уровня не определено, а следовательно и конвенции по их использованию не связывают руки программисту
3) широкая распространённость, означающая быструю взаимопомощь внутри сообщества программистов
4) возможность исполнения программ на различных типах процессора

К недостаткам относятся:

1) сложность самих получающихся программ, так как вы сами объясняете программе как получить желаемый результат условно “с нуля”
2) проблемы с данными: потеря, невоспроизводимость, поскольку во главе стола процедурного программирования стоят именно процедуры, которые не знают ничего про данные, кроме того, “что с ними необходимо сделать”
3) уязвимости в программе, возникающие в следствие отсутствия разделения переменных на классы, как в ООП. То есть как правило, все переменные

видны всем процедурам, и таким образом, при исполнении кода могут быть изменены те переменные, которые по задумке автора не должны были быть изменены. Конечно, в некоторых случаях можно использовать разделение на local и global (то есть локальную и глобальную видимость), но это не панацея. На одних локальных переменных программу написать может быть сложно, а глобальные как раз и порождают уязвимости, описанные выше.

На этом наша лекция подходит к концу. К структурному и процедурному программированию мы ещё вернёмся в следующих уроках. Теперь давайте подведём итоги лекции.

Итоги лекции

Сегодня мы начали изучать парадигмы программирования. Мы выяснили что такое парадигмы программирования, рассмотрев несколько определений, а также разобрали зачем они нужны. Погрузились в историю возникновения данного термина. Подробно изучили чем императивное программирование отличается от декларативного и разобрали конкретные примеры кода и псевдокода написанного в этих парадигмах. Начали разбирать структурную и процедурную парадигмы и разобрали примеры этих стилей.

Спасибо что прослушали эту лекцию и увидимся в следующей!

—

Домашнее задание

Позже

Текст домашнего задания и условия его сдачи (ссылка, файл и пр.) Обычно после лекций мы не даем домашнее задание - оно чаще всего будет дано студенту после семинара. Посоветуйся с методистом, должно ли тут быть домашнее задание.

Что можно почитать еще?

Позже

Используемая литература

1. Статья Роберта Флойда - “The Paradigms of Programming”
2. Книга Томаса Куна - “Структура научных революций”