

**Д.А. Ахметшин, Н.К. Нуриев,
С.Д. Старыгина, З.Х. Шакирова**

**ПРОЕКТИРОВАНИЕ ИНФОРМАЦИОННЫХ СИСТЕМ:
РАЗРАБОТКА ПРИЛОЖЕНИЙ НА ЯЗЫКЕ PYTHON**

подготовка IT инженеров в метрическом компетентностном формате

Казань

2016

УДК 004.45(075.8)
ББК 32.973-018.2я73
А 95

Рецензенты:

д-р педагог. наук, профессор, Э.Р. Хайруллина
канд. физ.-мат. наук, А.Н. Нуриев

Ахметшин Д.А.

Проектирование информационных систем: разработка приложений на языке Python: учебное пособие / Д.А. Ахметшин, Н.К. Нуриев, С.Д. Старыгина, З.Х. Шакирова. – Казань: Отечество, 2016. – 172 с.

Рассмотрена методология (спиральная модель) проектирования программного обеспечения информационных систем. Разработана концептуальная модель сложной системы. Приведен пример проектирования прототипа программного обеспечения дидактической системы на языке Python. С учетом того, что Python является удобным инструментальным средством для проектирования программного обеспечения информационных систем, работающих в Web сети, составлено руководство по программированию на этом языке.

Учебное пособие предназначено для магистрантов по направлению 09.03.02 «Информационные системы и технологии».

Работа выполнена при поддержке гранта РФФИ
(проект № 15-07-05761).

ISBN 978-5-9222-1130-7

©Д.А. Ахметшин, Н.К. Нуриев,
С.Д. Старыгина, З.Х. Шакирова

Оглавление

Предисловие	6
Основные понятия	8
Глава 1. Методология, методика, технология проектирования ПО.....	12
1.1. Методология – спиральная модель проектирования ПО ИС	13
1.2. Концептуальная модель ПО дидактической системы для подготовки IT-инженеров.....	17
1.3. Пример организации техногенной образовательной среды	27
1.4. Реализация программного продукта	35
Глава 2. Язык программирования Python	46
2.1. Python и его особенности	46
2.2. Синтаксические правила языка Python	47
2.3. Средства программирования на Python	48
2.4. Исключения в Python	49
Вопросы для повторения	52
Глава 3. Условные операторы и циклы	53
3.1. Условный оператор <i>if</i>	53
3.2. Циклы <i>for</i> и <i>while</i>	54
3.2.1. Цикл <i>for</i>	55
3.2.2. Цикл <i>while</i>	55
Вопросы для повторения	57
Глава 4. Типы данных.....	58
4.1. Числа	58
4.2. Структуры данных	60
4.2.1. неизменяемые последовательности – строки	60
4.2.2. неизменяемые последовательности – кортежи	64
4.2.3. изменяемые последовательности – списки	65
4.2.4. отображения – словари	68

4.3. Другие базовые типы	70
4.3.1. Множества	70
4.3.2. Файлы	72
Вопросы для повторения	76
Глава 5. Функции в Python	77
Вопросы для повторения	79
Глава 6. Объектно-ориентированное программирование	80
6.1. Создание класса	81
6.2. Методы в классе	81
6.3. Конструктор класса — метод <code>__init__</code>	82
6.4. Атрибуты	83
6.4.1. Доступ к атрибутам	83
6.4.2. Атрибуты класса	86
Вопросы для повторения	89
Глава 7. Веб-фреймворк Django	90
7.1. Веб-фреймворки	90
7.2. О Django	90
7.3. Архитектура Django	91
7.4. Краткое руководство по установке Django	94
7.4.1. Установка официальной версии с помощью <code>pip</code>	94
7.4.2. Установка официальной версии вручную	95
7.4.3. Взаимодействие Django с базами данных	95
7.5. Создание проекта на Django	96
7.5.1. Настройка базы данных	98
7.5.2. Основные настройки проекта	99
7.5.3. Запуск сервера для разработки	101
7.5.4. Создание приложения	102
7.5.5. Создание моделей	103
7.5.6. Создание представлений	120
7.5.7. Шаблоны	126

7.5.8. Подключение административного интерфейса.....	137
7.5.9. Формы	140
7.5.10. Встроенная система аутентификации пользователей	151
Список использованных источников	155
Приложение 1. Вопросы для тестирования	156

Предисловие

Все люди решают проблемы по одному и тому же универсальному алгоритму, состоящему из трех макроопераций:

1. Формализуют проблему, т.е. человек согласно цели преобразует решаемую проблему в известную для него задачу(и), используя при этом знания как ресурсы, а формализационные (А) способности как умения практически использовать абстрактно-логические способы моделирования в когнитивной сфере для этого преобразования. Следует учесть, что в рассматриваемом контексте задача является моделью проблемы с меньшей неопределенностью.

2. Конструируют (строит) план решения задачи. Разумеется, для этого необходимо иметь знания и владеть конструктивными (В) способностями, т.е. умениями построить алгоритм поиска решения задачи в контексте решения проблемы.

3. Реализуют план на практике, т.е. исполняет этот план в среде (социальной, экологической, виртуальной). Следует отметить, что для реализации плана требуются знания, как о предметной области решаемой проблемы, так и о среде в которую это решение внедряется. Очевидно, что для реализации плана необходимо владеть исполнительскими (С) способностями, чтобы внедренное решение не была отторгнуто средой.

В рассматриваемом случае, проблема может быть сформулирована так.

Требуется спроектировать новый программный продукт (ПО–программное обеспечение), обладающий определенными свойствами (перечисление множества свойств системы).

Примерно так бывает сформулирован заказ на программный продукт. На этом основании команда разработчиков и заказчик формируют техническое задание (ТЗ) с указанием сроков исполнения, финансирования проекта, штрафных санкций за срыв обязательств и т. д.

Методологию предметной области можно рассматривать как самое общее руководство к деятельности в этой области. В целом, это руководство предохраняет от многих возможных ошибок на стратегическом уровне. Не следует забывать, что ошибки, допущенные при проектировании новой системы на стратегическом уровне, тактическими приемами уже не исправить. В этом случае лучше проектировать систему заново. Результатом проектирования на методологическом уровне является разработанная концептуальная модель системы. Разработчики сложных систем на этом уровне проектирования должны обладать очень высоким уровнем развития формализационных (А) способностей и иметь глубокие знания в области проектируемой системы. На практике их называют аналитиками.

На тактическом уровне проектирования, т.е. когда уже разработана концептуальная модель системы и конкретизированы задачи разработчиков, выбираются методики организации деятельности по разрешению этих задач, а также конструируются алгоритмы поиска их решения. На этой фазе работ от разработчиков требуются высокоразвитые конструктивные (В) способности и глубокие знания в этой области.

На исполнительском (технологическом) уровне требуется на высоком уровне владеть инструментальными средствами и техниками исполнения разработанного на тактическом уровне плана.

На практике разработчиков с высоким уровнем развития исполнительских (С) способностей называют программистами.

В целом, очевидно, что каждый разработчик ПО сложной системы должен обладать высоким уровнем развития АВС способностей и усвоенными глубокими знаниями.

Основные понятия

Существует множество определений понятия система. В данном пособии будем придерживаться следующего определения. Система – это выделенный из окружающей среды объект, который взаимодействует с этой средой и при этом обладает следующими основными свойствами:

1. Имеет цель (назначение), для достижения которой он функционирует.
2. Состоит из связанных между собой частей (компонентов), образующих многоуровневую (иерархическую) структуру, которые выполняют определенные функции, направленные на достижение цели объекта.
3. Имеет управление, благодаря которому все компоненты функционируют согласованно и целенаправленно.
4. Имеет в своем составе исполняющий механизм, необходимый и достаточный для достижения цели, а также во внешней среде источники ресурсов для функционирования.
5. Обладает системными (неаддитивными) свойствами суммы свойств, т.е. сумма свойств системы не сводимыми к сумме свойств его компонентов.

В модели систему можно представить с разных точек зрения. Например, в SADT (Structured Analysis and Design Technique) функциональная модель системы представляется диаграммой (рис. 1).



Рис. 1 Функциональная модель системы

В этом представлении, система функционирует следующим образом: ВХОД (I) преобразуется в ВЫХОД (O) под УПРАВЛЕНИЕМ (C) с помощью МЕХАНИЗМА (M).

Комментарий.

SADT— одна из самых известных методологий анализа и проектирования систем, введенная в 1973 г. Россом (Ross). Эта методология успешно использовалась в военных, промышленных и коммерческих организациях для решения широкого спектра задач. На основе этой методологии были решены множество проблем, например, разработано программное обеспечение телефонных сетей, спроектированы системы обеспечения автоматизированной поддержки производственных процессов, решены задачи конфигурирования компьютерных систем, обучения персонала, управления финансами и материально-техническим снабжением. В рамках этой методологии был разработан стандарт IDEF0 как подмножества SADT, что обеспечило методологии автоматизированную поддержку, сделало ее более доступной и простой в употреблении.

Информационная система (ИС) рассматривается как частный случай, т.е. специальная система, предназначенная для хранения, поиска, обработки и защиты информации.

В широком смысле ИС содержит: данные, техническое и программное обеспечение (ПО), а также персонал.

В узком смысле ИС содержит: данные, программное и аппаратное обеспечение.

Проект рассматривается как уникальная модель системы, которую собираются (планируют) использовать для эффективного решения проблем.

В свою очередь, проектирование представляет собой уникальной процесс создания новой системы согласно проекту.

Очевидно, что проектирование ПО ИС является частью работ при проектировании процессов и систем.

Комментарий.

Создание информационной системы – это сложный процесс, который, как правило, требует значительной взаимной интеграции разработчика и последующего потребителя соответствующего решения. На практике редко бывает так, что данный продукт поставляется в коробочном формате – как антивирус или, например, офисный пакет. Разработка информационных систем, как правило, например, для бизнеса начинается с изучения специфики бизнеса компании-заказчика или задач, которые предстоит выполнять с использованием создаваемого программно-аппаратного комплекса. И только после этого IT-фирма начинает закладывать соответствующие алгоритмы – под конкретные критерии заказчика.

Разумеется, что у любого ПО ИС как и у другого продукта есть жизненный цикл (ЖЦ). В целом, ЖЦ можно определить как непрерывный процесс, который начинается с принятия решения о необходимости создания ПО и заканчивается при полном изъятии его из эксплуатации. При этом, ЖЦ ПО можно представить как двухэтапный процесс (рис. 2).

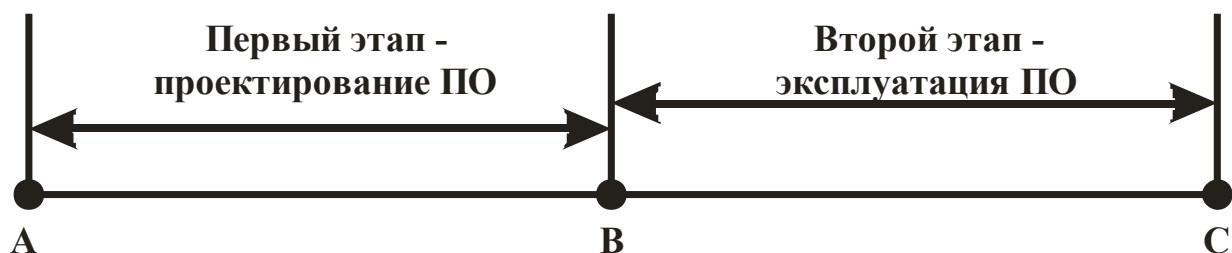


Рис. 2. Модель ЖЦ ПО как двухэтапный процесс

В простейшем случае ПО ИС представляет собой продукт, составленный из взаимосвязанных компонент: данные (база данных); интерфейс – как средство взаимодействия с ПО и приложение, которое позволяет реализовать предназначение системы в автоматическом режиме (рис. 3).

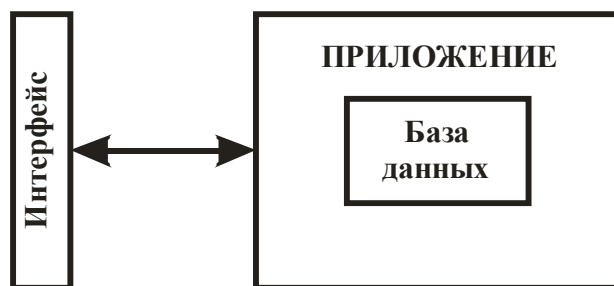


Рис. 3. Простейшая модель ПО ИС

Глава 1. Методология, методика, технология проектирования ПО

Методология это наука об организации деятельности, в какой – то предметной области. Методология, как и любая другая наука, имеет иерархическую структуру построения. На верхнем, т.е. стратегическом уровне иерархии определяются основные подходы и принципы в организации деятельности, которые в совокупности формируют концептуальную модель этой организации. В свою очередь, на тактическом уровне в рамках (в контексте) концептуальной модели рассматривают методики – процессы организации деятельности, т.е. детализованные с уточнением, модели организации деятельности в контексте концептуальной модели. Далее, на практическом уровне рассматриваются технологии, которые с использованием ресурсов и методик позволяют получить конкретный результат (продукт) деятельности (рис. 4).

Роль методологии в организации деятельности заключается в регламентации основ разработки сложных систем. Она описывает последовательность шагов, модели и подходы, тщательное следование которым приведет к хорошо работающим системам. Хотя методология, вообще говоря, не гарантируют качества построенных систем, тем не менее, они помогают справиться с проблемами размерности, и в конечном итоге оценить продвижение вперед. Более того, методология обеспечивают организационную поддержку, позволяющую большим коллективам разработчиков функционировать скоординированным образом.

Проектирование ПО ИС, в основном, может проходить в рамках двух методологий (двух моделей организации деятельности): спиральная и каскадная модели проектирования.

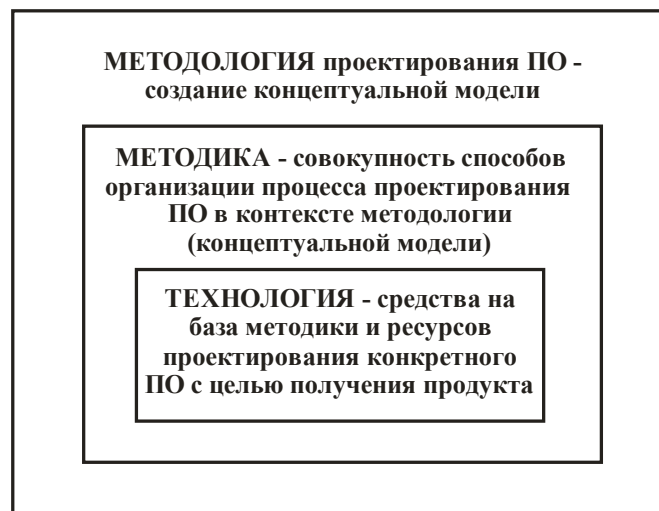


Рис. 4. Иерархическая модель проектирования ПО (методология, методика, технология)

1.1. Методология – спиральная модель проектирования ПО ИС

По методологии (спиральная модель) всю работу по проектированию (созданию нового ПО) можно разделить на четыре фазы, т.е. разделить во времени на четыре разнотипных работ. При этом работы в рамках каждой фазы имеют свои названия: 1. Исследование. 2. Уточнение. 3. Построение. 4. Развёртывание (рис. 5).

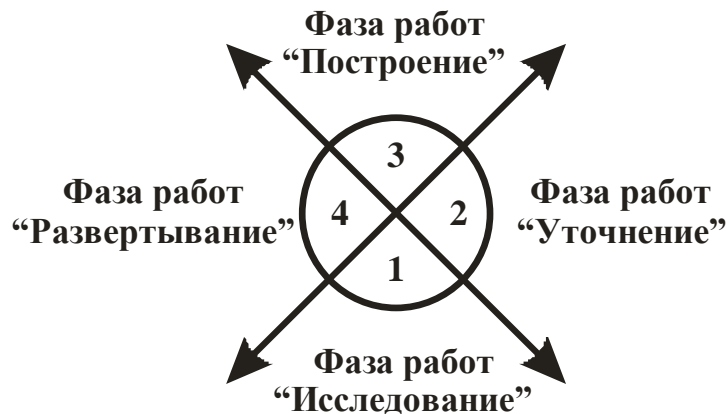


Рис. 5. Модель-представления работ при проектировании в пространстве и времени

Модель рис. 5, рассмотрим как некоторое координированное пространство, где можно зафиксировать начало работы по проектированию и проследить дальнейшее развитие процесса в рамках методологии «Спиральная модель».

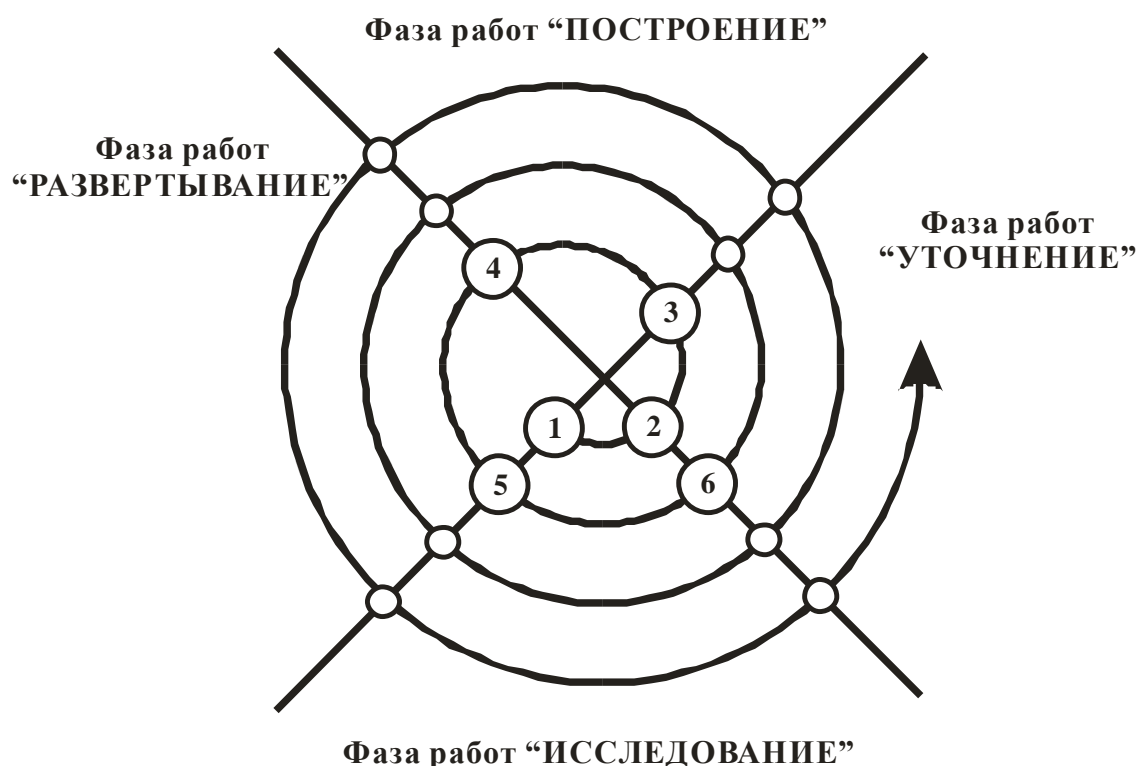


Рис. 6. Модель развития проекта по спирали

Распишем модель развития проекта подробнее. Точка 1 – начало процесса проектирования ПО, т.е. начало ЖЦ ПО и решения комплекса задач фазы работ «исследование» (рис. 6). На этой фазе работ необходимо решить следующие основные задачи:

- 1) исследовать рынок ПО ИС по теме заказа;
- 2) обсудить требования заказчика;
- 3) разработать концептуальную модель проекта.

К концу этой фазы работ, т.е. в точке 2 (см. рис. 6) должны быть получены следующие результаты:

- 1) построена концептуальная модель проектируемой системы, согласованная с заказчиком на основе его требований;
- 2) разработано техническое задание (ТЗ) к проекту.

Комментарий. При создании концептуальной модели проектируемой системы целесообразно использовать SADT, которая позволяет сократить ошибки, допускаемые на стратегическом уровне. Диаграммы SADT позволяют организовать взаимопонимание между пользователями и разработчиками, а также отладить переход от

исследования к проектированию. Диаграммы SADT легко отражаются такие характеристики как вход, управление, механизм, выход, обратная связь.

Точка 2 (рис. 6) – начало работ фазы «уточнение», т.е. процессов конкретизации и детализации концептуальной модели проекта, а также всех вопросов финансирования, сроков окончания, технического задания, документирования, юридического оформления и т.д. Следует подчеркнуть, что каждый проект уникален и имеет свои особенности. Основными результатами в точке 2 являются:

- 1) формализованный комплекс задач, который необходимо решить в контексте концептуальной модели;
- 2) юридически оформленное ТЗ;
- 3) решение финансовых, организационных, вопросов и отчетности.

Точка 3 (рис. 6) – начало работ из фазы построение. Эта фаза работ состоит из трех этапов:

1. Первый этап – эскизное проектирование. На этом этапе строятся блок-схемы решения формализованных задач, полученных в фазе работ «уточнение». Таким образом, основные результаты проектирования на этом этапе, это блок – схемы решения комплекса задач на эскизном уровне, а также множество диаграмм и рисунков.

2. Второй этап – программирование ПО. На этом этапе (на основе блок-схем проекта, полученные на предыдущем этапе) конструируются локальные модули ПО ИС.

3. Третий этап – сборка системы, т.е. ПО, как правило, сложный объект и состоит из множества модулей, которые проектируются разными исполнителями. На этом этапе все эти модули необходимо собрать в работающий единый программный комплекс. Как показывает опыт, проблема сборки является одной из самых сложных задач при проектировании ПО.

На рис. 7 на базе общей схемы (рис. 6) приводится все три этапа работ на фазе «построение».

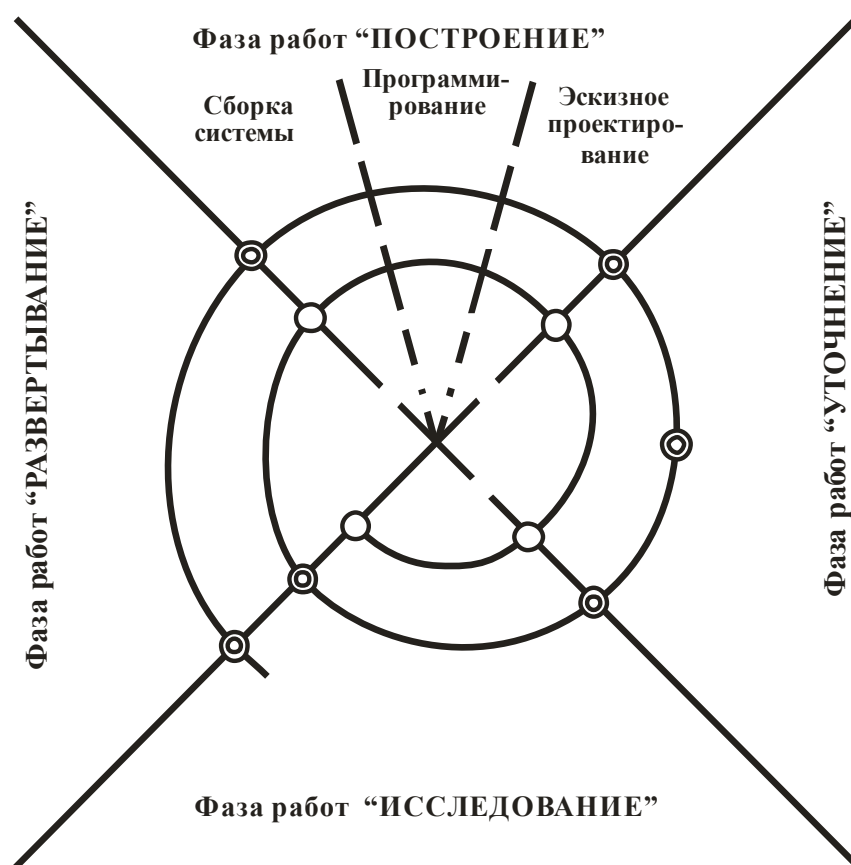


Рис. 7. Модель развития проекта (этапы фазы построение)

К концу фазы работ «построение», т.е. в точке 4 (см. рис. 6) должны быть получены следующие результаты:

1. ПО ИС, собранный из программных модулей, работающий в испытательном режиме.
2. Список замечаний по корректности работы.

Точка 4 (рис. 6) – начало работ из фазы развертывание (тестирование). На этой фазе происходит тестирование системы по методам «прозрачного (белого)» и «черного» ящиков. С помощью тестирования методом белого ящика устанавливается корректность, эффективность работы каждого модуля, согласованность их работы в едином комплексе. Тестированием по методу черного ящика устанавливается корректность преобразования входной информации в результат.

Точка 5 (рис. 6) – сформирован первый прототип ПО системы, как правило, с большим количеством недостатков, которые в последствие (в развитии первого прототипа системы) будут устранены в следующем прототипе ПО ИС.

1.2. Концептуальная модель ПО дидактической системы для подготовки ИТ-инженеров

При проектировании эффективных дидактических систем для подготовки ИТ-инженеров должны учитываться, по крайней мере, пять основных фактора: 1) ИТ-инженерия – это область с революционным темпом развития; 2) виртуальная среда бытия для ИТ-инженеров, не менее значимо, чем социальная среда, т.е. они там трудятся, созидают, отдыхают; 3) за время подготовки ИТ-инженеру необходимо быстро усвоить большие объемы знаний и выработать умения, навыки оперировать в техногенной среде; 4) находятся в ситуации постоянного риска потери компетентности, связанное с первым фактором; 5) основные результаты достижений ИТ – инженеров, как правило, невольно фиксируются в сети и доступны для оценки их качества.

В ходе исследования было установлена следующая фундаментальная закономерность, которую называли «решение проблем в три операции». Суть этой закономерности состоит в следующем. Любую проблему человек решает через свою деятельность в три связанных и следующих друг за другом операции. Первая операция (операция А) – формализация проблемы, т.е. человек в меру развития своих способностей (умений) а также знаний преобразует решаемую проблему в аналог известной для него задачи. Вторая операция (операция В) – конструирование плана решения задачи, т.е. человек на основе своих способностей и знаний формирует план решения этой задачи. Третья операция (операция С) – исполнение этих планов в реальной (для ИТ-инженеров виртуальной) среде.

1. Функциональная модель инженера. Опираясь на эту фундаментальную закономерность и используя методологию структурного системного анализа (SADT - *Structured Analysis and*

Design Technique), была построена функциональная модель инженера (рис. 8).

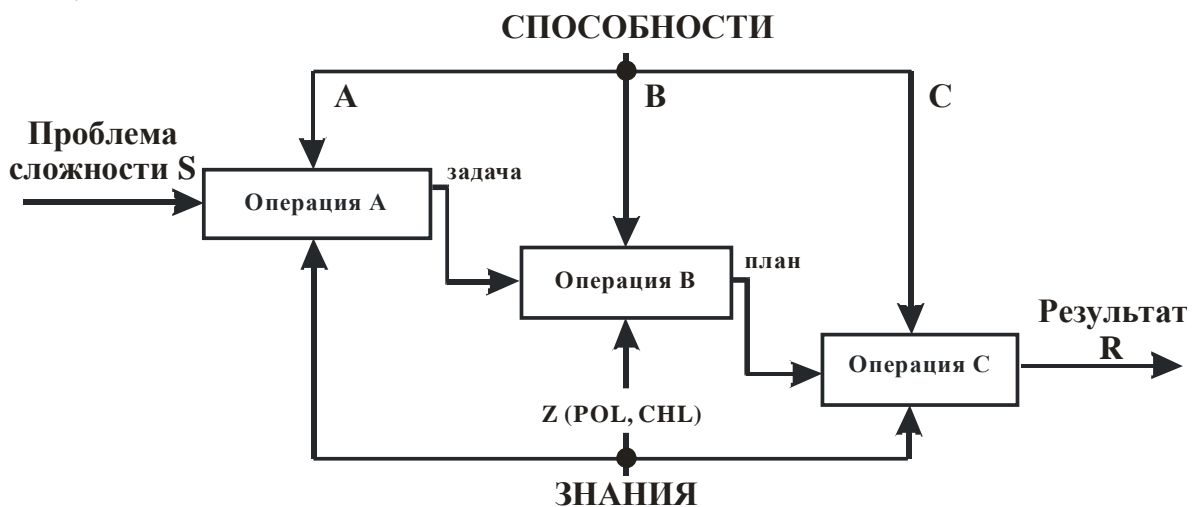


Рис. 8. Функциональная модель инженера

В модели приняты обозначения: через S – величина сложности проблемы; A, B, C – величины уровней развития формализационных, конструктивных, исполнительских способностей инженера в рамках какой-то компетенции; POL, CHL – величины полноты и целостности усвоенных знаний инженера в рамках компетенции; R – результат решения проблемы (изменяется от успешного до неудачного); Z – величина глубины усвоенных знаний, зависящая от двух параметров POL и CHL .

Модель функционирует следующим образом: проблема сложности S преобразуется в успешный результат R с вероятностью $P(\text{усп})$, в зависимости от уровня развития ABC -способностей инженера и его знаний глубиной Z . Поэтому, вероятность $P(\text{усп})$ трансформации проблемы сложности S в успешный результат R формально можно записать через функционал $F(*)$, т.е. $P(\text{усп}) = F(A, B, C, POL, CHL, S)$. Как следует из статистических данных, на качественном уровне можно утверждать, что чем выше уровень развития ABC - способностей инженера на фоне его знаний глубиной Z , тем выше вероятность успешного решения проблемы сложности S . Разумеется, при фиксированных значениях A, B, C, POL, CHL с увеличением сложности S проблемы эта вероятность $P(\text{усп})$ будет уменьшаться.

2. Построение квалитетрических шкал для оценки компетентности IT-инженера. Квалитетрические шкалы строятся: 1) основываясь на установленную фундаментальную закономерность «решение проблем в три операции»; 2) на основе выявленного комплекса параметров $\langle A, B, C, POL, CHL, S \rangle$, определяющего вероятность успешности при разрешении профессиональных проблем; 3) с учетом фактора, что реализованные проекты IT-инженера, как правило, доступны в Web-сети для экспертизы их сложности.

В целом, представляют интерес две шкалы: пятимерная шкала качества владения компетенцией (КВК(5)) и трехмерная шкала качества владения компетенцией (КВК(3)). Обе шкалы построены на пучке векторов (рис.9, рис. 10).

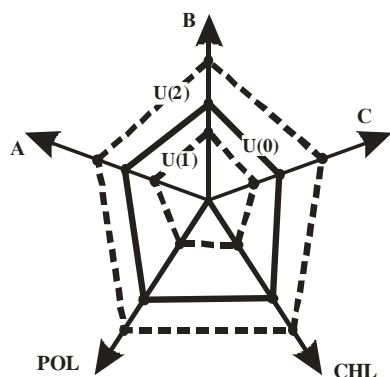


Рис. 9. Шкала КВК(5)

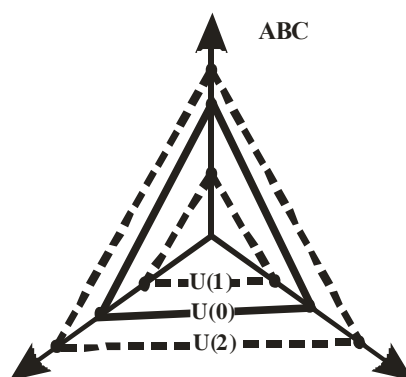


Рис. 10. Шкала КВК(3)

На этих шкалах профиль $U(0)$ определяется экспертом в рамках рассматриваемой компетенции. Инженеры, профиль которых «больше» (например, профиль $U(2)$) профиля $U(0)$ классифицируются как компетентные.

Необходимость этих двух квалитетрических шкал объясняется эвристикой, т.е. как показывает опыт, эксперты часто, исходя из качества выполненных проектов в Web-сети не могут оценить уровни развития А-формализационных, В-конструктивных, С-исполнительских способностей инженера по отдельности. Это обстоятельство приводит к необходимости использования второй шкалы (КВК(3)), в которой ABC-способности инженера оценены в комплексе.

3. Функциональная модель дидактической системы. На основе функциональной модели инженера и установленных характеризующих успешность инженера параметров A , B , C , POI , CHL , а также построенных квалитетических шкал $KVK(5)$, $KVK(3)$, была создана функциональная модель дидактической системы нового поколения с технологией подготовки студентов в метрическом компетентностном формате (МКФ) (рис. 11). Этот формат подготовки предполагает оценку (в числах) всех используемых переменных: 1) S – сложность проблемы измеряется в минутах/работы эксперта по решению этой проблемы; 2) уровень развития ABC -способностей студента оценивается в долях от производительности труда эксперта (работа/минутах); 3) POI и CHL измеряется в долях усвоенного теоретического материала, изложенного в рамках осваиваемого курса (подробнее об этом изложено в учебном пособии Печеный Е.А., Нуриев Н.К., Старыгина С.Д. Экономико-математические модели в управлении. – Казань: Центр инновационных технологий, 2016. – 224 с.)). В целом, система разворачивается в любой техногенной образовательной среде (ТОС).

Цель подготовки в МКФ – быстрое развития уровней ABC -способностей студента в рамках какой-то компетенции на фоне повышение значений показателей глубины усвоенных знаний. Средством достижения цели является дидактическая система нового типа (поколения), функционирующая в ТОС как ее часть.

Дидактическая система функционирует следующим образом: допустим, учебный курс состоит из n -разделов. Каждый раздел укомплектован, т.е. содержит теоретический, практический, диагностический материалы с оценкой их сложности. В рамках раздела осваивается теоретический, практический (решаются задачи) материалы. При этом в ходе учебной деятельности студент наращивает уровни развития ABC -способностей на ΔABC и углубляет знания на ΔZ .

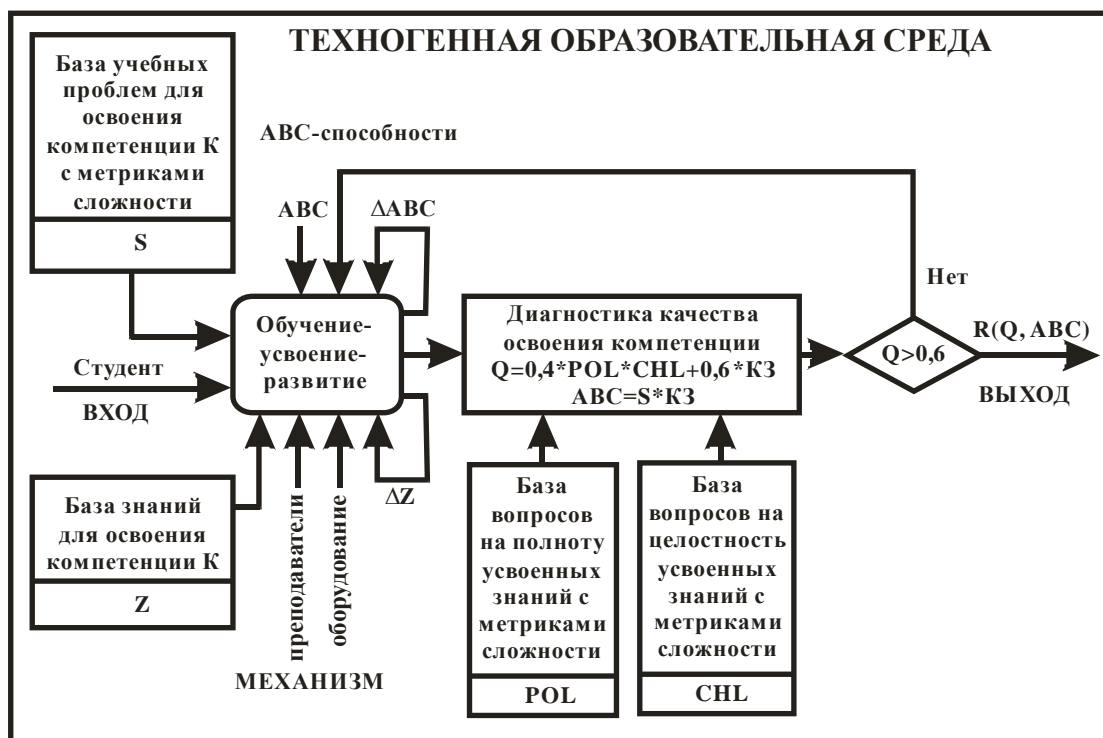


Рис. 11. Функциональная модель дидактической системы подготовки инженеров в МКФ

По окончании раздела студент проходит контроль, т.е. проходит тестирование на полноту и целостность усвоенных им знаний (метрические показатели POL, CHL принадлежат интервалу от 0 до 1). Преподаватель оценивает качество выполненного задания (метрический показатель $K3 = (0 \text{ до } 1)$). Автоматически оценивается качество освоения компетенции в рамках раздела (метрический показатель $Q = 0,4 * POL * CHL + 0,6 * K3$). Отдельно оцениваются уровни развития ABC – способностей с графической интерпретацией достижений (метрический показатель $ABC = S * K3$). Эвристическим путем полученный минимальный порог допустимого качества владения компетенцией в рамках раздела берется равным 60%. Если показатель качества развития превышает 0,6, то студент допускается к освоению следующего раздела, в противном случае заново осваивает этот же раздел.

4. Фундаментальные закономерности: принцип природосообразности, «зона ближайшего развития» «обучение на предельно - допустимой трудности». По своей сути, изначально инженерия базируется на ремесле, т.е. инженер в активном режиме

создает новые объекты и при этом взаимодействует с техникой со сложными информационными системами. Как было уже подчеркнуто, он должен уметь, иметь навыки и знать, как оперировать в техногенной среде в системе реального времени. Разумеется, это требует очень высокого уровня развития АВС-способностей, глубоких и больших объемов усвоенных знаний в инженерных компетенциях. Очевидно, достичь высокого уровня развития можно только через интенсивное высокоэффективное обучение. На практике, при проектировании таких высокоскоростных дидактических систем для подготовки IT-инженеров, педагоги сталкиваются с тремя фундаментальными закономерностями.

Первая фундаментальная закономерность (сформулировано в виде принципа), установленная Яном Коменским, которая утверждает, что обучение должно быть природосообразным, т.е. человек от природы должен быть приспособлен к деятельности к которому обучается.

Вторая фундаментальная закономерность (Л. С. Выготский [3]) гласит, что обучение только тогда хороша, когда проходит впереди развития (обучение через «зону ближайшего развития»). Третья фундаментальная закономерность (Л. В. Занков [4]) утверждает, что наиболее быстрое развитие происходит в режиме «обучения на предельно - допустимой трудности». В целом, эти три закономерности «ограничивают» возможную предельную скорость развития инженера. Поэтому, в дидактических системах на проектном уровне должно быть заложена возможность для каждого студента природосообразного развивать АВС - способности (из закономерности «решение проблем в три операции») через обучение на собственных предельных режимах.

Следует отметить, что при обучении, проблема доступности курса по сложности возникает перед каждым студентом, поэтому возникает задача синхронизации скорости развития АВС – способностей (на фоне усвоения знаний) с темпом подготовки.

Разумеется, предельные скорости развития ABC – способностей у каждого студента индивидуальны, также различаются уровни их развития на актуальный момент и глубина, усвоенных им знаний. Все это приводит к необходимости проектирования многоуровневых по сложности курсов (как это принято в сложных компьютерных играх).

К общесистемным требованиям относится то, что специально организованная техногенная среда вуза должна позволить автоматизировать процессы синхронизации, диагностики качества усвоенных знаний и умений через внедрение элементов искусственного интеллекта с использованием нейронных сетей и самообучающихся моделей.

Рассмотрим пример, допустим учебный курс определенной сложности, предназначенный для освоения какой-то компетенции К – состоит из 4 тем. Выделим три уровня сложности материала, сгруппированных в разделы. Разделы (1.1, 1.2, 1.3, 1.4) – первого уровня; разделы (2.1, 2.2, 2.3, 2.4); (3.1, 3.2., 3.3., 3.4) – соответственно второго и третьего уровней сложности (рис. 12).

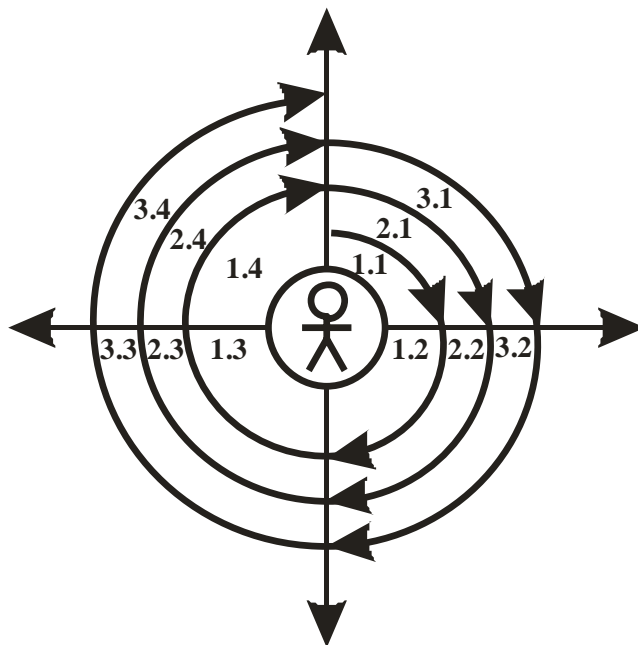


Рис. 12. Структура организации разделов в многоуровневом учебном материале

Процессы, т.е. усвоение знаний и развитие ABC-способностей студента происходит по спирали согласно последовательности:

разделы 1.1-1.4 – первый, 2.1-2.4 – второй, 3.1-3.4 – третий уровни сложности материала. Причем, разделы 1.1, 2.1., 3.1. составляют первую тему изучаемого материала; 1.2., 2.2., 3.2. – вторую и т.д. Сложность материалов разделов 1.1-1.4 должна соответствовать ЗБР студента.

Алгоритм организации индивидуальной работы студента FAM при подготовке в МКФ с учетом его ЗБР приводится на рис. 13.

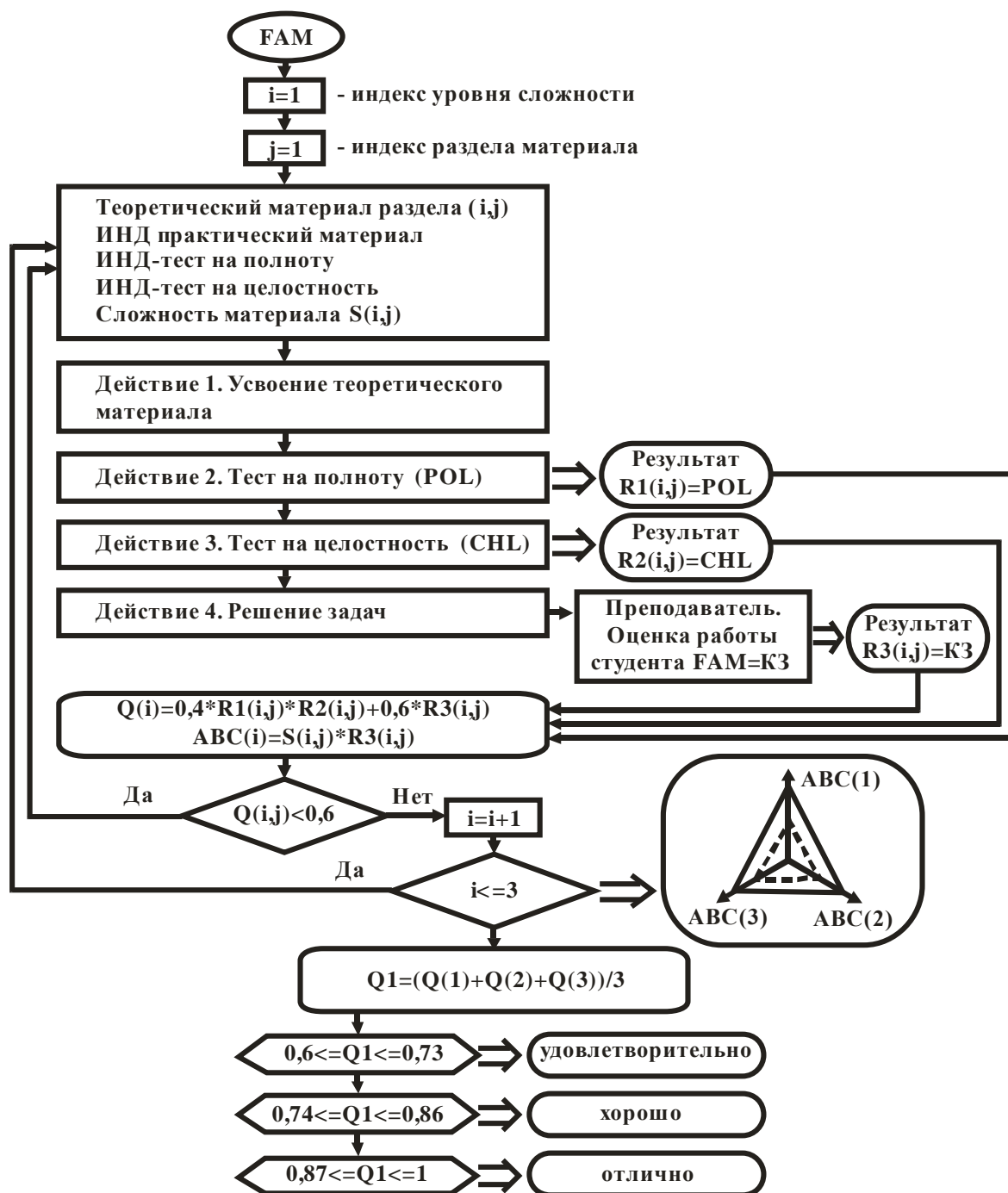


Рис. 13. Схема организации учебной работы IT-инженера в МКФ с учетом ЗБР

Очевидно, оптимальное количество уровней сложности, на которое разбивается традиционный курс (для того, чтобы студент на требуемом качественном уровне овладел компетенцией) зависит от множества факторов: сложность учебного курса, количество часов отпущенных по учебному плану, уровня развития АВС-способностей и глубины усвоенных знаний достигнутых на предыдущих курсах и т.д. Формализовать и оценить влияние этих основных факторов еще предстоит.

С точки зрения «сильных» и «слабых» студентов обучение в многоуровневой техногенной среде происходит по принципу «гонки с преследованием». «Сильные» студенты быстро проходят нижние уровни и замедляются только на верхних уровнях сложности материала, а слабые постепенно их догоняют. Необходимо отметить, что они догоняют успешно. Как показывает опыт, в многоуровневых курсах в среднем результат (качество освоения компетенции) оказывается на 16% выше, чем на одноуровневых курсах, организованных также в техногенной среде.

5. Оценка сложности учебных курсов. Рассмотрим две категории учебных курсов: 1) практико-ориентированные курсы; 2) теоретико-ориентированные курсы. Расчет сложности этих курсов будет происходить по разным методикам.

В практико-ориентированных курсах сложность овладения компетенцией будет оцениваться на основе сложности задач (учебных проблем) предлагаемых студентам для самостоятельной работы. При этом сложность курса должна оставаться в рамках требований, предусмотренных учебным планом.

Рассмотрим пример расчета сложности многоуровневого курса, который состоит из 4 тем и 12 разделов, в каждом разделе содержится 12 блоков заданий для самостоятельной работы.

На рис. 14. приведена структура организации блоков с заданиями и экспертными оценками их сложности.

Уровни сложности заданий	Курс (12 разделов, 12 блоков)			
	Тема 1	Тема 2	Тема 3	Тема 4
	Блок (1,1) Сложность SP(1,1)	Блок (1,2) Сложность SP(1,2)	Блок (1,3) Сложность SP(1,3)	Блок (1,4) Сложность SP(1,4)
	Блок (2,1) СложностьSP(2,1)	Блок (2,2) СложностьSP(2,2)	Блок (2,3) СложностьSP(2,3)	Блок (2,4) СложностьSP(2,4)
	Блок (3,1) Сложность SP(3,1)	Блок (3,2) Сложность SP(3,2)	Блок (3,3) Сложность SP(3,3)	Блок (3,4) Сложность SP(3,4)

Рис. 14. Структура организации блоков задач с оценкой их сложности

Каждый блок содержит множество вариантов однородных задач. Сложность (SP(*,*)-трудоемкость) блока задания оценивается преподавателем-экспертом по следующей методике: эксперт сам решает один вариант задач и отмечает, за какое время непрерывной работы ему удалось решить эти задачи. Например, вариант 5 из блока (1,1) эксперт может решить за 30 мин/раб, т.е. $SP(1,1)=30$. Как показывает опыт, студенту после освоения раздела курса требуется на решение своего варианта задач в пять раз больше времени непрерывной работы, т.е. его сложность (трудность решения задач) $SC(1,1)=5*SP(1,1)=150$. Разумеется, сложность задач из остальных блоков, оценивается аналогично. В результате сложность многоуровневого практико-ориентированного курса в целом можно оценить по формуле $SK = \sum_i \sum_j SP(i, j)$.

По учебному плану, для освоения курса отводится время на самостоятельную работу (СРС), например, в рамках курса «Вычислительная математика» - 108 часов. Исходя из этого, сделаем следующее разделение времени студента: в практико-ориентированных курсах 40% времени отведем на самостоятельное освоение теоретического материала; 60% времени на

самостоятельное решение своего варианта задач из курса. Таким образом, студенту для решения своего варианта заданий отводится всего 64,8 (час/раб), т.е. сложность практико-ориентированного курса не должно превышать $5 \cdot SK < 64,8$ (час/раб).

1.3. Пример организации техногенной образовательной среды

Рассмотрим пример проектирования программного обеспечения многомодульной информационной системы MYKNITU.

На рис. 15 перечисляются необходимые требования к функционалу системы.



Рис.15.Требования к функционалу информационной системе

Система должна:

- поддерживать учебный процесс в традиционном и метрико-ориентированном учебных форматах;
- иметь редактор, который позволяет разместить учебный материал;
- иметь редактор, который позволяет организовать тестирование в баллах и метриках сложности;
- оценить качество усвоенных знаний с помощью тестирования;
- управлять процедурой организации и контроля за выполнением лабораторных работ;
- контролировать активность студентов;

- размещать справочную информации (расписания экзаменов, мероприятий, новости, объявления);
- формировать диаграммы достижений студента в метрической шкале качества владения компетенцией;
- on-line и off-line общение;
- формировать портфолио студентов.

На рис. 16. представлена общая схема проектируемой системы.



Рис. 16. Общая схема проектируемой системы

На рис. 17 представлена общая структура организации проектируемой системы.

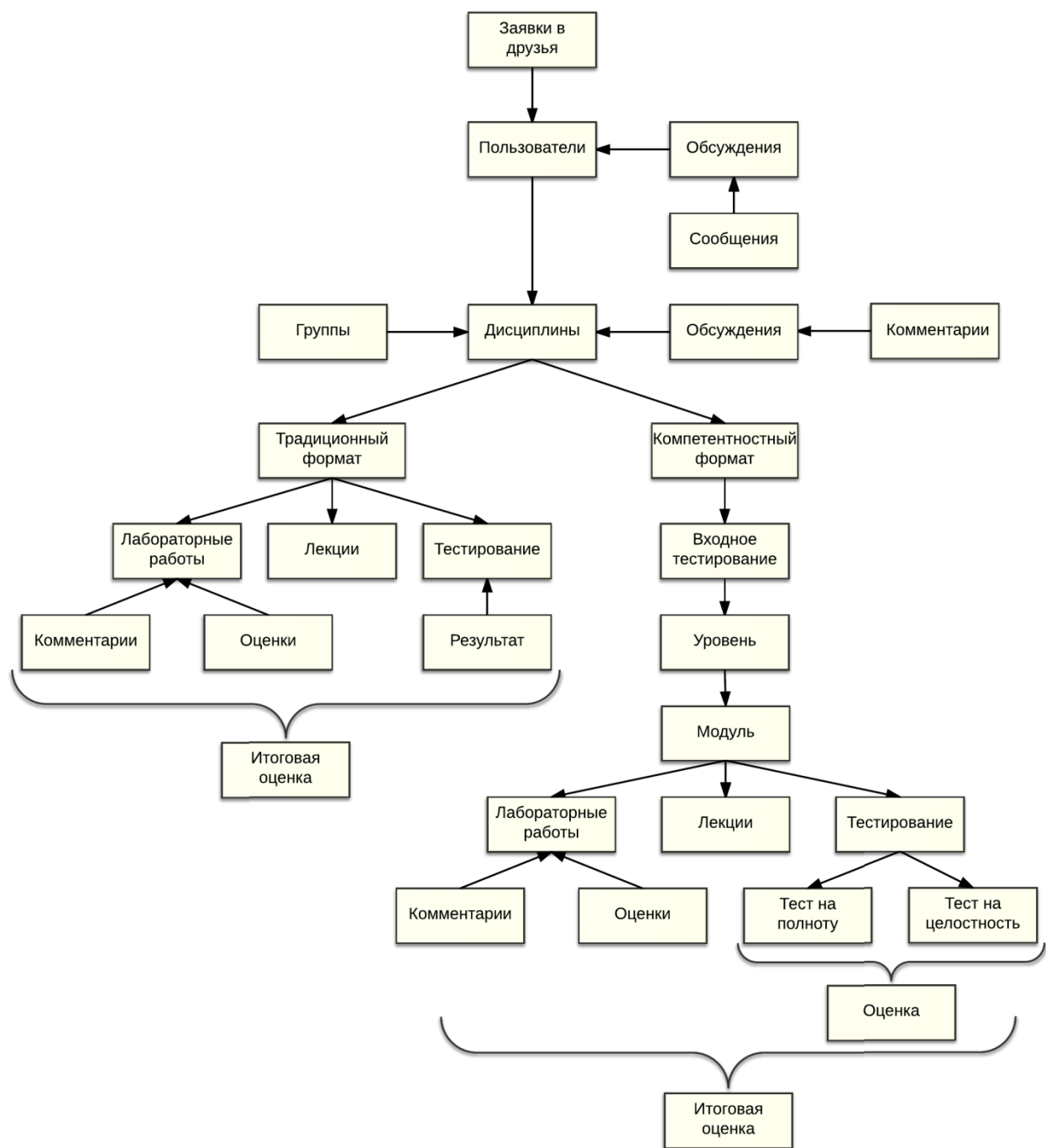


Рис. 17. Структура организации системы

Для создания информационной системы, разработчик на начальном этапе должен спроектировать базу данных, для этого строится инфологическая модель базы данных. На рис. 18 изображена инфологическая модель базы данных.

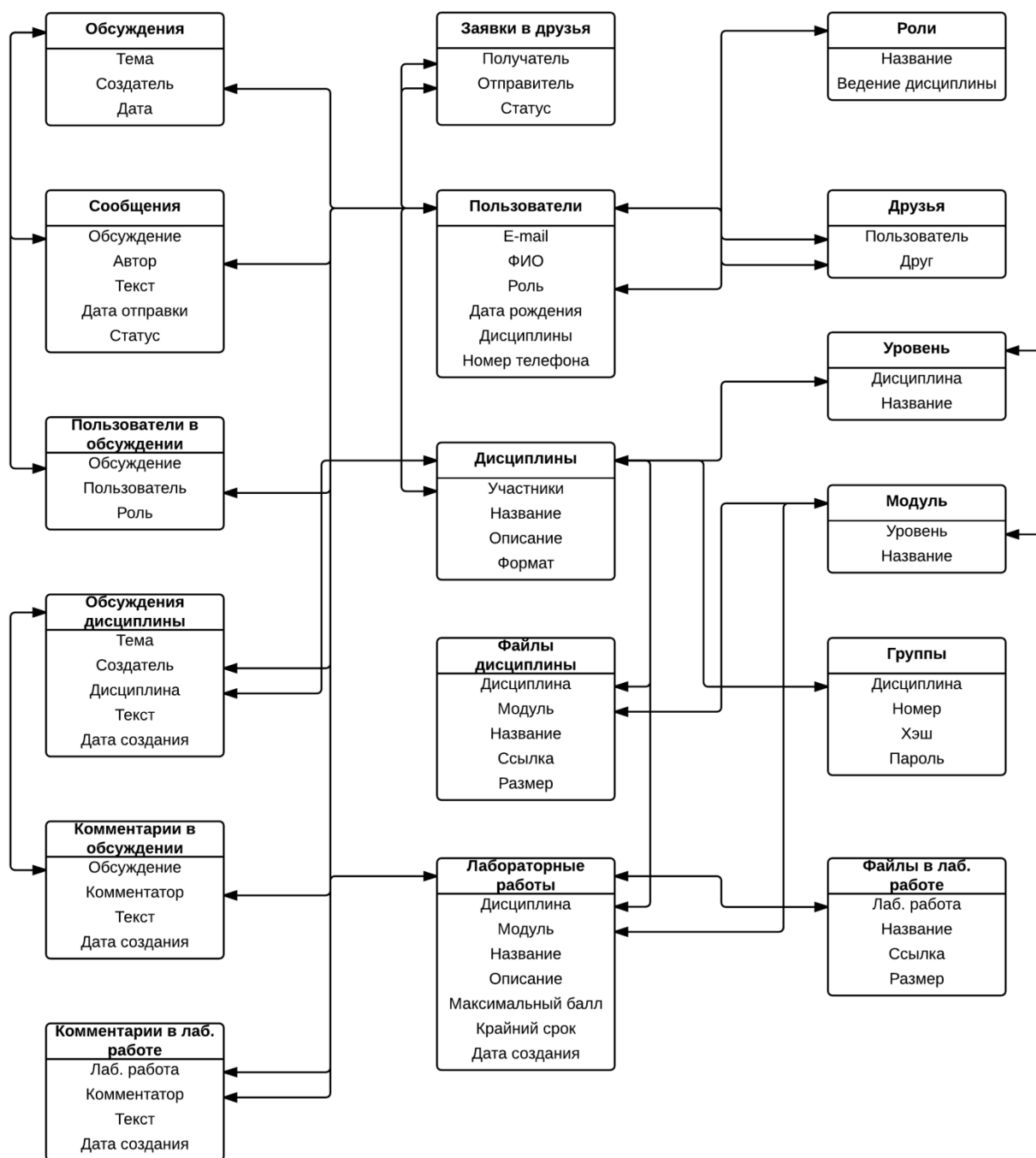


Рис. 18. Инфологическая модель базы данных

После создания концептуальной и инфологической модели необходимо представить визуализацию интерфейсов. На рис. 19 представлена схема экранов на мобильном устройстве.

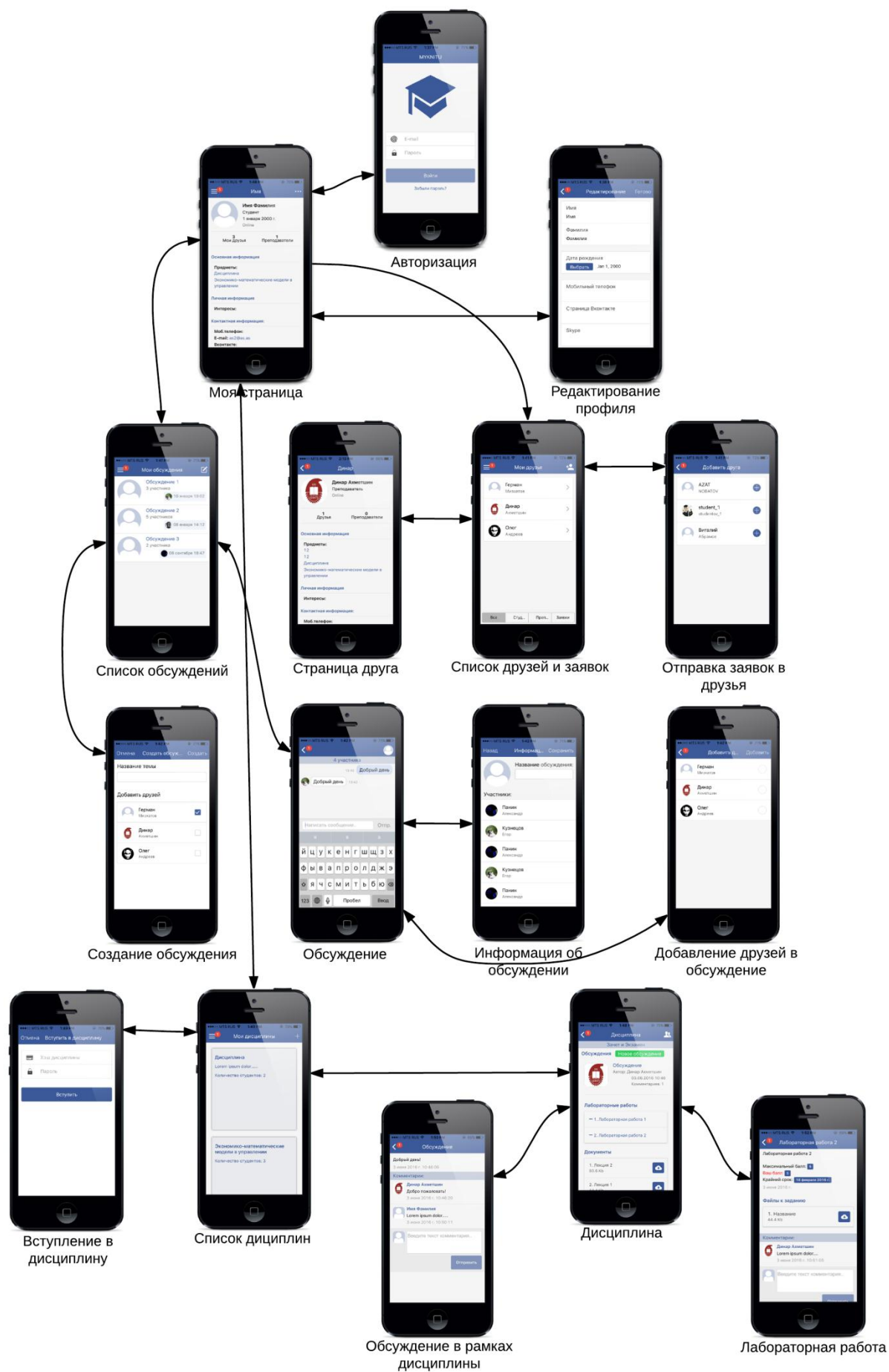


Рис. 19. Схема экранов на мобильном устройстве

На рис. 20 представлена схема работы пользователя.

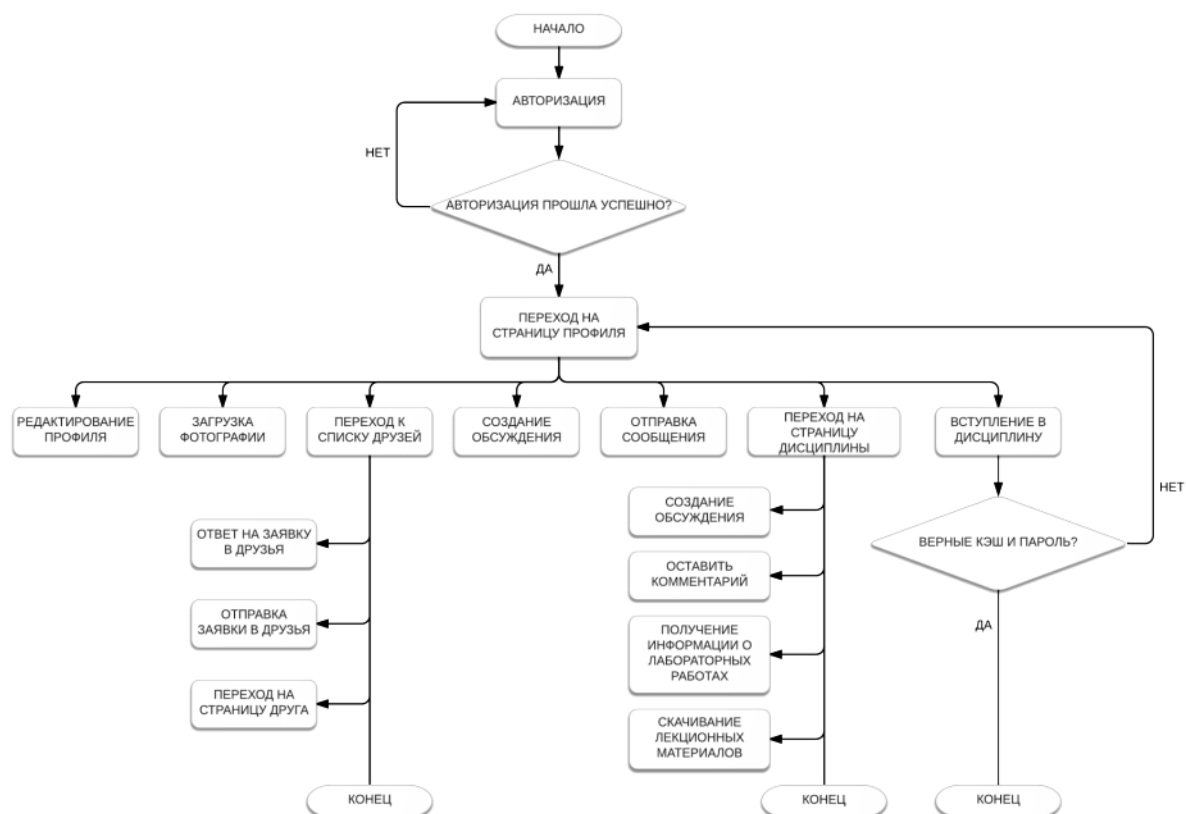


Рис. 20. Блок-схема алгоритма работы пользователя

На рис. 21 представлена структура организации учебного процесса в традиционном формате.



Рис. 21. Блок схема работы традиционного формата обучения

Учебный процесс реализуется в следующей последовательности:

1. Преподаватель создает дисциплину традиционного формата обучения.
2. В настройках создает группу и сообщает студентам хеш дисциплины и пароль.
3. В рамках созданной дисциплины преподаватель создает разделы обсуждения, раздел документы (лекционный материал, задания и т.д.), лабораторные работы.
4. Студенты при успешном входе в дисциплину выполняют указания преподавателя и получают соответствующие оценки.
5. У преподавателя на странице результаты автоматически формируется журнал успеваемости у студентов.

На рис. 22 представлена структура организации учебного процесса в компетентностном формате.



Рис. 22. Блок схема работы компетентностного формата обучения

Перечень действий преподавателя при компетентностном формате обучения:

1. Преподаватель создает дисциплину.
2. В настройках создает группу и сообщает студентам хеш дисциплины и пароль.
3. Формирует тест для входного тестирования по уровню сложности (например, 3 простых вопроса, 3 – средней уровни сложности и 3 – сложных вопроса).
4. Организует среду для обсуждения дисциплины.
5. Формирует компетенции (из выпадающего списка выбирает коды компетенций).
6. В рамках выбранных компетенций формирует раздел лекционного материала, лабораторные работы (обязательные лабораторные работы и дополнительные) по уровню сложности и тестирование (на полноту и целостность).

Перечень действий студента при компетентностном формате обучения:

1. Проходит входное тестирование, если студент неправильно отвечает подряд на 3 простых вопроса, то автоматически относится к группе неуспевающих. Это означает, что студент самостоятельно выполняет не относящиеся к учебному процессу лабораторные работы и если он выполнит и преподаватель примет их, то студент переходит к основной программе.

2. Выполняет лабораторные работы по очередности, каждая созданная лабораторная работа преподавателя имеет обязательные связи к созданным ранее лабораторным работам.

По результатам лабораторных работ и тестирований для каждого студента формируются графики компетенций.

1.4. Реализация программного продукта

После эскизного проектирования необходимо перейти к этапу разработки программного продукта. Программный продукт может быть реализован в любой программной среде, в данном пособии рассматривается процесс создания программного продукта на языке программирования Python с использованием фреймворка Django.

Разработка системы велась в операционной системе Linux Debian, использовалась СУБД PostgreSQL.

Обоснование выбора фреймворка Django:

Django – это свободный высокоуровневый веб-фреймворк на языке Python, реализованный на основе архитектуры MVC (Модель-Представление-Контроллер).

MVC представление:

- Модель (Model). Модель предназначена для работы с данными, которая взаимодействует с базой данных.
- Представление (View). Представление отвечает за то, как эти данные будут выглядеть.
- Контроллер (Controller). Контроллер является посредником между моделью, представлением и посетителями сайта.

Django называют MTV-ориентированной средой разработки (Модель-Шаблон-Представление) поскольку здесь View выполняет функцию контроллера, а Template – представления (рис.23).

1. Модель (Model). Модель является важнейшей составляющей приложения, при помощи которой происходит обращение к данным при запросах. Любая модель является стандартным классом Python.

2. Представление (View). На слой представления возложена задача обеспечения логики получения доступа к моделям и применения соответствующего шаблона. Представление является своеобразным мостом между моделями и шаблонами.

3. Шаблон (Template). Этот слой является формой представления данных. Шаблон имеет свой собственный простой метаязык.

Архитектура Django представлена на рисунке 16.

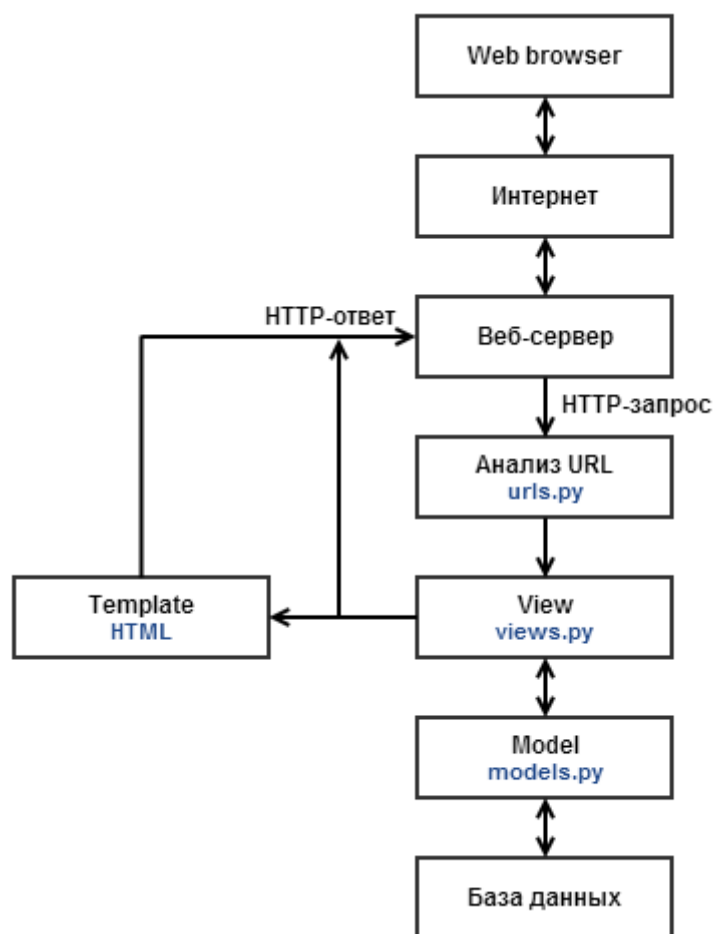


Рис.23. Архитектура Django

При запросе к странице Django-проекта, используется следующий алгоритм.

1. Django определяет какой из корневых модулей URLconf использовать. Обычно, это значение настройки `ROOT_URLCONF`.
2. Django загружает модуль конфигурации URL и ищет переменную `urlpatterns`.
3. Django перебирает каждый URL-шаблон по порядку, и останавливается при первом совпадении с запрошенным URL-адресом.
4. Если одно из регулярных выражений соответствует URL-у, Django импортирует и вызывает соответствующее представление.
5. Если ни одно регулярное выражение не соответствует, или возникла ошибка на любом из этапов, Django вызывает соответствующий обработчик ошибок.

Встроенные возможности Django:

- ORM, API доступа к БД с поддержкой транзакций;

ORM — технология программирования, которая связывает базы данных с концепциями объектно-ориентированных языков программирования, создавая «виртуальную объектную базу данных».

- собственный веб-сервер для разработки, который ускоряет процесс разработки на Python;
- встроенный интерфейс администратора, разработчикам не нужно создавать его отдельно;
- расширяемая система шаблонов с тегами и наследованием;
- система кеширования;
- интернационализация;
- встроенная авторизация и аутентификация;
- библиотека для работы с формами (наследование, построение форм по существующей модели БД);
- большая документация для изучения Django и т.д.

Процесс создания информационной системы

В начальном этапе расписываем структуру базы данных, в фреймворке Django принято называть это моделями, для этого в созданном приложении присутствует файл `models.py`, где в виде классов мы расписываем структуру базы данных, рассмотрим на примере класс профиля пользователя:

```
class UserProfile(models.Model):
    ROLE_CHOICES = (
        (1, u'Студент'),
        (2, u'Преподаватель'),
    )
    user = models.ForeignKey(User, unique=True, verbose_name =
'Пользователь')
    nameuser = models.CharField(max_length=200, verbose_name = 'Имя')
    family = models.CharField(max_length=200, verbose_name = 'Фамилия')
    role = models.PositiveIntegerField(choices=ROLE_CHOICES,
verbose_name = 'Роль', default=ROLE_CHOICES[0][0])
    phonenummer = models.CharField(max_length=100, verbose_name =
'Номер телефона', blank=True)
    birthday = models.DateField(verbose_name = 'Датарождения', blank=True,
null=True,)
    vk = models.CharField(max_length=200, verbose_name = 'Вконтакте',
blank=True)
```

```

    skype = models.CharField(max_length=200, verbose_name = 'Skype',
blank=True)
    interests = models.TextField(verbose_name = 'Интересы', blank=True)
    img = ResizedImageField(upload_to='Images', blank=True, verbose_name =
", max_width=217, default='foto.png')
    def __unicode__(self):
        return '%s %s' % (self.nameuser, self.family)

class Meta:
    ordering = ["nameuser"]
    verbose_name = 'Пользователь'
    verbose_name_plural = 'Пользователи'
    def img_or_default(self, default_path="default/ContactPhoto5.png"):
        if self.img:
            return self.img
return default_path

```

Далее разработчик для работы создания форм использует в работе инструменты по созданию форм, в Django формами принято работать в файле forms.py. Рассмотрим фрагмент кода созданного класса в файле forms.py

```

class MessageForm(ModelForm):
    class Meta:
        model = Messages
        fields = ('message',)
    def __init__(self, *args, **kwargs):
        super(MessageForm, self).__init__(*args, **kwargs)
        self.fields['message'].widget = Textarea(attrs={'rows':3,
            'class': 'form-control', 'onkeypress': 'usl(this.value, event,
this.form)', })

```

Далее разработчик расписывает логику работы приложения, для этого в созданном приложении рекомендуется писать код в существующем файле views.py. Рассмотрим фрагмента кода получение и добавление сообщений в текущем обсуждении

```

@login_required(login_url='/login/')
def discussion(request, id):
    user = UserProfile.objects.get(user=request.user)
    dis = DiscussionThemes.objects.get(id=id)

```

```

        if not DiscussionUsers.objects.filter(user__username = request.user,
discuss_theme=dis):
            return redirect('/discussions/')
        disuser = DiscussionUsers.objects.filter(discuss_theme=dis)
        dismessage =
DiscussionMessages.objects.filter(discuss_theme=dis).order_by('id')
        if request.method == "POST":
            form = DiscussionForm(request.POST, request.FILES)
            if form.is_valid():
                form_for_save = form.save(commit=False)
                form_for_save.user = request.user
                form_for_save.discuss_theme =dis
                form_for_save.save()
                return redirect('/discussion/'+id)
            form = DiscussionForm()
            return render(request, 'discussion.html', { 'dis':dis, 'disuser':disuser,
'dismessage':dismessage, 'form':form, 'user':user})

```

В конце разработчик создает htmlстраницу и делается вывод данных при помощи шаблонных тегов. Ниже представлен пример вывода всех дисциплин

```

{% foraindisciplina %}

```

```

        <div class="col-xs-4">
            <div class="disciplina">
                <a href="/disciplina/{ { a.disciplinaid.id } }">
                    <h5>{ { a.disciplinaid.name } }</h5>
                    <p>{ { a.disciplinaid.description } }</p>
                    <p>Количествостудентов: { {
a.disciplinaid.disciplinainvite_set.all.count } }</p>
                </a>
            </div>
        </div>

{% endfor %}

```

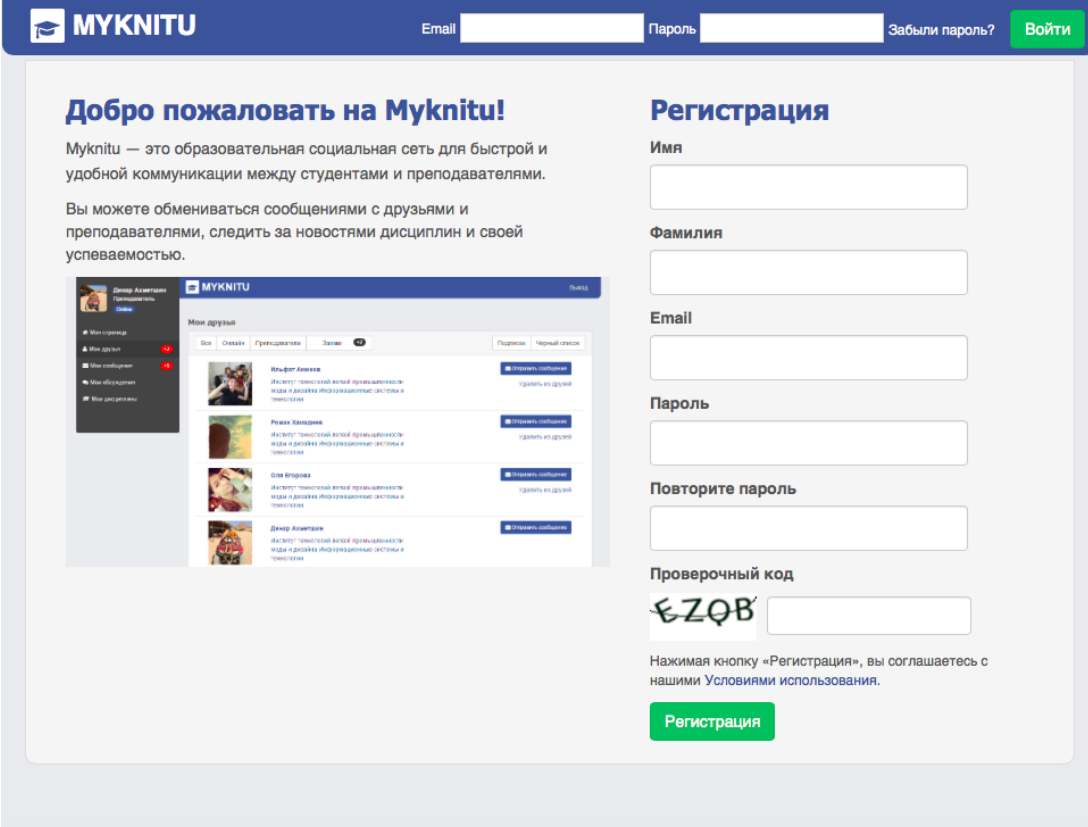
Таким подходом создается все приложение, т.е. вначале расписывается модель, затем создается на основе модели класс для форм, после чего

программируется логика работы с моделями и формами, а после данные передаются в шаблон html

В конечном итоге разработан программный продукт myknitu.ru

Ниже продемонстрирована работа информационной системы

1. Пользователь авторизуется в системе (рис.24)



MYKNITU Email Пароль Забыли пароль? Войти

Добро пожаловать на Myknitu!

Myknitu — это образовательная социальная сеть для быстрой и удобной коммуникации между студентами и преподавателями.

Вы можете обмениваться сообщениями с друзьями и преподавателями, следить за новостями дисциплин и своей успеваемостью.

Регистрация

Имя

Фамилия

Email

Пароль

Повторите пароль

Проверочный код

Нажимая кнопку «Регистрация», вы соглашаетесь с нашими [Условиями использования](#).

Регистрация

Рис. 24. Главная страница

2. Система проверяет пользователя к какой группе он относится
3. Если пользователь относится к группе преподаватель, то для него доступен следующий функционал:
 - а. Создание дисциплины с выбором формата обучения (традиционный формат, компетентностный формат, АВС компетентностный формат)

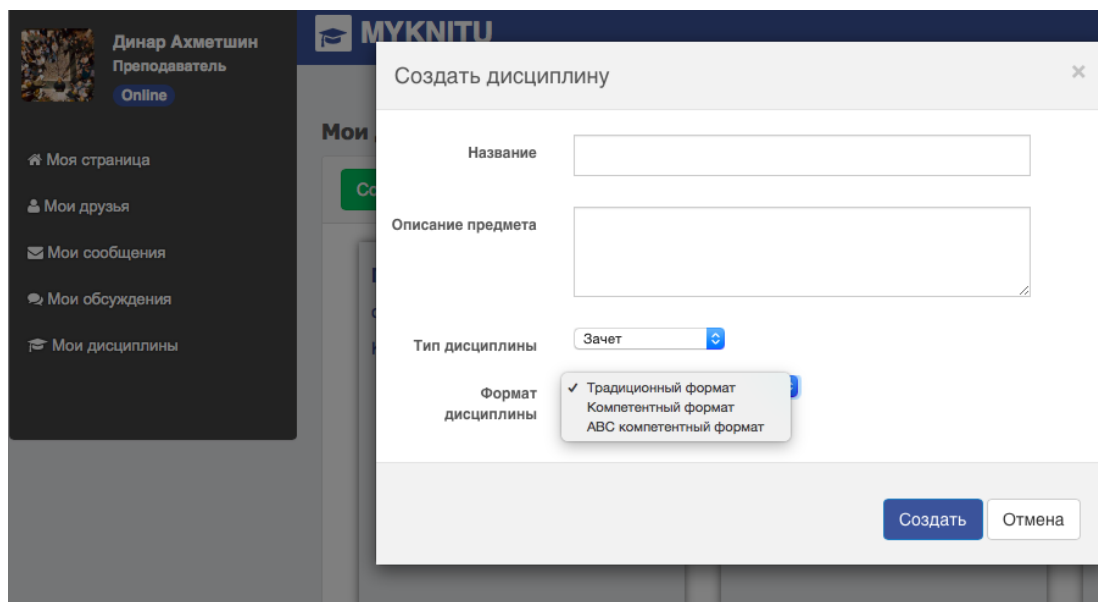


Рис. 25. Создание дисциплины

б. В зависимости от выбранной дисциплины у преподавателя формируется страница формирования курса дисциплины

с. В настройках дисциплины преподаватель создает группу студентов и сообщает другим пользователям код доступа к этой дисциплине

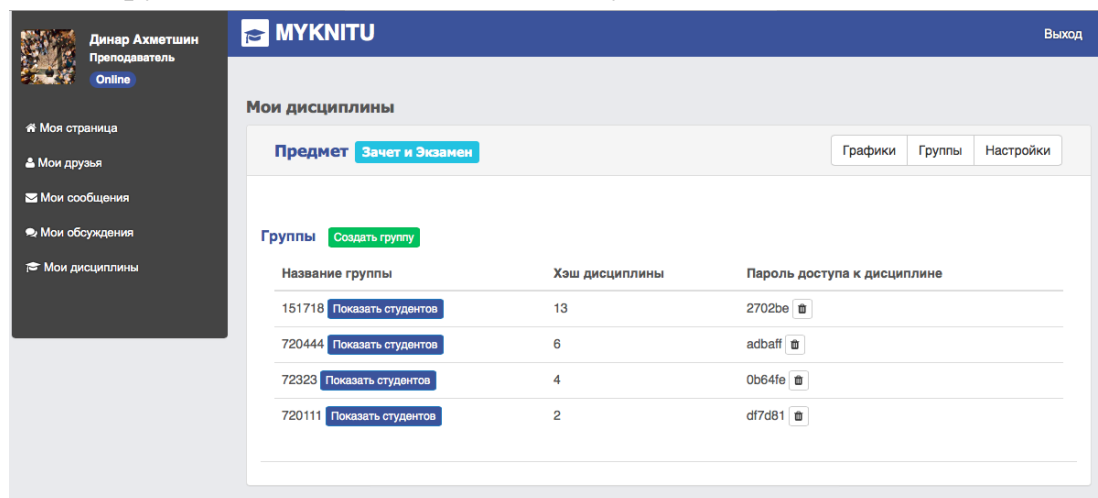


Рис. 26. Управление группами в выбранной дисциплины

д. В каждом формате дисциплины присутствуют следующие модули: обсуждения, документы, лабораторные работы

е. Если выбран традиционный формат обучения:

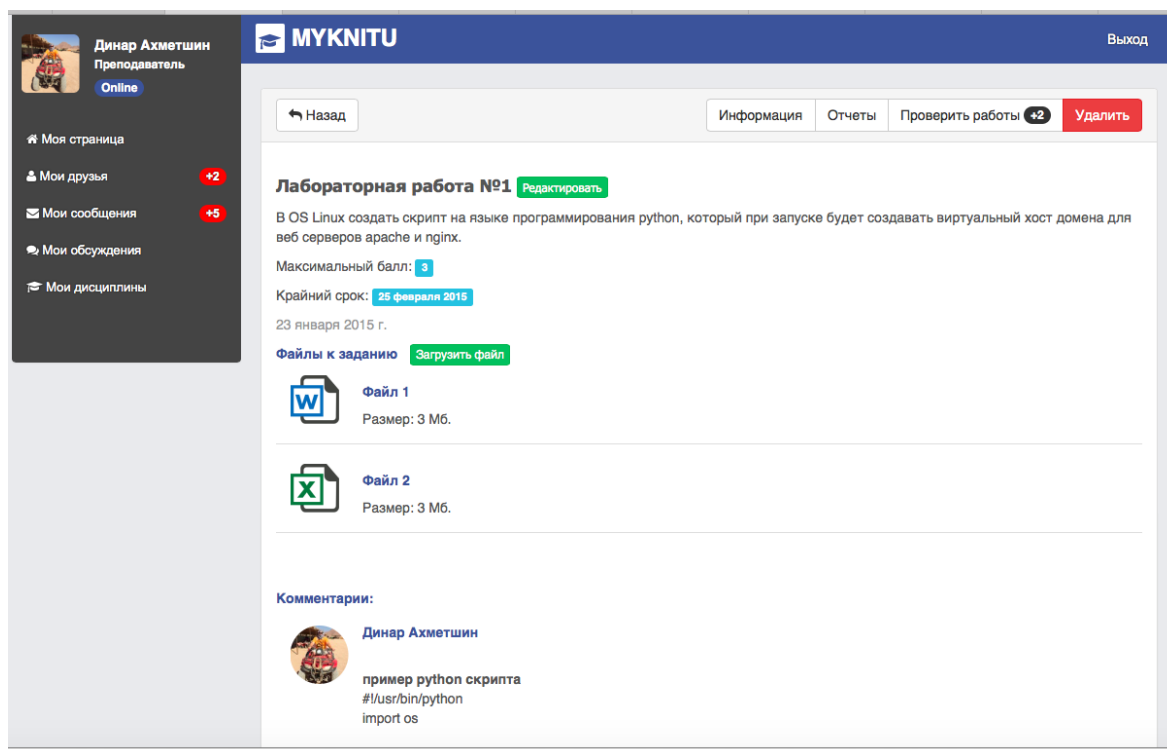


Рис. 27. Скриншот лабораторной работы

Пример:

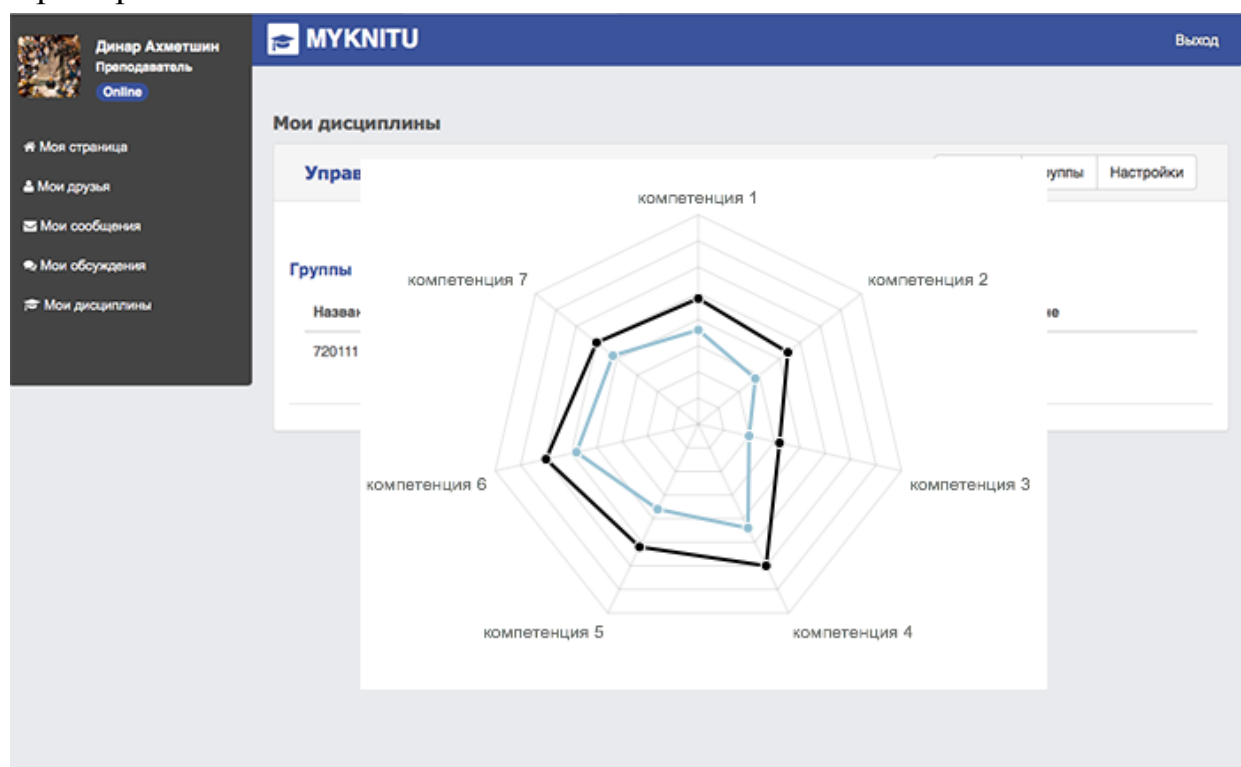


Рис. 28. График освоенных компетенций в выбранной группе

Черная линия на рис. 28 – максимальные возможные баллы, синяя линия – набранные баллы студента.

Если выбран ABC-компетентностный формат обучения:

Выполняются аналогичные действия как в компетентностном формате, но с одним отличием. Преподаватель создает работы, разделяя их по уровню сложности на работы по формализации, конструированию и исполнению.

Новая лабораторная работа

Название: Лабораторная работа №3

Описание:

Срок сдачи: 30.05.2015

Обязательная работа: Лабораторная работа №2

Уровень сложности (минуты/раб)

Уровень сложности	Время (минуты)
Формализация (А):	120
Конструирование (В):	140
Исполнение (С):	150

Создать Отмена

Рис. 29. Пример создания лабораторной работы

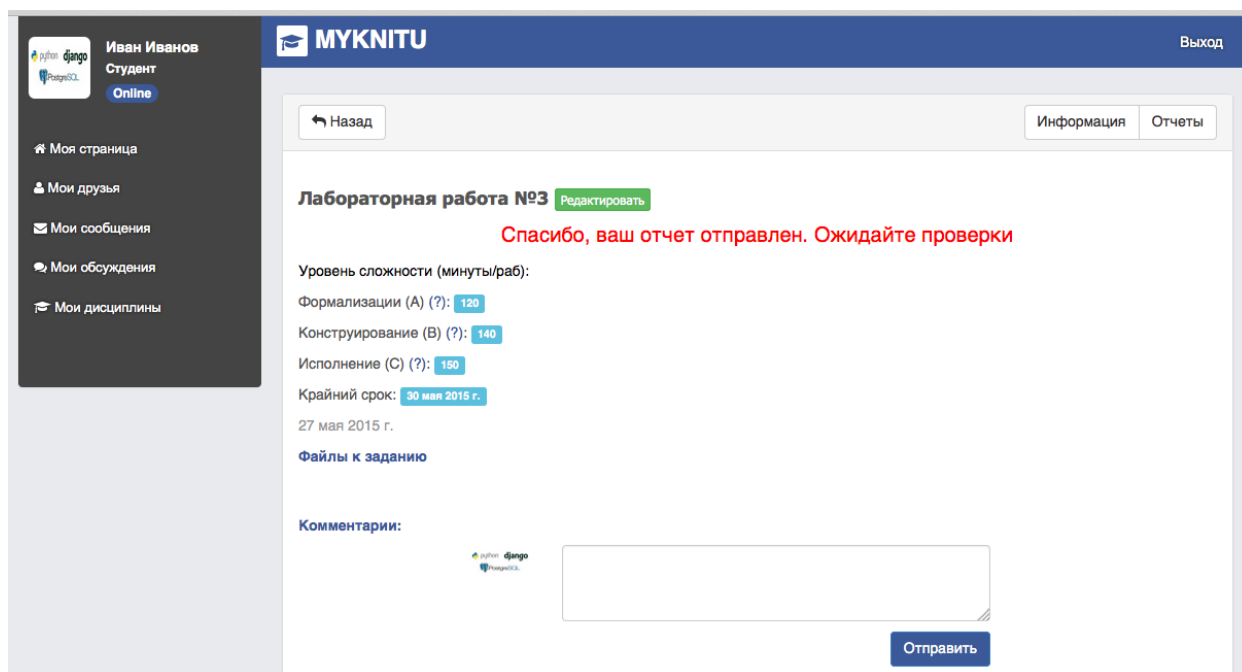


Рис.30. Отправка отчета студентом

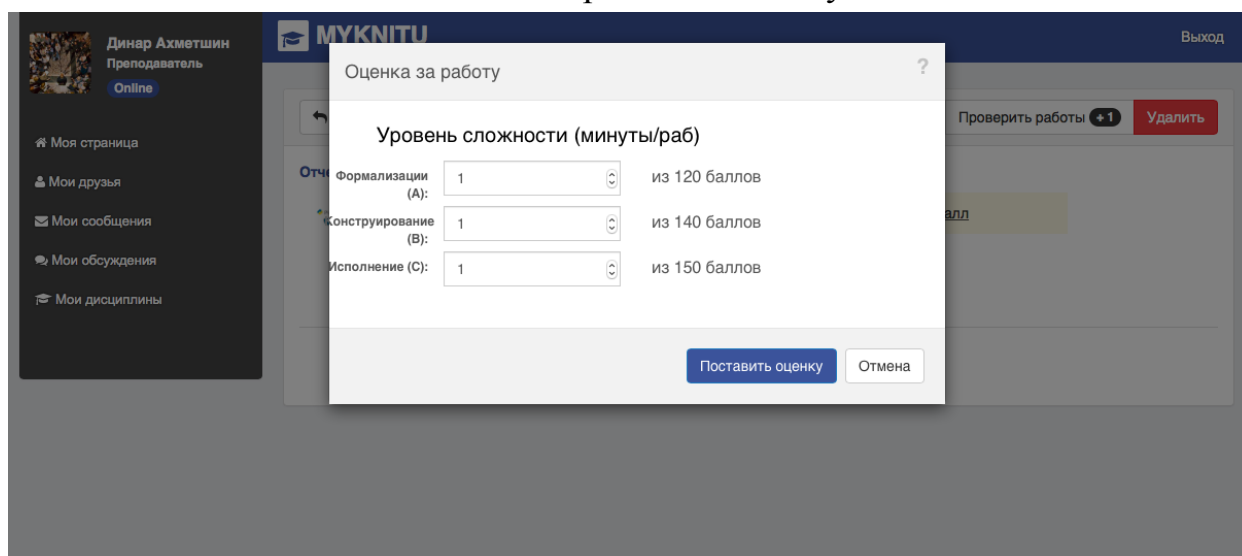


Рис.31. Преподаватель ставит оценку за л/р

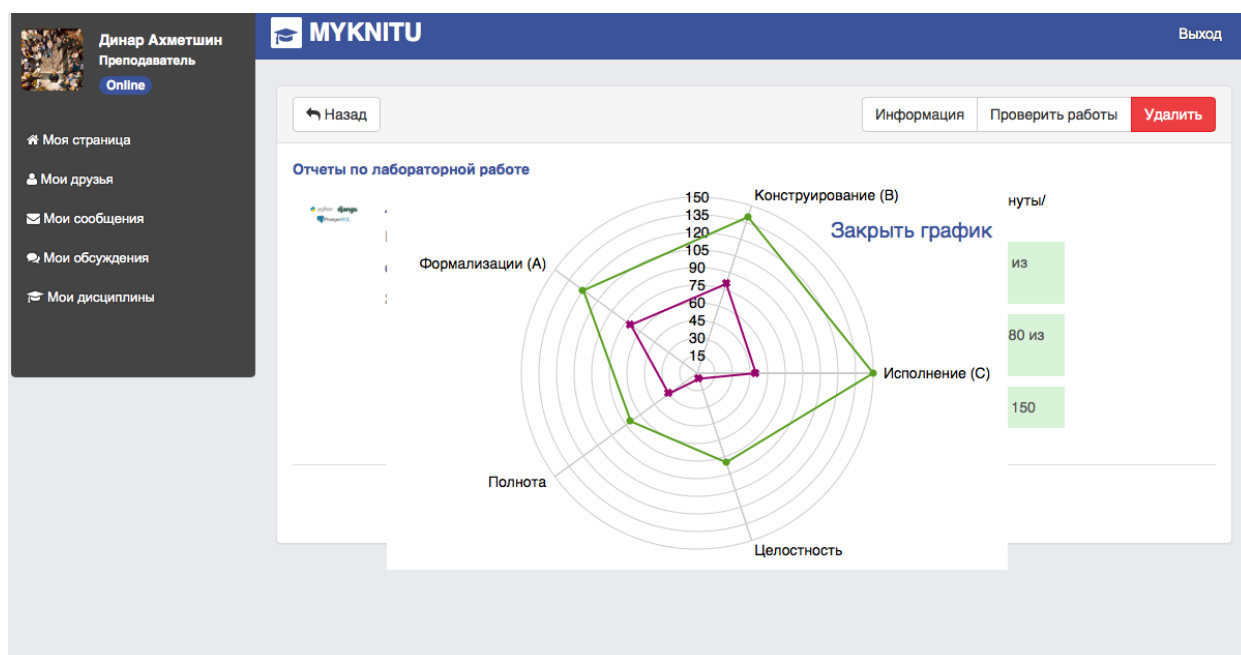


Рис.32. Формируется график выполнения данной работы

Глава 2. Язык программирования Python

2.1. Python и его особенности

Python является мощным, но в то же время простым для изучения языком программирования, где предоставлены проработанные высокоуровневые структуры данных. Язык обладает теми же возможностями, что и другие языки программирования: динамичностью, поддержкой ООП и кросс-платформенностью, что позволяет легко и быстро внедрять проекты в разные операционные системы. Синтаксис Python минималистичен, библиотека включает множество полезных функций. Язык пригоден для решения разнообразных задач, он прекрасно подойдёт для написания сценариев и быстрой разработки приложений в любой области разработки программного обеспечения и на большом количестве платформ.

С точки зрения профессионального программиста, легкость Python - залог высокой производительности труда: программы на Python короткие и требуют меньше времени на разработку, чем программы на многих других популярных языках.

С помощью Python можно написать много разнообразных задач:

- системные скрипты;
- программы с графическим интерфейсом;
- веб-системы от простых сайтов до сложных CRM-систем;
- приложения баз данных;
- программы для математических и научных вычислений;
- игры, медиа-проекты и другие.

Преимущества Python:

- Python поддерживает несколько парадигм программирования, в том числе структурное, объектно-ориентированное, функциональное, императивное и аспектно-ориентированное;

- язык распространяется под свободной лицензией Python Software Foundation License, позволяющей использовать его без ограничений в любых приложениях;
- импортируемость – Python импортирован и работает почти на всех известных платформах;
- чёткий и последовательный синтаксис, модульность и масштабируемость, благодаря чему исходный код написанных на Python программ легко читаем;
- большое количество документации, в том числе и на русском языке;
- большая коллекция инструментов стандартной библиотеки;
- возможность подключения сторонних разработок;
- возможность интеграции с программами C/C++ и другие.

2.2. Синтаксические правила языка Python

Про Python говорят, что это язык программирования с достаточно ясным и легко читаемым кодом.

В Python отсутствуют операторные скобки типа *begin ... end* или *DO ... LOOP*. Вместо них в составных операторах (ветвления, циклы, определения функций) используются отступы от начала строки (пробелы).

В Python не надо объявлять тип переменной, достаточно просто присвоить ей значение. Python чувствителен к регистру: «s» и «S» являются разными переменными.

Конец строки не требует точки с запятой, как принято в некоторых других языках.

Комментарии в Python начинаются с «#» и продолжаются до конца строки, несколько строк можно закомментировать при помощи тройного апострофа в начале и конце комментария.

В Python пустые строки, пробелы и комментарии обычно игнорируются. Пустые строки игнорируются в файлах (но не в интерактивной оболочке, когда они завершают составные инструкции). Пробелы внутри инструкций и выражений

игнорируются практически всегда (за исключением строковых литералов, а также когда они используются для оформления отступов). Комментарии игнорируются всегда.

Присваивание в Python обозначается «=», а равенство знаком «==».

2.3. Средства программирования на Python

Python имеет возможность работы в режиме интерпретатора, в котором команды и операции выполняются сразу после их ввода.

Запустить интерактивную оболочку можно разными способами. Например, в интегрированной среде разработки или в системной консоли.

Вызов интерпретатора Python в командной строке осуществляется набором команды *python*.

В интерактивной оболочке команды и операции вводятся с клавиатуры после знака приглашения интерпретатора `>>>`. Ввод каждой операции завершается нажатием на клавишу *Enter*, после чего Python выполняет эту операцию и выдаёт результат или сообщение об ошибке.

Однако в интерактивной оболочке неудобно работать с файлами программ. Кроме того, полезно видеть текст программы одновременно с результатами её выполнения. Такие функции обеспечивают интегрированные среды разработки IDE (IntegratedDevelopmentEnvironment).

Самая простая IDE для Python называется IDLE. В этой среде можно редактировать тексты программ в окне редактора и запускать их на выполнение.

По отношению к интерпретируемым языкам программирования часто исходный код называют скриптом. Файлы с кодом на Python обычно имеют расширение *.py*.

Скрипты можно готовить в любом текстовом редакторе. Кроме того, существуют специальные программы для разработки.

Запускать подготовленные файлы можно не только в IDLE, но и в консоли с помощью команды `python адрес/имя_файла`.

Кроме того, существует возможность настроить выполнение скриптов с помощью двойного клика по файлу.

2.4. Исключения в Python

Иногда во время выполнения программы в Python возникают ошибки, которые называются исключениями. Они появляются при попытке выполнения недопустимых операций. Например, при вводе данных с несоответствующим типом или количеством элементов или при попытке использования функций с неверным типом.

Таблица 1.1. Основные типы исключений

Тип исключения	Описание
IOError	Исключение генерируется, если невозможно выполнить операцию ввода/вывода. Например, при открытии на чтение несуществующего файла.
IndexError	Генерируется, если не найден элемент с указанным индексом.
KeyError	Исключение генерируется, если в словаре не найден указанный ключ.
NameError	Генерируется, если не найдено имя (функции, переменной и т.п.).
SyntaxError	Генерируется при обнаружении синтаксической ошибки в коде.

Тип исключения	Описание
TypeError	Генерируется, если стандартная операция или функция применяется к объекту неподходящего типа.
ValueError	Генерируется, если стандартная операция или функция принимает аргумент с неподходящим значением.
UnicodeError	Ошибка, связанная с кодированием/раскодированием unicode в строках.

Обработка исключений при помощи конструкции try/except

Типичный способ обработки исключений – конструкция *try/except*.

Блок *try* генерирует исключение. Если код содержит какие-либо ошибки, на экране появляется содержимое блока *except*. При этом оставшаяся часть блока *try* не выполняется.

```
try:
    <блок действий>
except<тип исключения>:
    <блок действий>
else:
    <блок действий>
```

После всех блоков с *except* в конструкции с оператором *try* можно добавить заключительный блок *else*. Он будет исполнен лишь в том случае, если блок *try* работает безошибочно.

#Демонстрация обработки исключительных ситуаций при помощи try/except

```
try:
    num = int(input('Введите целое число: '))
except:
    print('Введите целое число!')
else:
    print('Ваше число: %s' % num)
```

Оператор *except* позволяет точно указать, какой тип исключения будет обрабатываться.

Чтобы назначить один и только один тип исключения, достаточно вписать его после *except*.

```
# Перехват исключения ValueError
try:
    num = int(input("Введите целое число: "))
except ValueError:
    print("Вы ввели не число!")
```

Указывать типы исключений – полезный приём, который позволяет в каждом частном случае применить разное действие. Но перехват всех исключений, показанный в первом примере, может быть не безопасен.

В примере выше вы встретись с функциями *input()* и *print()*, которые нельзя оставить без внимания.

Ввод текста пользователем в Python осуществляется при помощи функции *input()*.

Когда данная функция выполняется, то поток выполнения программы останавливается в ожидании данных, которые пользователь должен ввести с помощью клавиатуры. После ввода данных и нажатия *Enter*, функция *input()* завершает свое выполнение и возвращает результат, который представляет собой строку символов, введенных пользователем. Строковый аргумент функции выводится на экран, до запроса пользователя о вводе.

Для вывода результатов работы используется функция *print()*.

```
# Демонстрация функций input() и print()
```

```
a = input('Ваш любимый афоризм? ')
print('Счастливые часов не наблюдают')
```

Вопросы для повторения

1. Что для вас является главным при выборе языка программирования?
 2. Выделите основные преимущества языка Python.
 3. Какие задачи можно решать с помощью Python?
 4. Знаете ли вы какие-нибудь известные компании или организации, использующие Python?
 5. Выделите основные синтаксические правила в Python.
 6. Какие исключения могут возникнуть во время выполнения программы в Python?
 7. Как можно избежать возникновения исключения?
-
-

Глава 3. Условные операторы и циклы

3.1. Условный оператор *if*

Ход выполнения программы может быть линейным, то есть таким, когда выражения выполняются, начиная с первого и заканчивая последним, по порядку, не пропуская ни одной строки кода. Но чаще бывает совсем не так. Нелинейность действий встречается в любой непростой задаче. Например, тогда, когда часть кода должна выполняться лишь при определенном значении конкретной переменной.

Условный оператор *if* является основным инструментом выбора в Python, который отражает большую часть логики программ.

Условная инструкция *if*(если) в языке Python – это типичная условная инструкция. Синтаксически сначала записывается часть *if* с условным выражением, далее могут следовать одна или более необязательных частей *elif*(иначе если) с условными выражениями (*elif* в Python является сокращением от *else if* и используется для организации вложенных условий) и, наконец, необязательная часть *else*(иначе).

```
if<условие>:  
    <блок действий>  
elif<условие>:  
    <блок действий>  
else:  
    <блок действий>
```

Первая строка конструкции *if*– это заголовок, в котором проверяется условие выполнения строк кода после двоеточия (тела конструкции).

Если условия при *if* и *elif* оказывается ложным, то выполняется блок кода при инструкции *else*.

Про Python говорят, что это язык программирования с достаточно ясным и легко читаемым кодом. Это связано с тем, что в нем сведены к минимуму вспомогательные элементы (скобки, точка с

запятой), а для разделения синтаксических конструкций используются отступы от начала строки. Учитывая это, в конструкции *if*, код, который выполняется при соблюдении условия, должен обязательно иметь отступ вправо.

Python 3.0 считает ошибкой непоследовательное смешивание пробелов и символов табуляции в пределах блока (то есть, когда величина отступов из смешанных символов в пределах блока может отличаться в зависимости от интерпретации ширины символов табуляции). В Python 2.6 допускается подобное смешивание.

```
# Демонстрация программы, которая выводит на экран смайл,  
зависящий от настроения пользователя  
print('Как дела?')  
answer = input('хорошо, плохо, нормально? ')  
if answer == 'хорошо':  
    print(':-)')  
elif answer == 'плохо':  
    print(':-( ')  
elif answer == 'нормально':  
    print(':-/')  
else:  
    print('Выберите хорошо, плохо или нормально')
```

3.2. Циклы **for** и **while**

Циклом называется фрагмент алгоритма или программы, который может повторяться несколько раз.

Для организации циклов с параметром в языках программирования используется составной оператор **for** («для»), а в циклах с условием чаще всего используется составной оператор **while** («пока»).

В случае цикла с параметром количество повторений задаётся специальным выражением в заголовке, а в случае цикла с условием

при каждом следующем повторении требуется проверять условие прекращения цикла.

! Если при написании операторов в теле цикла допущена ошибка, условие прекращения цикла может не выполниться никогда и цикл окажется бесконечным (программа заиклится).

3.2.1. Цикл *for*

Цикл *for* является универсальным итератором последовательностей в Python. Он может выполнять обход элементов в любых упорядоченных объектах последовательностей. Инструкция *for* способна работать со строками, списками, кортежами, с другими встроенными объектами.

```
for <element> in <object>:  
    <блокдействий>  
else:  
    <блок действий>
```

Цикл *for* в языке Python начинаются с заголовка, где указывается переменная для присваивания, а также объект, обход которого будет выполнен. Вслед за заголовком следует блок действий, которые требуется выполнить.

```
# Демонстрация программы вывода чётных чисел в диапазоне 0-10  
for i in range(10):  
    if i % 2 == 0:  
        print(i)
```

3.2.2. Цикл *while*

Если количество повторений операций заранее неизвестно, но известно условие прекращения выполнения операций, используется цикл (составной оператор) *while*.

Инструкция *while* состоит из строки заголовка с условным выражением, тела цикла и необязательной части *else*, которая

выполняется, когда управление передается за пределы цикла без использования инструкции *break*.

```
while <условие>:  
    <блок действий >  
else:  
    <блок действий>
```

Блок кода (тело цикла) будет исполнен, если условие истинно, причем в конструкции *while* компьютер проверяет условие на истинность снова и снова, цикл за циклом, пока оно не окажется ложным.

```
# Демонстрация программы, задачей которой является получение  
# верного решения  
answer = 0  
while answer != '6':  
    answer = input('2+2*2 = ')
```

Иногда возникают ситуации, когда приходится использовать необычные инструкции.

```
while <условие>:  
    <блок действий>  
        if <условие>: break          # Выйти из цикла, пропустив часть else  
        if <условие>: continue      # Перейти в начало цикла  
else:  
    <блок действий>                # Выполняется, если не выполнен break
```

Инструкции *break* и *continue* могут появляться в любом месте внутри тела цикла *while* (или *for*), но, как правило, они используются в условных инструкциях *if*, чтобы выполнить необходимое действие в ответ на некоторое условие.

В языке Python:

- ***break*** производит переход за пределы цикла;
- ***continue*** производит переход в начало цикла;
- ***pass*** ничего не делает: это пустая инструкция, используемая как заполнитель.

Вопросы для повторения

1. Назовите основной инструмент выбора в Python, который отражает основную часть логики программ.
 2. Для чего используют *else* в условной конструкции?
 3. Придумайте пример, в котором можно использовать следующую условную конструкцию *if ... elif ... else ... endif*.
 4. Для чего нужны циклы?
 5. Какие способы организации циклов в Python вы знаете?
 6. Какой оператор используют для организации циклов с параметром?
 7. Расскажите про цикл с условием. В каких случаях его применяют?
 8. В чем заключаются основные различия между инструкциями *break* и *continue*?
 9. Для чего нужен *pass*?
-

Глава 4. Типы данных

Таблица 3.1. Типы данных Python

Тип объекта	Пример
Числа	<i>1234, 3.1415, 3+4j</i>
Строки	<i>'дом', b'a\x01c'</i>
Списки	<i>[1, [2, 'дерево'], 4]</i>
Кортежи	<i>(1, 'цветок', 4, 'U')</i>
Словари	<i>{'цветок': 'василек', 'дерево': 'дуб'}</i>
Множества	<i>set('abc'), {'a', 'b', 'c'}</i>
Файлы	<i>myfile = open('file.txt', 'r')</i>
Прочие базовые типы	None, логические значения

4.1. Числа

Числа в Python могут быть:

- обычными целыми (тип *int*);
- длинными целыми (тип *long*);
- вещественными (тип *float*);
- комплексными (они не будут рассматриваться и использоваться).

Для преобразования чисел из вещественных в целые и наоборот в Python определены функции *int()* и *float()*. Например, *int(36.6)* даст в результате 36, а *float(17)* даёт в результате 17.0 (десятичный разделитель – точка). Основные операции с числами приведены ниже.

Таблица 3.2. Основные операции с числами

Операция	Описание
$(x+y)$	Сложение
$(x-y)$	Вычитание

$(x*y)$	Умножение
(x/y)	Деление
$(x//y)$	Целочисленное деление
$(x\%y)$	Остаток от целочисленного деления x на y
$(x**y)$	Возведение в степень

!Стоит обратить внимание, что если при делении x и y – целые числа, то результат всегда будет целым числом. Для получения вещественного результата хотя бы одно из чисел должно быть вещественным.

Кроме того, в Python для операций с числами используются функции **abs()** (вычисление абсолютного значения модуля, $abs(-3) \rightarrow 3$), **pow()** (возведение в степень, $pow(2,3) \rightarrow 8$), **divmod()** (вычисление результата целочисленного деления и остатка, $divmod(17,5) \rightarrow (3,2)$) и **round()** (округление, $round(100.0/6) \rightarrow 17.0$).

Эти функции являются встроенными, это означает, что для их использования нет необходимости подключать дополнительные модули. Другие математические функции, такие как вычисление квадратного корня, синуса, логарифма и другие требуют подключения модуля **math**.

Модули – это файлы с кодом, пригодные для использования в других программах. Модуль обычно содержит в себе функции, относящиеся к одной и той же области. Так, модуль **random** содержит функции, связанные с генерацией случайных чисел и получением случайных результатов. Например, функция **randint()** возвращает случайное целое число.

```
# Подключение модуля в начале программы
import math
# Подключение функции
from имя_модуля import функция1, ... функцияN
```

1. Практическое задание «Числа и математические

вычисления»

Напишите программу для решения квадратного уравнения. Пользователь должен иметь возможность ввода значений переменных.

Примечание: среда разработки IDLE.

4.2. Структуры данных

В Python определены такие структуры данных (составные типы) как последовательности и отображения (называемые также словарями).

Последовательности подразделяются на изменяемые и неизменяемые. Под изменяемостью (изменчивостью) последовательности понимается возможность добавлять или удалять элементы этой последовательности (т.е. изменять количество элементов последовательности).

4.2.1. Неизменяемые последовательности – строки

Строки (последовательности символов-букв и других знаков) могут состоять из символов английского и другого алфавита.

В Python строки и символы нужно заключать в кавычки. Элементы (символы) в строке нумеруются, начиная с нуля.

Числа могут быть преобразованы в строки с помощью функции *str()*. Например, *str(123)* даст строку «123». Для любого символа можно узнать его номер с помощью функции *ord()*. Получить символ по числовому коду можно с помощью *chr()*.

Таблица 3.3. Основные операции со строками

Операция	Описание
<i>len(s)</i>	Вычисляется длина строки.
<i>s1 + s2</i>	К концу строки <i>s1</i> присоединяется строка <i>s2</i> , в результате получается новая строка.
<i>s*n</i>	Дублирование строки.

$s[i]$	<p>Выбор из строки элемента с индексом i, (нумерация начинается с 0).</p> <div>$s = \text{'дерево'}, s[-2] \rightarrow \text{'в'};$</div>
$s[i:j:k]$	<p>Срез, содержащий символы строки s с номерами от i до j с шагом k (если k не указан, то символы идут подряд).</p> <div>$s = \text{'дерево'}, s[1:5:2] \rightarrow \text{'ee'};$</div>

Таблица 3.4. Основные методы над строками

Метод	Описание
$s.center(n)$	Строка s выравнивается справа и слева до ширины в n символов.
$s.ljust(n)$	Строка s выравнивается по левому краю (дополняется пробелами справа) в пространстве шириной n символов.
$s.rjust(n)$	Строка s выравнивается по правому краю (дополняется пробелами слева) в пространстве шириной n символов.
$s.count(s1[,i,j])$	<p>Определяется количество вхождений подстроки $s1$ в строку s. Результатом является число. Можно указать позицию начала поиска i и окончания поиска j.</p> <div> $s = \text{'abracadabra'}$ $s.count(\text{'ab'}) \rightarrow 2$ $s.count(\text{'ab'}, 1) \rightarrow 1$ </div>
$s.find(s1[,i,j])$	Определяется позиция первого вхождения подстроки $s1$ в строку s . Результатом является число. Необязательные аргументы i и j определяют начало и конец области поиска.

<i>s.rfind(s1[,i,j])</i>	<p>Определяется позиция последнего (считая слева) вхождения подстроки <i>s1</i> в строку <i>s</i>.</p> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <pre>s = 'abracadabra' s.rfind('br') → 8</pre> </div>
<i>s.strip()</i>	Создаётся копия строки, в которой удалены пробелы в начале и в конце.
<i>s.lstrip()</i>	Создаётся копия строки, в которой удалены пробелы в начале.
<i>s.rstrip()</i>	Создаётся копия строки, в которой удалены пробелы в конце.
<i>s.split()</i>	Разбиение строки по разделителю.
<i>s.join(list)</i>	Сборка строки из списка с разделителем <i>s</i> .
<i>s.replace(s1,s2[,n])</i>	Создаётся новая строка, в которой подстрока <i>s1</i> исходной строки заменяется на подстроку <i>s2</i> . Необязательный аргумент <i>n</i> указывает количество замен.
<i>s.capitalize()</i>	Создаётся новая строка, в которой первая буква исходной строки становится заглавной.
<i>s.swapcase()</i>	Создаётся новая строка, в которой прописные буквы исходной строки заменяются на строчные и наоборот.
<i>s.upper()</i>	Создаётся новая строка, в которой все буквы исходной строки становятся заглавными.
<i>s.lower()</i>	Создаётся новая строка, в которой все буквы исходной строки становятся строчными.
<i>s.title()</i>	Первую букву каждого слова переводит в верхний регистр, а все остальные в нижний.

Иногда при решении задач возникает необходимость в использовании экранированных последовательностей, при помощи которых можно получить некоторые эффекты, недостижимые иным образом.

Таблица 3.5. Экранированные последовательности

Последовательность	Описание
\\	Обратный слеш. Выводит: \
\'	Апостроф, или одиночная кавычка. Выводит : '
\"	Кавычка. Выводит : "
\a	Звук системного динамика.
\n	Новая строка.
\t	Табуляция.

2. Практическое задание «Строки»

1. Напишите программу с возможностью ввода новостного текста. Оформите вывод текста, придерживаясь следующего шаблона:

01.01.2016

«ПОПОЛНЕНИЕ»

Большая панда в зоопарке на западе Японии принесла потомство. При этом у неё вместо одного родилось сразу два детёныша. Близнецы появились на свет в парке «Вакаяма». С разницей в 3 часа. Каждый весит приблизительно по 180 граммов.

Количество слов в тексте: 37

*Примечание: для вывода текущей даты используйте модуль для работы с датой и временем **datetime**.*

2. Представьте себе, что вы находитесь в стране Зазеркалье. Напишите программу, которая перевернёт, введенные пользователем предложения.

4.2.2. Неизменяемые последовательности – кортежи

Кортеж в Python – это упорядоченный набор объектов, в который могут одновременно входить объекты разных типов (числа, строки и другие структуры, в том числе и кортежи).

```
# Создание кортежа при помощи литерала
k = (12, 'a', 36.6, 'kom')
k1 = () # пустой кортеж
# Создание кортежа с помощью встроенной функции tuple()
k = tuple()
```

С использованием допустимой в Python цепочки присваиваний можно элементам кортежа сразу сопоставить какие-нибудь переменные.

```
k = (x, s1, y, s2) = (12, 'a', 36.6, 'kom')
```

Кортежи могут получаться в результате работы функций Python, например, уже упоминавшаяся функция *divmod()* возвращает кортеж из двух элементов.

Кортежи могут использоваться для хранения характеристик каких-нибудь предметов, существ или явлений, если эти предметы, существа или явления характеризуются фиксированным набором свойств. Например, в виде кортежа можно записать фамилию студента и его оценки за семестр.

Для кортежей можно применять все те же операции, что и для списков, за исключением операций, пытающихся изменить кортеж.

!Важно понимать, что при сортировке имён объектов принято использовать определённый порядок: сначала числа по возрастанию, затем строки, начинающиеся на цифры в порядке их возрастания, затем строки, начинающиеся на прописные буквы в алфавитном порядке, а затем строки, начинающиеся на строчные буквы также в алфавитном порядке.

Зачем нужны кортежи? Кортеж защищен от изменений. Его удобно использовать в случаях, когда важно быть уверенным, что данные не изменялись в ходе выполнения программы. То есть

кортежи обеспечивают своего рода ограничение целостности, что может оказаться полезным в крупных программах.

4.2.3. Изменяемые последовательности – списки

Список в Python – это упорядоченный набор объектов, в список могут одновременно входить объекты разных типов (числа, строки и другие структуры, в частности, списки и кортежи).

```
# Создание списка при помощи литерала
lst = [12, 'a', 3.6, 'ком']
lst1 = [] # пустой список
# Создание списка с помощью встроенной функции list()
lst = list('список')
lst → ['с', 'н', 'и', 'с', 'о', 'к']
# Создание списка с помощью генераторов списков
c = [c * 2 for c in 'list']
c → ['ll', 'ii', 'ss', 'tt']
```

С использованием допустимой в Python цепочки присваиваний можно элементам списка сразу сопоставить какие-нибудь переменные.

```
lst = [x, s1, y, s2] = [12, 'a', 3.6, 'ком']
```

В отличие от кортежа, значения элементов списка можно изменять, добавлять элементы в список и удалять их.

! Списки являются очень полезными структурами данных в Python, и с использованием списков, их методов и операций с ними можно эффективно решать самые разнообразные задачи.

Кроме операций `len(lst)`, `lst1+lst2`, `lst*n`, `lst[i]`, `lst[i:j:k]`, `min(lst)`, `max(lst)` для списков также можно применить следующие операции.

Таблица 3.6. Основные операции со списками

Операция	Описание
<code>lst[i]=x</code>	Замена элемента списка с номером <i>i</i> на значение <i>x</i> . Если <i>x</i> является списком, то на место

	элемента списка будет вставлен список. При этом новый список не создаётся.
<i>del lst[i]</i>	Удаление из списка элемента с номером <i>i</i> . Новый список не создаётся.
<i>lst[i:j]=x</i>	Замена среза списка <i>lst</i> на элемент или список <i>x</i> (несколько элементов заменяются на <i>x</i>). <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <i>lst = [1, 2, 3, 'ком', 'дерево']</i> <i>lst[2:4] = 'дом'</i> <i>lst → [1, 2, 'д', 'о', 'м', 'дерево']</i> </div>
<i>del lst[i:j]</i>	Удаление элементов, входящих в указанный срез.

Списки в Python, как и строки, являются объектами, поэтому для списков существуют методы.

Таблица 3.7. Основные методы над списками

Метод	Описание
<i>lst.append(x)</i>	Добавление элемента <i>x</i> в конец списка <i>lst</i> . <i>x</i> не может быть списком. Создания нового списка не происходит.
<i>lst.extend(t)</i>	Добавление кортежа или списка <i>t</i> в конец списка <i>lst</i> (похоже на объединение списков, но создания нового списка не происходит).
<i>lst.count(x)</i>	Определение количества элементов, равных <i>x</i> , в списке <i>lst</i> . Результат является числом.
<i>lst.index(x, [start [, end]])</i>	Возвращает положение первого элемента от <i>start</i> до <i>end</i> со значением <i>x</i> .
<i>lst.remove(x)</i>	Удаление элемента <i>x</i> в списке <i>lst</i> в первой слева позиции.
<i>lst.pop(i)</i>	Удаление элемента с номером <i>i</i> из списка <i>lst</i> . При этом выдаётся значение этого

	элемента. Если номер не указан, удаляется последний элемент. Новый список не создаётся.
<i>lst.insert(i,x)</i>	Вставка элемента или списка <i>x</i> в позицию <i>i</i> списка <i>lst</i> . Если $i \geq 0$, вставка идёт в начало списка. Если $i > len(lst)$, вставка идёт в конец списка. Новый список не создаётся.
<i>lst.sort()</i>	Сортировка списка по возрастанию (в алфавитном порядке).
<i>lst.reverse()</i>	Сортировка списка в обратном порядке.

При помощи функции **zip()** можно получить список кортежей из элементов различных списков. Аргументами функции **zip()** являются два или более списков, а результатом – список кортежей.

```
# Демонстрация функции zip()
lst1 = [ 1 , 2 , 3 , 4 ]
lst2 = ['ком', 'дом', 'дерево']
lst = zip(lst1, lst2)
lst → [ (1, 'ком'), (2, 'дом'), (3, 'дерево') ]
```

Количество элементов в итоговом списке равно количеству элементов в самом коротком исходном списке. «Лишние» элементы других списков игнорируются.

Для списков и кортежей, состоящих только из чисел, возможно применение функции **sum()**, которая вычисляет сумму элементов списка (кортежа).

Метод **split()** делит строку по заданному символу-разделителю и создаёт список из фрагментов строки.

```
# Демонстрация метода split()
str = 'Ночь. Улица. Фонарь.'
lst = str.split(' _ ')
lst → ['Ночь.', 'Улица.', 'Фонарь.']
```

Метод **join()** формирует строку из элементов списка.

```
# Демонстрация метода join()
```

```
lst = [ '1', '2', '3' ]  
s = ' _ '.join(lst)  
s → '1 2 3'
```

3. Практическое задание «Списки»

1. Получите список из отрицательных чисел другого списка, стоящих на нечётных местах.

*Примечание: используйте модуль **random**, предназначенный для генерации случайных элементов.*

2. Определите количество прописных (больших) и строчных (малых) букв в списке.

*Примечание 1: список может быть определён заранее с помощью модуля **random** или введён пользователем на выбор.*

Примечание 2: в кодировках символы упорядочены, то есть 'a' < 'b'. Поэтому если очередной символ принадлежит диапазону от 'a' до 'z', значит это строчная буква; если диапазону от 'A' до 'Z' - то прописная.

4.2.4. Отображения – словари

Словари – единственный тип отображения в наборе базовых объектов Python – также относятся к классу изменяемых объектов: они могут изменяться непосредственно и в случае необходимости могут увеличиваться и уменьшаться в размерах подобно спискам.

Словари позволяют устанавливать связи (ассоциации) «ключ-значение» (например, «имя-адрес»), поэтому с их помощью создаются так называемые ассоциативные массивы, которые хранят данные в виде пар (ключ, значение).

```
# Создание словаря при помощи пустой пары фигурных скобок {}  
s = {}  
# Создание словаря при помощи функции dict()  
s = dict()  
# Создание словаря при помощи dict.fromkeys(seq[, value]) с
```

```

ключами из seq и значением value
s = dict.fromkeys(['a', 'b', 'c'], 'элемент')
s → {'c': 'элемент', 'b': 'элемент', 'a': 'элемент'}
# Создание словаря при помощи генераторов словарей
s = {a: a ** 2 for a in range(7)}
s → {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}

```

Таблица 3.8. Основные операции со словарем

Операция	Описание
<i>len(s)</i>	Возвращает число элементов в словаре.
<i>s[key]</i>	Возвращает элемент словаря с ключом <i>key</i> .
<i>s[key] = value</i>	Присваивает значение элементу словаря.
<i>del s[key]</i>	Удаляет элемент по ключу; вызывает <i>KeyError</i> , если ключа нет.
1) <i>key in s</i> , 2) <i>key not in s</i>	1) Возвращает <i>True</i> , если у словаря есть ключ <i>key</i> . 2) Возвращает <i>True</i> , если у словаря нет ключа <i>key</i> .
<i>s.get(key[, default])</i>	Возвращает элемент словаря с ключом <i>key</i> , если ключ есть в словаре, иначе <i>default</i> .
<i>s.items()</i>	Возвращает список пар (ключ, значение).
<i>s.keys()</i>	Возвращает список ключей словаря
<i>s.pop(key[, default])</i>	Удаляет ключ и возвращает значение. Если ключа нет, возвращает <i>default</i> (по умолчанию вызывает исключение).
<i>s.popitem()</i>	Удаляет и возвращает пару (ключ, значение). Если словарь пуст, то вызывает исключение <i>KeyError</i> .
<i>s.setdefault(key[, default])</i>	Если ключ <i>key</i> есть в словаре — возвращает его значение; если нет — создает и возвращает ключ со значением из <i>default</i> (по умолчанию <i>None</i>).
<i>s.update([other])</i>	Обновляет словарь, добавляя пары (ключ,

	значение) из <i>other</i> . Существующие ключи перезаписываются.
<i>s.values()</i>	Возвращает значения словаря.
<i>s.copy()</i>	Возвращает копию словаря.
<i>s.clear()</i>	Удаляет все элементы из словаря.

4. Практическое задание «Словари»

Создайте простой русско-английский словарь со следующими возможностями:

1. Словарь отображает перевод слова, введенного пользователем.
2. Если слово отсутствует, то пользователь может добавить его в словарь.

4.3. Другие базовые типы

Помимо базовых типов данных, которые были рассмотрены, существуют и другие, которые могут считаться базовыми.

Например, множества, совсем недавно появившиеся в языке, – которые не являются ни последовательностями, ни отображениями.

Также в языке Python имеется логический тип данных (представленный предопределенными объектами *True* и *False*), а кроме того, давно уже существует специальный объект *None*.

4.3.1. Множества

Множества – это неупорядоченные коллекции уникальных и неизменяемых объектов.

Множества создаются встроенной функцией *set()* или с помощью новых синтаксических конструкций определения литералов и генераторов множеств, появившихся в версии 3.0, и поддерживают типичные математические операции над множествами.

```
# Создание множеств при помощи функции set()
m = set(['a', 'b', 'c', 'c', 'a'])
```

```

m → {'c', 'b', 'a'}
# Создание множеств при помощи генератора множеств
m = {i ** 2 for i in range(10)}
m → {0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

```

Таблица 3.9. Основные операции над множествами

Операция	Описание
<i>len(m)</i>	Возвращает число элементов в множестве.
<i>x in m</i>	Определяет принадлежность <i>x</i> множеству <i>m</i> .
<i>m.isdisjoint(m1)</i>	Выводит <i>True</i> , если <i>m</i> и <i>m1</i> не имеют общих элементов.
<i>m.issubset(m1)</i>	Определяет, принадлежат ли все элементы <i>m</i> множеству <i>m1</i> .
<i>m.issuperset(m1)</i>	Определяет, принадлежат ли все элементы <i>m1</i> множеству <i>m</i> .
<i>m.union(m1, ...)</i> <i>m & m1</i>	Объединение множеств.
<i>m.intersection(m1, ...)</i> <i>m m1</i>	Пересечение множеств.
<i>m.difference(m1, ...)</i>	Вычитание множеств. Возвращает множество из всех элементов <i>m</i> , не принадлежащие <i>m1</i> .
<i>m.symmetric_difference(m1)</i>	Возвращает множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
<i>m.copy()</i>	Копирует множество.

```

# Операции над множествами
m = set('spam')
m1 = {'h', 'a', 'm'} # В 3.0 можно определять литералы множеств
m, m1 → ({'a', 'p', 's', 'm'}, {'a', 'h', 'm'})

```

$m \& m1 \rightarrow \{'a', 'm'\}$
$m / m1 \rightarrow \{'a', 'p', 's', 'h', 'm'\}$
$m - m1 \rightarrow \{'p', 's'\}$

Таблица 3.10. Основные методы, изменяющие множество

Операция	Описание
$m.update(m1, ...)$	Объединение.
$m.intersection_update(m1)$	Пересечение.
$m.difference_update(m1)$	Вычитание.
$m.symmetric_difference_update(m1)$	Возвращает множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих.
$m.add(elem)$	Добавляет элемент в множество.
$m.remove(elem)$	Удаляет элемент из множества; вызывает <i>KeyError</i> , если нет такого элемента.
$m.discard(elem)$	Удаляет элемент, если он находится в множестве.
$m.pop()$	Удаляет и возвращает произвольный элемент из множества; вызывает <i>KeyError</i> , если множество пустое.
$m.clear()$	Очистка множества.

4.3.2. Файлы

Кроме того в языке Python имеется логический тип данных (представленный предопределенными объектами *True* и *False*) и специальный объект *None*, который обычно используют для инициализации переменных и объектов.

Объекты-файлы – это основной интерфейс между программным кодом на языке Python и внешними файлами на компьютере. Файлы являются одним из базовых типов, но они представляют собой нечто необычное, поскольку для файлов отсутствует возможность создания объектов в виде литералов. Вместо этого, чтобы создать объект

файла, необходимо вызвать встроенную функцию *open()*, передав ей имя внешнего файла и строку режима доступа к файлу.

Таблица 3.11. Виды режимов доступа к файлу

Режим доступа	Описание
<i>r</i>	Открытие файла на чтение.
<i>w</i>	Открытие на запись, содержимое файла удаляется. Если файла не существует, то создается новый.
<i>a</i>	Открытие на добавление данных в файл.
<i>b</i>	Открытие бинарных файлов
<i>t</i>	Открытие в текстовом режиме.
<i>+</i>	Открытие на чтение и запись.

Режимы могут быть объединены, то есть, к примеру, «*rb*» — чтение в двоичном режиме. По умолчанию режим равен «*rt*».

И последний аргумент, *encoding*, нужен только в текстовом режиме чтения файла. Этот аргумент задает кодировку.

!В Python 3.0 проводится очень четкая грань между текстовыми и двоичными данными. Python 3 использует понятия текста и (бинарных) данных вместо строк Unicode и 8-битных строк. Весь текст – Unicode. Однако кодированные Unicode строки представлены в виде двоичных данных. Тип, используемый для хранения текста является *str*, тип, используемый для хранения данных - *bytes*. Самое большое различие с Python 2.x является то, что любая попытка комбинировать текст и данные в Python 3.0 поднимает *TypeError*.

В Python 3.0 можно больше не использовать литерал *u'...'* для текста Unicode. Тем не менее, необходимо использовать литерал *b'...'* для бинарных данных. Следует использовать *str.encode()*, чтобы перейти от *str* к *bytes* и *bytes.decode()*, чтобы перейти от *bytes* к *str*. Также можно применить *bytes(s, encoding=...)* и *str(b, encoding=...)*.

Таблица 3.12. Основные методы работы с файлами

Метод	Описание
<i>f.read()</i>	Чтение файла целиком в строку.
<i>f.read(N)</i>	Чтение следующих N символов (или байтов) в строку.
<i>f.readline()</i>	Чтение следующей строки (включая символ конца строки) в строку.
<i>f.readlines()</i>	Чтение файла целиком в список строк (включая символ конца строки).
<i>f.write(string)</i>	Запись строки символов (или байтов) в файл.
<i>f.writelines(list)</i>	Запись всех строк из списка в файл.
<i>f.close()</i>	Закрывание файла вручную (выполняется по окончании работы с файлом).
<i>f.seek(N)</i>	Изменяет текущую позицию в файле для следующей операции, смещая ее на Nбайтов от начала файла.
<i>for line in open('data'):</i> ...	Итерации по файлу, построчное чтение.
<i>open('f.txt', encoding='latin-1')</i>	Файлы с текстом Юникода в Python 3.0.
<i>open('f.bin', 'rb')</i>	Чтение двоичных файлов.
# Демонстрация работы с файлом <i>f = open('example.txt', 'w')</i> <i>f.write('Hello\n')</i> <i>f.close()</i>	

Для загрузки файла по частям обычно используется либо цикл *while*, завершающийся инструкцией *break* по достижении конца файла, либо *for*.

Таблица 3.13. Циклы *while* и *for* для загрузки файла

<i>while</i>	<i>for</i>
Чтение из файла по количеству символов	

<pre> file = open('example.txt') while True: char = file.read(x) if not char: break print(char) </pre>	<pre> for char in open('example.txt').read(): print(char) </pre> <p>!Цикл <i>for</i> выполняет обработку каждого символа, но загрузка производится однократно.</p>
Чтение строками	
<pre> file = open('example.txt') while True: str = file.readline() if not str: break print(str, end="") </pre>	<pre> for str in open('test.txt').readlines(): print(str, end=' ') </pre> <p>или</p> <pre> for str in open('test.txt'): print(str, end=' ') </pre> <p>! Метод файлов <i>readlines()</i> загружает файл целиком в список строк, тогда как при использовании итератора файла в каждой итерации загружается только одна строка</p>

5. Итоговое практическое задание по теме «Типы данных»

Шифрование сообщения при помощи кода Цезаря

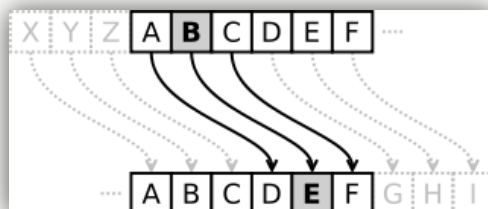
1. Закодировать любое текстовое сообщение с помощью кода Цезаря.

Код Цезаря – один из древнейших шифров. При шифровании каждый символ заменяется другим, отстоящим от него в алфавите на фиксированное число позиций. Шифр Цезаря можно классифицировать как шифр подстановки, при более узкой классификации – шифр простой замены. Шифрование с использованием ключа $k = 3$. Буква «С» «сдвигается» на три буквы

вперед и становится буквой «Ф». Твердый знак, перемещённый на три буквы вперед, становится буквой «э», и так далее:

Оригинальный текст: Съешь же ещё этих мягких французских булок, да выпей чаю.

Шифрованный текст: Фэзыя йз зы ахли пвёшли чугрицкфнли дцосн, жг еютзм ъгб.



2. Ключ записать в текстовый файл key.txt.
3. Попросить соседа ввести ключ для дешифрования сообщения. Если введенный записанный в файле ключи совпадают, то вывести сообщение об успехе и декодированное сообщение.
4. При выходе вывести сообщение о количестве использованных попыток.

Вопросы для повторения

1. Перечислите все типы данных, какие вы запомнили. Расскажите коротко о них. Приведите примеры.
 2. Какие два типа последовательностей вы знаете? Какая между ними разница?
 3. Какие неизменяемые последовательности вы знаете?
 4. Для чего нужны кортежи?
 5. Могут ли в список одновременно входить объекты разных типов? Придумайте пример для работы со списком.
 6. Чем список отличается от кортежа?
 7. Какое значение используется по умолчанию в аргументе режима обработки файла в функции `open()`?
-

Глава 5. Функции в Python

Все примеры и задания, которые были представлены ранее, состояли из длинных цепочек команд.

Но при написании сложных больших программ, такая организация кода становится неоптимальной и неудобной.

Использование функций - это способ разбить большую программу на более мелкие части, управлять которыми по отдельности легче, чем целой программой.

Функция является независимой частью кода, связывающая один или несколько входных параметров с одним или несколькими выходными параметрами.

Что даёт использование функций?

- обращение к функциям в процессе выполнения программы может быть многократным;
- сокращается объем исходного кода;
- затрачивается меньше времени и труда;
- создаётся **абстракция**, которая позволяет не отвлекаться от главной задачи и не думать о том, как реализована та или иная функция.

В Python имеется много встроенных функций: `print()`, `min()`, `sqrt()`, `input()`. Но вы также можете создать свои собственные функции.

```
# Функция вывода приветствия на экран
def message(name):
    """Вывести на экран приветствие для пользователя"""
    print('Здравствуйте, %s!' % name)
    name = input('Как вас зовут?')
    message(name)
```

Правила написания функций:

1. `def` – это команда, которая сообщает интерпретатору, что следующий блок кода является функцией.
2. `message` – это имя функции, может быть почти любым, не зарезервированным в Python словом. Хорошим тоном является

давать осмысленные имена функциям, параметрам и переменным.

3. После имени функции в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми.
4. Далее идет двоеточие, обозначающее окончание заголовка функции.
5. Тело функции содержит блок выражений с отступом.
6. В конце тела функции присутствует инструкция *return* (может и не быть), которая возвращает одно или несколько значений в основной блок программы.

! Переменные и параметры не доступны вне функции. Данная техника называется **инкапсуляцией**. Она помогает сохранять независимость отдельных фрагментов кода. Данный вид изоляции очень полезен, разработчику не нужно следить за значениями переменных, созданных внутри функции. Чем больше программа, тем значительнее выгода от этого.

Инкапсуляция тесно связана с абстракцией и является ее важнейшим элементом. Абстракция позволяет не заботиться о деталях, а инкапсуляция попросту прячет детали.

Также при составлении функций важно не забывать их документировать. Это даёт ясность, что делает данный блок программы. Особенно это необходимо, если вы работаете в команде, документирование повысит читаемость и ускорит понимание вашего кода. В приведённом примере строка `"""Вывести на экран приветствие для пользователя"""` документирует нашу функцию. Для документации также необходимо придерживаться отступа.

6. Практическое задание «Функции»

Парсинг веб-ресурсов

Парсинг – это синтаксический анализ сайтов, который автоматически производится парсером – специальной программой или скриптом. Характер парсинга определяется заданием

получить определенную информацию со страниц сайта, параметры анализа заранее задаются.

Три стадии парсинга:

1. сбор информации;
2. анализ данных, обработка и преобразование в нужный формат.
3. вывод данных.

Наиболее часто парсинг опирается на систему регулярных выражений.

1. Создайте программу, которая получит данные первых двух страниц из поисковой системы по какому-либо запросу.

Примечание 1: пользователю должна быть предоставлена возможность ввода поискового запроса.

Примечание 2: допускается использование сторонних библиотек для парсинга.

2. Создайте список из полученных данных, который содержит:

- номер страницы;
- ссылку;
- название.

Результат должен выглядеть следующий образом:

```
list = [{ page: '', url: '', title: '' }, { page: '', url: '', title: '' }, ...]
```

Примечание: программа должна содержать столько функций, сколько это возможно.

Вопросы для повторения

1. Какие преимущества вы получите при использовании функций?
 2. Расскажите о правилах написания функций.
 3. Что значит абстракция?
 4. Что даёт инкапсуляция?
-

Глава 6. Объектно-ориентированное программирование

До сих пор наши программы состояли из функций, т.е. блоков выражений, которые манипулируют данными. Это называется процедурно-ориентированным стилем программирования. Существует и другой способ организации программ: объединять данные и функционал внутри некоторого объекта. Это называется объектно-ориентированной парадигмой программирования. В большинстве случаев можно ограничиться процедурным программированием, а при написании большой программы или если решение конкретной задачи того требует, можно переходить к техникам объектно-ориентированного программирования.

Принципы ООП:

- **Инкапсуляция** (скрывает внутренние подробности работы объекта от окружающего мира).
- **Полиморфизм** (позволяет работать с различными типами объектов так, что не нужно задумываться о том, к какому типу они принадлежат).
- **Наследование** (можно создавать специализированные классы на основе базовых, что позволяет избежать написания повторного кода).

Два основных аспекта объектно-ориентированного программирования – классы и объекты. **Класс** создаёт новый тип, а **объекты** являются экземплярами класса. Аналогично, когда мы говорим о переменных типа *int*, это означает, что переменные, которые хранят целочисленные значения, являются экземплярами (объектами) класса *int*.

У программного объекта есть несколько характеристик, которые на языке ООП называются **атрибутами**, и способов поведения, которые принято называть **методами**.

6.1. Создание класса

Класс создаётся ключевым словом *class*. Новый класс можно создать на основе от *object* или любого ранее объявленного класса.

При наследовании разрешение имен атрибутов работает сверху вниз: если атрибут не найден в текущем классе, поиск продолжается в базовом классе, и так далее по рекурсии. Производные классы могут переопределить методы базовых классов. Базовых классов может быть несколько.

Имя класса лучше начинать с прописной буквы, как этого требует общепринятая практика.

После создания класса, вызвать его (создать объект-экземпляр) можно всего одной строкой кода:

```
p = Person()
```

! За именем класса обязательно следуют скобки.

6.2. Методы в классе

В составе класса объявляется метод. По своей структуре метод очень схож с функцией, но он имеет дополнительный параметр, по договоренности называемый *self*. Данный параметр позволяет методу сослаться на объект класса.

```
# Класс-приветствие
class Person(object):
    """Метод, создающий приветствие"""
    def hello(self):
        print('Привет! Как дела?')
```

У нового объекта есть метод *hello()*. Этот метод, как и любой другой, - не что иное, как функция, принадлежащая объекту. Вызвать этот метод можно обычным образом, с помощью точечной нотации:

```
p = Person()
p.hello()
```

Данный пример можно вызвать при помощи одной строки:

```
Person().hello()
```

При вызове метода, вы увидите печатный текст: «Привет! Как дела?».

6.3. Конструктор класса — метод `__init__`

Существует много методов, играющих специальную роль в классах Python.

Метод `__init__` запускается, как только объект класса реализуется, то есть вызывать его не нужно. Этот метод полезен для осуществления разного рода инициализации, необходимой для данного объекта. Обратите внимание на двойные подчёркивания в начале и в конце имени. Если обозначить метод как `__init__`, Python сочтет, что это метод-конструктор.

! «`__*__`» — конструкция зарезервированных классов идентификаторов. Приложения не должны определять дополнительные имена используя эту форму.

Первым параметром, как и у любого другого метода, у `__init__` является *self*, на место которого подставляется объект в момент его создания.

Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса.

```
# Класс-приветствие с инициализацией
class Person(object):
    def __init__(self, name):
        self.name = name
    def hello(self):
        print('Добро пожаловать,', self.name)
# основная часть
p = Person('Иван')
p.hello()
```

В программе у объекта появляется атрибут *name*, который инициализируется при создании объекта и значение должно

передаваться в скобках при вызове класса. Во избежание ошибок лучше в теле метода использовать значения по умолчанию: `def __init__(self, name='Аноним')`:

6.4. Атрибуты

Состояние объекта характеризуется текущим значением его атрибутов. Для лучшего понимания рассмотрим объект «человек», у которого есть несколько атрибутов. Возраст может характеризоваться числом. Состояние «голоден» или «сыт» — логическим значением.

Как правило, атрибуты объекта доступны для чтения и изменения не только внутри класса, но и вне его.

Чтобы получить доступ к атрибутам класса в Python следует после объекта поставить точку и написать имя переменной или метода, которые вы хотите использовать: `object.attribute`.

! Обычно стараются избежать прямого доступа к атрибутам объекта вне объявления класса.

6.4.1. Доступ к атрибутам

По умолчанию атрибут является открытым (*public*). Ноего можно сделать приватным (*private*), то есть недоступным снаружи. Для этого слева нужно поставить два символа подчеркивания. Например, `self.__name = name`. При попытке вызвать такой атрибут извне, программа выдаст исключение.

! Закрытыми могут быть и методы. Закрытый метод можно создать тем же самым несложным способом, что и закрытый атрибут: добавить в начало его имени два символа подчеркивания.

! При реализации класса придерживайтесь следующих правил: создавайте методы, существование которых сделает ненужным прямой доступ из клиентского кода к атрибутам объекта, закрывайте лишь те атрибуты и методы, которые обслуживают внутренние операции объекта, избегайте непосредственного чтения и изменения

значений атрибутов и никогда не пытайтесь прямо обращаться к закрытым атрибутам и методам объекта.

Иногда вместо того, чтобы закрыть доступ к атрибуту, целесообразно только ограничить его. В некоторых случаях полезно, например, иметь атрибут, который можно будет прочесть из клиентского кода, но не изменить. В Python для этого есть кое-какие инструменты, в первую очередь **свойства** – объект с методами, которые позволяют косвенно обращаться к атрибутам и зачастую в чем-либо ограничивают такой косвенный доступ.

```
# Контроль атрибутов через декоратор @property
class Person(object):
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
# Основная часть
p = Person('Иван')
print('Здравствуйте, %s!' % p.name)
```

После реализации данного кода на экране появится приветствие «Здравствуйте, Иван!».

При попытке изменить свойство объекта с помощью команды `p.name = Oleg` программа выдаст исключение `AttributeError: can't set attribute`.

Чтобы создать свойство, я пишу метод, который возвращает интересующее меня значение (в данном случае `__name`), и «обворачиваю» метод декоратором `@property`. Свойство одноименно методу; в нашем примере оно называется `name`. Теперь через свойство `name` любого объекта класса `Person` можно узнавать значение закрытого атрибута `__name` этого объекта - неважно, внутри или вне объявления класса. Для этого применяется уже знакомая вам точечная нотация.

! Если запрашиваемый атрибут – закрытый, то принято называть свойство так же, как этот атрибут, но без начальных символов подчеркивания.

Создавая свойство, мы «приоткрываем» закрытый атрибут, делая его доступным для чтения, но не обязательно только для чтения. Можно разрешить запись в закрытый атрибут и даже установить ограничения на запись. Через свойство *name* закрытый атрибут *__name* становится доступным для записи с некоторыми оговорками.

```
# Демонстрация изменения свойств
class Person(object):
    def __init__(self, name):
        self.__name = name
    @property
    def name(self):
        return self.__name
    @name.setter
    def name(self, new_name):
        if new_name == "":
            print("Имя не может быть пустой строкой!")
        else:
            self.__name = new_name
            print('Имя изменено. Здравствуйте, %s!' % self.__name)
p = Person('Ivan')
print('Здравствуйте, %s' % p.name)
p.name = 'Oleg'
```

Данный код начинается с декоратора *@name.setter*. При обращении к атрибуту *setter* свойства *name*, метод, приводимый далее, устанавливает новое значение свойства *name*. Для создания нового декоратора для изменения значения свойства, следуйте следующему образцу: *@имя_свойства.setter*.

За декоратором следует метод *name*, который вызывается из клиентского кода при попытке присвоить новое значение скрытому атрибуту через свойство. Метод-сеттер должен носить то же имя, что и свойство.

При вызове метода параметру *new_name* передается новое имя. Если это будет пустая строка, то атрибут *__name* не изменится и программа известит пользователя о том, что попытка закончилась неудачей.

! Если вы хотите удалить значение скрытого атрибута, можно воспользоваться следующей конструкцией декоратора: *@имя_свойства.deleter*.

6.4.2. Атрибуты класса

Атрибуты позволяют присвоить уникальные значения разным объектам одного и того же класса. Но бывает и такая информация, которая относится не к индивидуальным объектам, а ко всему классу. То есть существует два типа: **атрибуты класса** и атрибуты объекта, которые различаются в зависимости от того, принадлежат ли они классу или объекту соответственно.

```
# Демонстрация атрибутов класса и статических методов
class Person(object):
    """Попорядку расчитайся!"""
    total = 0
    @staticmethod
    def status():
        print("\nВсего людей:", Person.total)
    def __init__(self, name):
        self.name = name
        print(self.name)
        Person.total += 1
    print("\nСчитаю... ")
    critl = Person("Первый")
```

```
crit2 = Person("Второй")
crit3 = Person("Третий")
Person.status()
input("\n\nНажмите Enter, чтобы выйти.")
```

Вы видите, что мы обращаемся к атрибуту класса как *Person.total*, а не *self.total*. К атрибуту же объекта *name* мы обращаемся при помощи обозначения *self.name*. Помните об этой простой разнице между переменными класса и объекта.

! Также имейте в виду, что переменная объекта с тем же именем, что и переменная класса, сделает недоступной переменную класса!

! В Python также существуют методы, связанные с целым классом. Они называются статическими и в силу своих свойств часто применяются вместе с атрибутами класса. Для создания статического метода (только «новые» классы могут иметь статические методы) используется декоратор *@staticmethod*.

Заключение об ООП

В Python можно выделить следующие особенности, связанные с объектно-ориентированным программированием:

1. Любое данное (значение) — это объект. Число, строка, список, массив — все является объектом. Бывают объекты встроенных классов, а бывают объекты пользовательских классов (тех, что создает программист).
2. Основные свойства ООП — полиморфизм, наследование, инкапсуляция. Полиморфизм позволяет нам работать с различными типами объектов так, что нам не нужно задумываться о том, к какому типу они принадлежат. Объекты могут скрывать (инкапсулировать) свое внутреннее состояние. Класс может быть производным от одного или нескольких классов. Производный класс наследует все методы базового класса. Базовых классов может быть несколько.

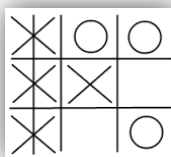
3. Объект состоит из атрибутов и методов. Атрибут — это переменная, метод — это функция. Отличия метода от функции в том, что у него есть первый параметр — *self*.
4. Инкапсуляции в Python не уделяется особого внимания. В других языках программирования обычно нельзя получить напрямую доступ к свойству, описанному в классе. Для его изменения может быть предусмотрен специальный метод. В Python же это легко сделать, просто обратившись к свойству класса из вне. Несмотря на это в Python все-таки предусмотрены специальные способы ограничения доступа к переменным в классе.

7. Практическое задание «ООП»

Игра «Крестики-нолики»

Крестики-нолики — логическая игра между двумя противниками на квадратном поле 3 на 3 клетки. Один из игроков играет «крестиками», второй — «ноликами».

В классическом варианте игроки по очереди ставят на свободные клетки поля 3x3 знаки (один всегда крестики, другой всегда нолики). Первый, выстроивший в ряд 3 своих фигуры по вертикали, горизонтали или диагонали, выигрывает. Первый ход делает игрок, ставящий крестики.



1. Создайте игру «Крестики-нолики», используя ООП.

Примечание: не забудьте добавить визуализацию ходов игроков.

2. Добавьте возможность просмотра числа попыток и выигрышей в игре.
3. Усложните игру: предложите пользователю самому выбрать размерность сетки в начале игры.

4. *Доп.* Добавьте возможность сохранения текущей игры в любой момент игры и восстановления состояния игры из файла.

Вопросы для повторения

1. Расскажите об особенностях объектно-ориентированного программирования.
 2. Перечислите принципы ООП.
 3. Как создаётся класс?
 4. Объясните, что такое метод. Чем он отличается от функции?
 5. Для чего используют метод `__init__`? Нужно ли его вызывать?
 6. Расскажите всё, что знаете об атрибутах.
 7. Какое различие между атрибутом класса и атрибутом объекта?
 8. Почему так важно перенести обработку атрибутов в методы, а не выполнять её за пределами класса?
-
-

Глава 7. Веб-фреймворк Django

7.1. Веб-фреймворки

Всем разработчикам знакомо ощущение, что им катастрофически не хватает времени. Заказчикам важно, чтобы проект был не только качественным и производительным, но и быстро написан.

Оптимизировать процесс написания проектов, увеличить их производительность, сократить время, затрачиваемое на типичные задачи, становится возможным благодаря разнообразным фреймворкам.

Веб-фреймворки - это группы готовых веб-компонентов и моделей, которые облегчают веб-программирование и делают его более организованным.

7.2. О Django

Одним из самых популярных фреймворков на сегодняшний день является веб-фреймворк Django, написанный на языке программирования Python.

Django был написан группой разработчиков из Лоуренса, штат Канзас, США в 2003 году. Первая версия была выпущена в июле 2005 года. Его название связано с именем джазовского гитариста Джанго Рейнхардта. С каждым годом Django совершенствуется и затачивается новыми возможностями, избавляясь от своих недостатков.

Веб-сайты на Django строятся из одного или нескольких приложений, которые рекомендуется делать отчуждаемыми и подключаемыми. Это одно из существенных архитектурных отличий этого фреймворка от некоторых других (например, Ruby on Rails). Один из основных принципов фреймворка – DRY (Don't repeat yourself).

Одним из главных достоинств Django является слабая связанность его компонентов. К примеру, разработчик может

изменить URL страницы, даже не редактируя html-код. Верстальщик, дизайнер и программист могут работать независимо друг от друга и вносить поправки, не вмешиваясь в чужой код.

Для работы с базой данных Django использует собственный ORM, в котором модель данных описывается классами Python, и по ней генерируется схема базы данных.

Django взаимодействует с разными СУБД: MySQL, PostgreSQL, Oracle и даже простая SQLite, для которой не требуется отдельной установки.

Также к преимуществам Django можно отнести: собственный веб-сервер для разработки, который ускоряет процесс разработки на Python, встроенный интерфейс администратора, расширяемую систему шаблонов с тегами и наследованием; систему кеширования, интернационализацию, встроенную авторизацию и аутентификацию, библиотеку для работы с формами (наследование, построение форм по существующей модели БД) и другие.

Если у вас имеется опыт веб-программирования, далее, при построении своего веб-сайта, вы поймете, почему Django-программисты предпочитают данный веб-фреймворк, и насколько упрощается процесс разработки при его использовании.

7.3. Архитектура Django

Django построен на архитектуре MVC. Архитектура MVC представляет собой:

- Модель (Model). Модель предназначена для работы с данными, которая взаимодействует с базой данных.
- Представление (View). Представление отвечает за то, как эти данные будут выглядеть.
- Контроллер (Controller). Контроллер является посредником между моделью, представлением и посетителями сайта.

Говоря простыми словами, MVC – это такой способ разработки программного обеспечения, при котором код определения и доступа к данным (модель) отделен от логики взаимодействия с приложением

(контроллер), которая, в свою очередь, отделена от пользовательского интерфейса (представление).

Главным плюсом такого подхода является то, что компоненты слабо связаны между собой. Каждый компонент имеет единственное назначение, поэтому его можно изменять независимо от остальных компонентов.

Django называют MTV-ориентированной средой разработки (Модель-Шаблон-Представление) поскольку здесь View выполняет функцию контроллера, а Template – представления.

1. Модель (Model). Модель является важнейшей составляющей приложения, при помощи которой происходит обращение к данным при запросах. Любая модель является стандартным классом Python.

2. Представление (View). На слой представления возложена задача обеспечения логики получения доступа к моделям и применения соответствующего шаблона. Представление является своеобразным мостом между моделями и шаблонами.

3. Шаблон (Template). Этот слой является формой представления данных. Шаблон имеет свой собственный простой метаязык.

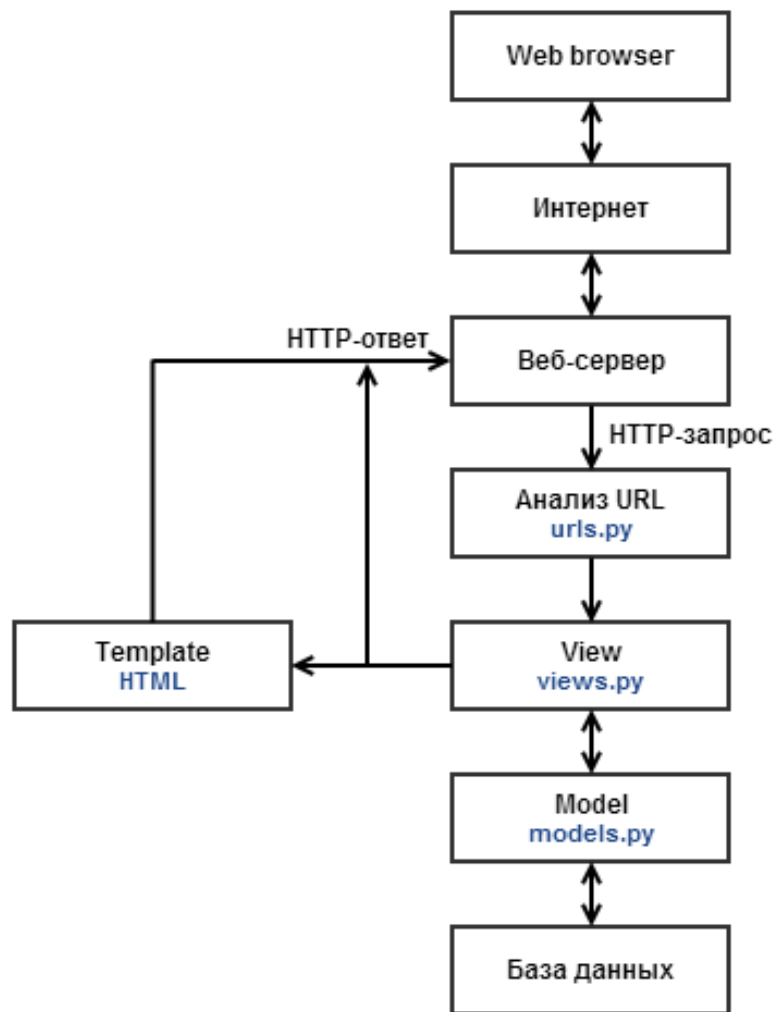


Рис.33. Архитектура Django

При запросе к странице Django-проекта, используется следующий алгоритм.

6. Django определяет какой из корневых модулей URLconf использовать. Обычно, это значение настройки *ROOT_URLCONF*.
7. Django загружает модуль конфигурации URL и ищет переменную *urlpatterns*.
8. Django перебирает каждый URL-шаблон по порядку, и останавливается при первом совпадении с запрошенным URL-адресом.
9. Если одно из регулярных выражений соответствует URL, Django импортирует и вызывает соответствующее представление.

10. Если ни одно регулярное выражение не соответствует, или возникла ошибка на любом из этапов, Django вызывает соответствующее исключение.

7.4. Краткое руководство по установке Django

Будучи веб-фреймворком Python, Django требует Python. Подходит любая версия Python 2.7, 3.2, 3.3 или 3.4. Эти версии Python содержат базу данных SQLite, поэтому вам не обязательно устанавливать базу данных для ознакомления с Django.

Последнюю версию Python можно найти на <http://www.python.org/download/>, или установить его с помощью пакетного менеджера вашей операционной системы.

Варианты по установке Django:

- установить версию Django предоставленную дистрибутивом вашей операционной системой;
- установить официальный релиз. Это самый лучший вариант для пользователей, которые хотят использовать последнюю стабильную версию;
- установить текущую разрабатываемую версию. Этот вариант для тех, кто хочет использовать самые последние возможности и не боится использовать новый код.

Удалите старые версии перед установкой Django. В противном случае вы можете столкнуться с невозможностью установки новой версии.

7.4.1. Установка официальной версии с помощью pip

Данный способ установки является рекомендуемым.

1. Установите **pip**. Если ваш дистрибутив уже включает установленную версию pip, возможно, вам потребуется обновить её до более свежей. (Если версия устарела, вы сразу поймёте это, поскольку установка не будет работать.)

2. Если вы используете Unix-подобные системы, введите команду *sudo pip install Django* в терминале. При использовании Windows запустите командную оболочку с правами администратора и впишите *pip install Django*. Эта команда установит Django в системную директорию *site-packages*.

7.4.2. Установка официальной версии вручную

1. Скачайте последнюю стабильную версию со страницы загрузки.
2. Распакуйте загруженный файл.
3. Перейдите в каталог.
4. Если вы используете Linux, Mac OS X или иные Unix-подобные системы, введите команду *sudo python setup.py install* в терминале. Если у вас установлена Windows, запустите командную оболочку с правами администратора и впишите *python setup.py install*. Эта команда установит Django в системную директорию *site-packages*.

! Разработка на Unix-подобных системах является более предпочтительным.

Чтобы проверить что Django доступен для Python, выполните *python* в терминале. Теперь в консоли Python выполните импорт Django:

```
import django
print(django.get_version())
```

Если Django установлен, то терминал вам выдаст текущую версию фреймворка.

7.4.3. Взаимодействие Django с базами данных

Django поддерживает много различных СУБД. В частности к ним относятся PostgreSQL, MySQL, Oracle и SQLite.

Если вы заняты разработкой простого проекта или чего-то, что вы не планируете развернуть в производственной среде, SQLite в целом является наипростейшим вариантом, поскольку он вообще не

требует установки отдельного веб-сервера. Однако, SQLite очень отличается от других баз данных, поэтому, если вы работаете над чем-то существенным, рекомендуется выбрать ту же базу данных, которая будет использована на «боевом» сервере.

В дополнение к установке необходимой БД вы также должны убедиться, что вами выбран и установлен соответствующий модуль Python.

При использовании PostgreSQL, вам понадобится пакет **postgresql-psycopg2**.

Пользователям Windows следует проверить наличие неофициальной скомпилированной для Windows версии.

При использовании MySQL, вам понадобится пакет **MySQL-python** версии 1.2.1p2 или выше.

Если вы используете SQLite, вам следует прочитать про особенности использования SQLite.

При использовании Oracle, вам понадобится копия **cx_Oracle**.

При использовании неофициальных бэкендов от сторонних разработчиков необходимо ознакомиться с документацией о дополнительных требованиях.

Если вы планируете использовать команду *manage.py migrate* для автоматической генерации таблиц БД для ваших моделей (после первого запуска Django и создания проекта), вам нужно убедиться, что Django имеет разрешение на создание и изменение таблиц БД; если же вы хотите создавать таблицы вручную, вам надо просто предоставить Django разрешения *SELECT*, *INSERT*, *UPDATE* и *DELETE*.

7.5. Создание проекта на Django

Самый лучший способ обучиться программированию – это начать программировать.

Мы будем создавать не просто сайт-визитку, а персональный веб-сайт с базой данных и системой аутентификации.

Для того чтобы создать проект, перейдите в каталог, где вы хотите хранить код, и выполните следующую команду:

```
django-admin.py startproject mysite
```

Структура проекта на Django выглядит следующим образом:

```
mysite/  
  manage.py  
  mysite/  
    __init__.py  
    settings.py  
    urls.py  
    wsgi.py
```

Рассмотрим эти файлы.

- Внешний каталог ***mysite/*** – это просто контейнер для вашего проекта. Его название никак не используется Django, и вы можете переименовать его во что угодно.
- ***manage.py***: Скрипт, который позволяет вам взаимодействовать с проектом Django.
- Внутренний каталог ***mysite/*** - это пакет Python вашего проекта. Его название – это название пакета Python, которое вы будете использовать для импорта чего-либо из проекта (например, ***mysite.urls***).
- ***mysite/__init__.py***: Пустой файл, который указывает Python, что текущий каталог является пакетом Python.
- ***mysite/settings.py***: Настройки/конфигурация проекта.
- ***mysite/urls.py***: Конфигурация URL-ов для вашего проекта Django.
- ***mysite/wsgi.py***: Точка входа вашего проекта для WSGI-совместимых веб-серверов.

7.5.1. Настройка базы данных

Откроем файл настроек *settings.py*.

По умолчанию используется SQLite. Это хороший вариант, если вы новичок в базах данных, или хотите попробовать Django. SQLite включен в Python, вам не нужно устанавливать что либо еще.

! Если вы используете PostgreSQL или MySQL, убедитесь, что вы создали базу данных. Если вы используете SQLite, вам ничего не нужно создавать самостоятельно - файл базы данных будет создан автоматически при необходимости.

В наших проектах мы будем использовать PostgreSQL для работы с большими объемами данных.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql_psycopg2',
        'NAME': 'mybd',
        'USER': 'username',
        'PASSWORD': 'password',
        'HOST': 'localhost',
        'PORT': '',
    }
}
```

Рассмотрим подробнее поля.

В поле **ENGINE** вы указываете, какой бэкэнд будете использовать. Это могут быть:

- `'django.db.backends.sqlite3'`;
- `'django.db.backends.postgresql_psycopg2'`;
- `'django.db.backends.mysql'`;
- `'django.db.backends.oracle'` и другие.

Поле **NAME** предназначено для указания названия базы данных, созданной для проекта. Если вы используете SQLite, база данных будет файлом на вашем компьютере; в таком случае NAME должна

содержать полный путь, включая название этого файла. Значение по умолчанию, `os.path.join(BASE_DIR, 'db.sqlite3')`, сохранит файл в каталоге проекта.

Поля **USER** и **PASSWORD** предназначены для указания имени и пароля пользователя базы данных. Не используется для SQLite.

HOST – имя хоста используемого при подключении к базе данных. Пустая строка подразумевает localhost. Не используется для SQLite.

! При использовании PostgreSQL, по умолчанию (при пустом HOST) подключение к базе данных будет выполнено через UNIX domain сокет ('local' в `pg_hba.conf`). Если вы хотите использовать TCP сокеты, укажите в HOST '`localhost`' или '`127.0.0.1`' ('`host`' в `pg_hba.conf`). В Windows необходимо указать HOST, т.к. UNIX domain сокеты не доступны.

PORT. Порт, используемый при подключении к базе данных. Пустая строка подразумевает порт по умолчанию. Не используется для SQLite. По умолчанию: "".

7.5.2. Основные настройки проекта

- **ALLOWED_HOSTS** – список хостов/доменов, для которых может работать текущий сайт.
- **DEBUG** включает/выключает режим отладки.

! Не рекомендуется включать DEBUG на «боевом» сервере.

- **INSTALLED_APPS** – приложения Django проекта.

По умолчанию содержит следующие приложения:

- `django.contrib.admin` – интерфейс администратора.
- `django.contrib.auth` – система аутентификации.
- `django.contrib.contenttypes` – «content types» фреймворк.
- `django.contrib.sessions` – фреймворк сессии.
- `django.contrib.messages` – фреймворк сообщений.
- `django.contrib.staticfiles` – фреймворк для работы со статическими файлами.

- ***LANGUAGE_CODE*** - код используемого в проекте языка. Для России: *LANGUAGE_CODE* = 'ru-ru'.
- ***MEDIA_ROOT*** – абсолютный путь к каталогу, в котором хранятся медиа-файлы проекта.
- ***MEDIA_URL*** – URL, который указывает на каталог *MEDIA_ROOT*. Должен оканчиваться слешем при не пустом значении. Необходимо настроить раздачу файлов, как dev-сервером, так и боевым.
- ***STATICFILES_DIRS*** указывает каталоги, в которых необходимо искать статические файлы.
- ***STATIC_ROOT*** – абсолютный путь к каталогу, в который команда *collectstatic* соберёт все статические файлы.

!Это должен быть каталог (изначально пустой), куда будут скопированы все статические файлы для более простой настройки сервера; это **не** каталог, в котором вы создаете статические файлы при разработке. Вы должны создавать статические файлы в каталогах, которые будут найдены модулями поиска статических файлов, это каталоги 'static/' в приложениях и каталоги, указанные в *STATICFILES_DIRS*.

- ***STATIC_URL*** – URL, указывающий на каталог со статическими файлами *STATIC_ROOT*. Например: *'/static/'* или *'http://static.mysite.com/'*.
- ***TEMPLATES***– список настроек для шаблонизаторов. Каждый элемент – это список настроек для шаблонизатора.

Некоторые опции:

- ***BACKEND*** – Бэкенд шаблонизатора, который используется. Django:
'django.template.backends.django.DjangoTemplates' и *'django.template.backends.jinja2.Jinja2'*.
- ***DIRS*** – список каталогов, в которых будут искать файлы шаблонов в указанном порядке.

- ***APP_DIRS*** – должен ли шаблонизатор искать файлы шаблонов в приложениях. Мы укажем *True*.
- ***OPTIONS*** – дополнительные параметры для бэкенда шаблонизатора. Доступные параметры зависят от используемого бэкенда.

Например, для бэкенда DjangoTemplates одним из параметров является '*context_processors*', который принимает объект запроса и возвращает словарь с данными, которые будут добавлены в контекст. Удобно при использовании каких-либо данных в нескольких местах веб-сайта.

- ***TIME_ZONE*** – строка, указывающая на используемый проектом часовой пояс. Для России: *TIME_ZONE = 'Europe/Moscow'*.
- ***USE_TZ*** указывает, используется ли часовой пояс. При *True* Django будет использовать объекты даты и времени с указанным часовым поясом. Иначе Django будет использовать объекты даты и времени без учета часового пояса.

После выполнения настроек выполните:

```
python manage.py migrate
```

Команда ***migrate*** анализирует значение *INSTALLED_APPS* и создает таблицы в базе данных, используя настройки базы данных из файла *mysite/settings.py* и миграции из приложения. Вы увидите сообщение о каждой выполненной миграции.

7.5.3. Запуск сервера для разработки

Для запуска сервера перейдите во внешний каталог *mysite* и выполните команду:

```
python manage.py runserver
```

Только что был запущен встроенный Web-сервер для разработки Django проектов.

Теперь, когда сервер запущен, перейдите на страницу *http://127.0.0.1:8000/* в браузере. Если вы увидели сообщение

«Заработало! Поздравляем вас с вашей первой страницей, работающей на Django.», то поздравляем, всё работает!

По умолчанию, команда *runserver* запускает сервер для разработки на локальном IP используя порт 8000.

Если вы хотите изменить порт, укажите его как аргумент. Например, эта команда запускает сервер, используя порт 8080:

```
python manage.py runserver 8080
```

Если вы хотите показать свою работу на других компьютерах, то передайте публичный IP вместе со значением порта.

```
python manage.py runserver 0.0.0.0:8000
```

7.5.4. Создание приложения

Создавая приложение, убедитесь, что вы находитесь в том же каталоге, что и файл *manage.py*, и выполните команду:

```
python manage.py startapp mypersonal
```

! Приложение может находиться где угодно в путях Python и оно может быть импортировано как независимый модуль, а не подмодуль *mysite*. Эта команда создаст каталог *mypersonal*.

```
mypersonal/  
  migrations/  
    __init__.py  
  __init__.py  
  admin.py  
  models.py  
  tests.py  
  views.py
```

Приложение необходимо подключить к проекту.

Отредактируйте файл *mysite/settings.py* и измените *INSTALLED_APPS*, добавив строку '*mypersonal*'.

```
INSTALLED_APPS = (  
    'django.contrib.admin',  
    'django.contrib.auth',  
    'django.contrib.contenttypes',
```

```
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles',
'mypersonal',
)
```

7.5.5. Создание моделей

Модель реализуется в файле *models.py*. Она представляет собой описание данных в базе и является эквивалентом SQL-кода *CREATE TABLE*. Django автоматически генерирует поле *id* для каждой таблицы. Так же, как и при работе с любой базой данных, можно создать отношения между таблицами.

Поля модели

Рассмотрим, какие могут быть поля у модели.

Таблица 6.1. Основные типы полей модели

Поле	Описание
<i>AutoField</i>	<i>class AutoField(**options)</i> Автоинкрементное поле <i>IntegerField</i> . Используется для хранения <i>id</i> . Добавляется автоматически.
<i>BooleanField</i>	<i>class BooleanField(**options)</i> Поле хранящее значение <i>true/false</i> .
<i>NullBooleanField</i>	<i>class NullBooleanField([**options])</i> Как и <i>BooleanField</i> , но принимает значение <i>NULL</i> .
<i>CharField</i>	<i>class CharField(max_length=None[, **options])</i> Строковое поле для хранения коротких строк. Принимает один аргумент: <ul style="list-style-type: none"> <i>max_length</i> (максимальная длина (в символах) этого поля).

<i>DateField</i>	<p><i>class DateField([auto_now=False, auto_now_add=False, **options])</i></p> <p>Поле для даты, представленное в виде объекта <i>datetime.date</i> Python. Принимает несколько дополнительных параметров:</p> <ul style="list-style-type: none"> • <i>auto_now</i> (значение поля будет автоматически установлено в текущую дату при каждом сохранении объекта); • <i>auto_now_add</i> (значение поля будет автоматически установлено в текущую дату при создании).
<i>DateTimeField</i>	<p><i>class DateTimeField([auto_now=False, auto_now_add=False, **options])</i></p> <p>Поле для ввода даты и времени.</p>
<i>DecimalField</i>	<p><i>class DecimalField(max_digits=None, decimal_places=None[, **options])</i></p> <p>Десятичное число с фиксированной точностью, представленное объектом <i>Decimal</i> Python. Принимает два обязательных параметра:</p> <ul style="list-style-type: none"> • <i>max_digits</i> (максимальное количество цифр в числе); • <i>decimal_places</i> (количество знаков после запятой).
<i>EmailField</i>	<p><i>class EmailField([max_length=254, **options])</i></p> <p>Поле <i>CharField</i> для хранения правильного email-адреса.</p>
<i>FileField</i>	<p><i>class FileField([upload_to=None, max_length=100, **options])</i></p>

	<p>Поле для загрузки файла. Принимает два дополнительных аргумента:</p> <ul style="list-style-type: none"> • <i>upload_to</i> (путь для хранения файлов относительно значения настройки MEDIA_ROOT); • <i>storage</i> (отвечает за хранение и получение файлов).
<i>FloatField</i>	<p><i>class FloatField(**options)</i> Число с плавающей точкой представленное объектом <i>float</i>.</p>
<i>ImageField</i>	<p><i>class ImageField(upload_to=None, height_field=None, width_field=None, max_length=100, **options)</i> Наследует все атрибуты и методы поля <i>FileField</i>, но также проверяет, является ли загруженный файл изображением. В дополнение к атрибутам <i>upload_to</i> и <i>storage</i> содержит также два дополнительных атрибута:</p> <ul style="list-style-type: none"> • <i>height_field</i> (значение высоты, которое автоматически будет присвоено изображению при каждом сохранении объекта); • <i>width_field</i> (значение ширины, которое автоматически будет присвоено изображению при каждом сохранении объекта). <p>!Требуется библиотека Pillow.</p>
<i>IntegerField</i>	<p><i>class IntegerField(**options)</i> Целое число.</p>
<i>PositiveIntegerField</i>	<p><i>class PositiveIntegerField(**options)</i> Целое положительное число.</p>

<i>TextField</i>	<i>class TextField([**options])</i> Большое текстовое поле.
<i>TimeField</i>	<i>class TimeField([auto_now=False, auto_now_add=False, **options])</i> Поле для ввода времени. Принимает те же аргументы, что и <i>DateField</i> .
<i>URLField</i>	<i>class URLField([max_length=200, **options])</i> Поле CharField для URL.

Таблица 6.2. Основные параметры поля модели

Параметр	Описание
<i>null</i>	По умолчанию: <i>False</i> . При <i>True</i> Django сохранит пустое значение как <i>NULL</i> в базе данных. ! Избегайте использования <i>null</i> для строковых полей таких, как <i>CharField</i> и <i>TextField</i> , т.к. пустое значение всегда будет сохранено как пустая строка, а не <i>NULL</i> .
<i>blank</i>	Значение по умолчанию – <i>False</i> . При <i>blank=True</i> , проверка данных в форме позволит сохранять пустое значение в поле. При <i>blank=False</i> поле будет обязательным.
<i>choices</i>	Варианты значений для поля. Если этот параметр указан, в форме будет использоваться <i>select</i> для этого поля.
<i>db_column</i>	Имя колонки в базе данных для хранения данных этого поля. Если этот параметр не указан, Django будет использовать название поля.
<i>default</i>	Значение по умолчанию для поля.

<i>editable</i>	При <i>False</i> , поле не будет отображаться в админке или любой другой <i>ModelForm</i> для модели.
<i>error_messages</i>	<p>Позволяет переопределить сообщения ошибок возвращаемых полем. Используйте словарь с ключами соответствующими необходимым ошибкам.</p> <p>Ключи ошибок:</p> <ul style="list-style-type: none"> • <i>null</i>; • <i>blank</i>; • <i>invalid</i>; • <i>invalid_choice</i>; • <i>unique</i>; • <i>unique_for_date</i>.
<i>help_text</i>	Подсказка, отображаемая под полем в интерфейсе администратора. Полезно для описания поля, даже если модель не используется в форме.
<i>primary_key</i>	При <i>True</i> это поле будет первичным ключом.
<i>unique</i>	<p>При <i>True</i> значение поля должно быть уникальным.</p> <p>!Параметр можно использовать для любого типа кроме <i>ManyToManyField</i>, <i>OneToOneField</i> и <i>FileField</i>.</p>
<i>unique_for_date</i> <i>unique_for_month</i> <i>unique_for_year</i>	<p>Этот параметр должен быть равен названию поля с типом <i>DateField</i> или <i>DateTimeField</i>, для которого значение должно быть уникальным.</p> <p>Например, если модель имеет поле <i>task</i> с <i>unique_for_date="task_date"</i>, тогда</p>

	<p>Django позволит сохранять записи только с уникальной комбинацией <i>task</i> и <i>task_date</i> (1 задача в день).</p> <p>! Для полей типа <i>DateTimeField</i> учитывается только дата.</p> <p><i>unique_for_date</i> – значение уникально для даты.</p> <p><i>unique_for_month</i> – уникально для месяца.</p> <p><i>unique_for_year</i> –уникально для года.</p>
<i>verbose_name</i>	Отображаемое имя поля.
<i>validators</i>	Список проверок, выполняемых для поля.

С полным списком типов полей и параметров можете ознакомиться на официальном сайте [django](https://docs.djangoproject.com/en/3.2/ref/models/fields/).

Для примера создадим модель «Мои задачи», которая содержит следующие поля: дата, тип задачи, тема, текст, файл и статус. Задача последнего поля – пометать выполненные задачи.

```
# -*- coding:utf-8 -*-
from django.db import models

# Модель «Мои задачи»
class Task(models.Model):
    TYPE_CHOICES = (
        (0, u'Обычная задача'),
        (1, u'Срочная задача'),
        (2, u'Важная задача!'),
        (3, u'Другое'),
    )
    task_date = models.DateField(verbose_name = 'Дата',
auto_now_add=True)
    type = models.PositiveIntegerField(choices=TYPE_CHOICES,
```

```

verbose_name = 'Тун задачи', default=0)
task = models.CharField(max_length=200, verbose_name = 'Тема')
text = models.TextField(verbose_name = 'Текст', blank=True)
file = models.FileField(upload_to='task_files', blank=True)
status = models.NullBooleanField(verbose_name = 'Сматус')
def __unicode__(self):
    return unicode(self.task)
class Meta:
    verbose_name = 'Задача'
    verbose_name_plural = 'Задачи'

```

В данном примере к одной задаче можно прикрепить только один файл. Но если файлов несколько?

Необходимо создать ещё одну модель «Файлы» и связать таблицы отношением *ForeignKey*.

```

# Модель «Файлы»
class TaskFiles(models.Model):
    task = models.ForeignKey(Task, verbose_name = 'Задача')
    file = models.FileField(verbose_name = 'Файл',
upload_to='task_files')

```

Теперь к одной задаче можно привязать сколько угодно файлов.

Миграции

После создания модели необходимо добавить миграции.

Django использует миграции для переноса изменений в моделях (добавление поля, удаление модели и т.д.) в базу данных.

Django предоставляет две команды для работы с миграциями и структурой базы данных:

- ***makemigrations***, которая отвечает за создание новых миграций на основе изменений в моделях.
- ***migrate***, которая отвечает за применение миграций, за откат миграций и за вывод статуса миграций.

Необходимо отметить, что миграции создаются и работают в контексте отдельного приложения. Файлы с миграциями находятся в каталоге *migrations* приложения.

После того, как вы внесли изменения в модель, выполните *makemigrations*:

```
python manage.py makemigrations mypersonal
```

Ваши модели будут просканированы и сравнены с версией, которая содержится в файлах миграций, затем будут созданы новые миграции. Не забывайте проверять вывод команды, чтобы понимать, как *makemigrations* видит ваши изменения - для сложных изменений вы можете получить не совсем ожидаемый результат.

Создав новые миграции, вам следует применить их к вашей базе с помощью команды *migrate*.

Отношения между моделями

Таблица 6.3. Основные параметры поля модели

Тип отношения	Описание
<i>ForeignKey</i>	<p><i>class ForeignKey(othermodel[, **options])</i></p> <p>Связьное-к-одному.</p> <p>Принимает позиционный аргумент: класс связанной модели.</p> <p>Основные параметры:</p> <ul style="list-style-type: none"> • <i>limit_choices_to</i>(указывает полю <i>ModelForm</i> показывать объекты, которые соответствуют некоторому условию); <div> <pre><i>user = models.ForeignKey(User, limit_choices_to={'is_staff': True})</i></pre> </div> <p>Вместо словаря можете использовать объект <i>Q</i> или функцию.</p> <ul style="list-style-type: none"> • <i>related_name</i>(по этому названию

	<p>устанавливается обратная связь); Если вы не хотите обратной связи, установите <i>related_name</i> в '+' или в конце '+'. <ul style="list-style-type: none"> • <i>related_query_name</i> (название обратной связи используемое при фильтрации результата запроса); По умолчанию: <i>related_name</i>, или название модели. • <i>to_field</i> (поле связанной модели, которое используется для создания связи между таблицами); По-умолчанию, Django использует первичный ключ. • <i>on_delete</i> (очень важный аргумент, который предпринимает какие-либо действия при удалении связанного объекта). <p>Возможные значения для <i>on_delete</i>:</p> <ul style="list-style-type: none"> ○ <i>CASCADE</i> (каскадное удаление, значение по умолчанию); ○ <i>PROTECT</i>(препятствует удалению связанного объекта, вызывая исключение); ○ <i>SET_NULL</i>(устанавливает <i>ForeignKey</i> в <i>NULL</i>; ВОЗМОЖНО при <i>null</i> равен <i>True</i>); ○ <i>SET_DEFAULT</i>(устанавливает <i>ForeignKey</i> в значение по умолчанию; значение по- </p>
--	---

	<p>умолчанию должно быть указано для <code>ForeignKey</code>);</p> <ul style="list-style-type: none"> ○ <i>SET()</i>(устанавливает <i>ForeignKey</i> значение указанное в скобках); ○ <i>DO_NOTHING</i> (ничего не делать).
<i>ManyToManyField</i>	<p><code>class ManyToManyField(othermodel[, **options])</code></p> <p>Связь многие-ко-многим. Принимает позиционный аргумент: класс связанной модели. Работает, так же как и <i>ForeignKey</i>.</p>
<i>OneToOneField</i>	<p><code>class OneToOneField(othermodel[, parent_link=False, **options])</code></p> <p>Связь один-к-одному. Работает так же, как и <i>ForeignKey</i> с <i>unique=True</i>, но «обратная» связь возвращает один объект.</p> <p>Обязательный позиционный аргумент: класс связанной модели.</p> <p>В основном применяется как первичный ключ, которая «расширяет» другую модель. Например, если необходима дополнительная информация для модели <i>User</i> (номер телефона, фото и т.д.).</p>

С параметрами, используемыми в связях типа *ManyToManyField* и *OneToOneField*, можете ознакомиться на официальном сайте Django.

Выполнение запросов

В Django большинство взаимодействий с базой данных осуществляется посредством механизма объектно-ориентированного отображения (Object-Relational Mapper или ORM).

Вы можете получать, создавать, изменять, удалять объекты.

Получение объектов

Для получения объектов из базы данных, создается *QuerySet* через *Manager* модели. Менеджер (Manager) - это интерфейс, через который создаются запросы к моделям Django. Каждая модель имеет хотя бы один менеджер – *objects* по умолчанию.

Для получения одного объекта используйте метод *get()*. Данный способ можно применять, если вы знаете, что только один объект возвращается запросом. Иначе, если результат пустой или существует несколько объектов, удовлетворяющих запросу, то *get()* вызовет исключение.

```
t = Task.objects.get(pk=1)
```

Для получения всех объектов используйте метод *all()* менеджера:

```
t = Task.objects.all()
```

QuerySet, возвращенный *Manager*, описывает все объекты в таблице базы данных. Обычно вам нужно выбрать только подмножество всех объектов.

QuerySet может быть создан, отфильтрован, ограничен и использован фактически без выполнения запросов к базе данных. База данных не будет затронута, пока вы не спровоцируете выполнение *QuerySet*.

Таблица 6.4. Некоторые методы, возвращающие новый *QuerySet*

Метод	Описание
-------	----------

<i>filter()</i>	<p>Данный метод добавляет условия фильтрации.</p> <p>Например, для создания <i>QuerySet</i> с записями от 2014, используйте <i>filter()</i> таким образом:</p> <pre><i>Task.objects.filter(task_date__gte=2014)</i></pre>
<i>exclude()</i>	<p>Метод <i>exclude()</i> возвращает новый <i>QuerySet</i>, содержащий объекты, которые не удовлетворяют параметры фильтрации. Следующий пример получает записи до 2014 года.</p> <pre><i>Task.objects.exclude(task_date__gte=2014)</i></pre>
<i>order_by()</i>	<p>Для того чтобы отсортировать по полям объекты, существует метод <i>order_by()</i>.</p> <pre><i>Task.objects.filter(task_date__year=2015).order_by('-task_date')</i></pre> <p>Результат данного примера будет отсортирован в обратном порядке по полю <i>task_date</i>. Знак «-» указывает на обратную сортировку. По-умолчанию: сортировка по возрастанию. Что бы отсортировать случайно используйте "?". Можно передать несколько аргументов в метод. Тогда результат будет отсортирован по нескольким полям, в порядке очереди.</p> <p>Для сортировки по полю из другой модели, используйте синтаксис аналогичный тому, который</p>

	используется при фильтрации по полям связанной модели, то есть с применением двух знаков нижнего подчеркивания.
<i>reverse()</i>	Используйте метод, чтобы изменить порядок сортировки на обратный.
<i>distinct()</i>	Для исключения повторяющихся записей существует метод <i>distinct()</i> . Данный способ полезен, если запрос использует несколько таблиц.
<i>values()</i>	<p>Возвращает словари с результатом вместо объектов моделей. Каждый словарь представляет объект, ключи которого соответствуют полям модели.</p> <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> <i>Task.objects.filter(task_date__gte=2014).values()</i> </div> <p>Данный способ полезен, если вам нужны только данные некоторых полей и не нужен функционал объектов моделей.</p> <p>! Существует также метод, возвращающий кортеж <i>values_list()</i>.</p>
<i>none()</i>	Возвращает пустой список.

<i>select_related()</i>	<p>Возвращает <i>QuerySet</i> который автоматически включает в выборку данные связанных объектов при выполнении запроса. Повышает производительность, но увеличивает (иногда значительно) объем получаемых данных, в результате, при доступе к связанным объектам через модель, не потребуются дополнительные запросы в базу данных.</p> <pre> t = Task.objects.select_related().get(id=2) file = t.file </pre>
-------------------------	---

Также вы можете ограничить результат выборки *QuerySet*:

<p># Использование среза для ограничения выборки</p> <pre>Task.objects.all()[:10]</pre>

Данный код возвращает 10 первых объектов (*LIMIT 10*).

Фильтры полей

Фильтры полей – это «операторы» для составления условий SQL *WHERE*. Они задаются как именованные аргументы для метода *filter()*, *exclude()* и *get()* в *QuerySet*.

Чтобы применить фильтры используйте двойное подчеркивание после аргумента.

Таблица 6.5. Основные фильтры полей

Фильтр	Описание
<i>contains</i>	<p>Регистрозависимая проверка на вхождение.</p> <pre>Task.objects.filter(task__contains=«Позвонить»)</pre>

<i>icontains</i>	Регистронезависимая проверка на вхождение.
<i>year</i>	<p>Проверка года для полей <i>date/datetime</i>. Принимает числовое значение года.</p> <div> <i>Task.objects.filter(task_date__year=2015)</i> </div>
<i>month</i>	Проверка месяца для полей <i>date/datetime</i> (от 1(январь) до 12(декабрь)).
<i>day</i>	Проверка дня месяца для полей <i>date/datetime</i> . Принимает номер дня месяца.
<i>week_day</i>	Проверка дня недели для полей <i>date/datetime</i> (от 1 (воскресение) до 7 (суббота)).
<i>hour</i>	Проверка часа для полей <i>date/datetime</i> (от 0 до 23).
<i>minute</i>	Проверка минуты для полей <i>date/datetime</i> (от 0 до 59).
<i>second</i>	Проверка секунды для полей <i>date/datetime</i> (от 0 до 59).
<i>gt</i>	<p>Больше чем (>).</p> <div> <i>Task.objects.filter(task_date__year__gt=2014)</i> </div>
<i>gte</i>	Больше чем или равно (>=).
<i>exact</i>	<p>Регистронезависимое точное совпадение.</p> <div> <i>Task.objects.filter(task__exact=«Позволить преподавателю»)</i> </div> <p>! Регистрозависимое точное</p>

	совпадение <i>exact</i> и запрос без фильтра полей – одно и то же.
<i>isnull</i>	Принимает <i>True</i> или <i>False</i> . <div><i>Task.objects.filter(status__isnull=True)</i></div>
<i>lt</i>	Меньше чем (<). <div><i>Task.objects.filter(task_date__year__lt=2014)</i></div>
<i>lte</i>	Меньше чем или равно (<=).
<i>range</i>	Проверка на вхождение в диапазон (включающий). <div><i>Task.objects.filter(task_date__year__range(2014, 2015))</i></div>

К *QuerySet* можно добавлять несколько разных фильтров.

Для сложных запросов следует использовать объект *Q*.

Объект *Q* – объект, используемый для инкапсуляции множества именованных аргументов для фильтрации.

Объекты *Q* могут быть объединены операторами *&* и */*, при этом будет создан новый объект *Q*.

*Task.objects.filter(Q(task__icontains = 'Перезвонить')
&~Q(task_date__year=2014))*

Данный запрос вернёт все задачи не 2014 года, у которых в теме есть слово «перезвонить».

Связанные объекты

При обращении к полю типа *ForeignKey* возвращается связанный объект модели.

t = TaskFiles.objects.get(pk = 1)
t.task

Связь работает и в обратном направлении. Можно получить все файлы, относящиеся к какому-либо заданию.

```
t = Task.objects.get(pk = 1)
t.taskfiles_set.all()
```

Имя атрибута *taskfiles_set* образуется путём добавления суффикса *_set* к имени модели в нижнем регистре.

! Вы можете переопределить название *taskfiles_set*, установив параметр *related_name* при определении *ForeignKey*.

Создание объектов

Чтобы создать объект, создайте экземпляр класса модели, указав необходимые поля в аргументах, и вызовите метод *save()*, чтобы сохранить его в базе данных.

```
# Создание объекта «Новая задача»
t = Task(type = 0, theme='Позвонить преподавателю')
t.save()
```

Изменить несколько объектов можно при помощи *update()*.

```
# Изменение нескольких объектов
Task.objects.filter(task_date__year=2015).update(type = 2)
```

! Используется SQL запрос, метод *save()* не вызывается. Если хотите сохранить несколько объектов методом *save()*, то используйте цикл.

Для сохранения изменений в объект, который уже существует в базе данных, также используйте *save()*.

```
# Изменение объекта модели
t = Task.objects.get(pk=1)
t.theme = 'Позвонить преподавателю по социологии'
t.save()
```

Метод удаления называется *delete()*.

```
# Удаление объекта
t = Task.objects.get(pk=1)
t.delete()
```

Также можно удалить несколько объектов.

```
# Удаление нескольких объектов
```

```
Task.objects.filter(task_date__lte=2014).delete()
```

Если вы хотите удалить все объекты, сначала явно получите QuerySet, содержащий все записи, потом произведите удаление.

```
# Удаление всех объектов модели
```

```
Task.objects.all().delete()
```

7.5.6. Создание представлений

Файл *views.py* служит для реализации представлений, которые представляют собой разнообразные функции. Здесь содержится вся логика приложения.

В Django страницы и остальной контент отдается представлениями. Представление – это просто функция Python (или метод представления-класса). Django выбирает представление, анализируя запрошенный URL.

Откроем файл *views.py* приложения *mypersonal*.

Первое, что мы сделаем, создадим страницу с приветствием.

```
# -*- coding:utf-8 -*-
```

```
from django.http import HttpResponse
```

```
def index(request):
```

```
    return HttpResponse("Мояличнаястраница")
```

Это самое простое из возможных представлений в Django. Теперь необходимо связать это представление с URL. Для этого пропишем в файле *mypersonal/urls.py*:

```
from django.conf.urls import url
```

```
from mypersonal import views
```

```
urlpatterns = [
```

```
    url(r'^$', views.index, name='index'),
```

```
]
```

Функция *url()* принимает четыре аргумента, два обязательных: *regex* (регулярное выражение) и *view* (вызываемая функция), и два необязательных: *kwargs* (аргументы) и *name* (название URL-а).

Как это работает? При нахождении подходящего регулярного выражения, Django вызывает функцию Python, передавая первым аргументом объект *HttpRequest*, а потом все позиционные или именованные аргументы. Каждое представление должно вернуть объект *HttpResponse*.

Теперь в главном URLconf подключим модуль *mypersonal.urls*. В *mysite/urls.py* добавьте *include()*:

```
from django.conf.urls import include, url
from django.contrib import admin
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^mypersonal/', include('mypersonal.urls')),
]
```

Открыв <http://localhost:8000/mypersonal/> в браузере, вы должны увидеть ваш текст «Моя личная страница» из представления.

Динамические url

Красивый URL является важной составляющей любого веб-сайта. Django поощряет создание элегантных URL-адресов, что не свойственно для *.php* и *.asp*. Например, */tasks/3* вместо */tasks?id=3*.

Иногда содержимое страницы должно зависеть от какого-то параметра. Например, если необходимо рассмотреть каждую задачу по отдельности.

Регулярные выражения определяют синтаксис, позволяющий задать шаблоны, с которыми будет сверяться строки.

Рассмотрим динамический URL-адрес, в котором будет меняться номер задачи. Переменная часть адреса заключена в скобки.

```
url(r'^tasks/$', views.tasks, name='tasks'), # статический URL
url(r'^tasks/(?P<id>\d+)/$', views.task_view, name='task'), #
динамический URL
```

Мы видим, что для сопоставления с числом используется регулярное выражение `\d+`. Такой шаблон URL соответствует любому URL вида `/tasks/1/` и даже `/tasks/10000`.

Ниже описаны наиболее часто используемые шаблоны регулярных выражений.

Таблица 6.6. Часто используемые шаблоны регулярных выражений

Символ	Описание
<code>.</code> (точка)	Любой символ
<code>\d</code>	Любая цифра
<code>[A-Z]</code>	Любая буква (верхний регистр)
<code>[a-z]</code>	Любая буква (нижний регистр)
<code>[A-Za-z]</code>	Любая буква (любой регистр)
<code>+</code>	Один или более символов предыдущего выражения, т.е. <code>\d+</code> совпадает с одной или более цифрами
<code>[^/]+</code>	Все символы подряд, кроме слэша
<code>?</code>	Наличие или отсутствие предыдущего выражения, т.е. <code>\d?</code> описывает возможное наличие одной цифры
<code>{1, 3}</code>	От одного до трёх (символов) предыдущего выражения

Также необходимо отметить, что представление в данном случае принимает дополнительный аргумент – `id` задачи.

```
def task_view(request, id):
```

```
...
```

Вспомогательные функции

Пакет `django.shortcuts` содержит вспомогательные функции и классы, упрощающие разработку и код.

Таблица 6.7. Вспомогательные функции

Функция	Описание
---------	----------

<p><i>render_to_response</i></p>	<p><i>render_to_response(template_name[, context][, context_instance][, content_type][, status][, dirs][, using])</i></p> <p>Выполняет указанный шаблон с переданным словарем контекста и возвращает <i>HttpResponse</i> с полученным содержимым.</p> <p>Обязательные аргументы:</p> <ul style="list-style-type: none"> • <i>template_name</i> (шаблон или список шаблонов. Если передать список, будет использован первый существующий шаблон). <p>Некоторые необязательные аргументы:</p> <ul style="list-style-type: none"> • <i>context</i> (словарь переменных для контекста шаблона); • <i>content_type</i>(MIME-тип результата. По умолчанию: <i>'text/html'</i>); • <i>context_instance</i> (контекст, который будет использован при выполнении шаблона. При использовании процессоров контекста, необходимо передать экземпляр <i>RequestContext</i>).
----------------------------------	---

<p><i>render</i></p>	<p><i>render(request, template_name[, context][, context_instance][, content_type][, status][, current_app][, dirs][, using])</i></p> <p>Вызывает указанный шаблон с переданным словарем контекста и возвращает <i>HttpResponse</i> с полученным содержимым.</p> <p>Функция <i>render()</i> аналогична вызову функции <i>render_to_response()</i> с <i>context_instance</i>, который указывает использовать <i>RequestContext</i>.</p> <p>Обязательные аргументы:</p> <ul style="list-style-type: none"> • <i>request</i> (объект обрабатываемого запроса); • <i>template_name</i>. <pre>from django.shortcuts import render def my_tasks(request): task = Task.objects.filter(date = date.today()) return render(request, 'mypersonal/tasks.html', {"task_list": task}, content_type="application/xhtml+xml")</pre>
<p><i>redirect</i></p>	<p><i>redirect(to, [permanent=False,]*args, **kwargs)</i></p> <p>Направляет на указанный URL.</p> <p>В аргументах можно передать:</p> <ul style="list-style-type: none"> • экземпляр модели (в качестве

	<p>URL-а для перенаправления будет использоваться результат вызова метода <code>get_absolute_url()</code>;</p> <ul style="list-style-type: none"> • название представления, возможно с аргументами; • абсолютный или относительный URL, который будет использован для перенаправления на указанный адрес. <p>По умолчанию использует временное перенаправление, используйте аргумент <code>permanent=True</code> для постоянного перенаправления.</p>
<code>get_object_or_404</code>	<p><code>get_object_or_404(klass, *args, **kwargs)</code></p> <p>Вызывает <code>get()</code> и возвращает полученный объект. Если такого объекта нет, вызывает исключение <i>Http404</i> вместо <i>DoesNotExist</i>.</p> <p>Обязательные аргументы:</p> <ul style="list-style-type: none"> • <i>klass</i> (модель, который будет использован для получения объекта); • <i>**kwargs</i> (параметры поиска для методов <code>get()</code> и <code>filter()</code>). <div style="border: 1px solid black; padding: 5px; margin-top: 10px;"> <pre>from django.shortcuts import get_object_or_404 def my_tasks(request): task = get_object_or_404(Task, pk=1)</pre> </div>
<code>get_list_or_404</code>	<p><code>get_list_or_404(klass, *args, **kwargs)</code></p> <p>Возвращает результат метода</p>

	<i>filter()</i> для переданного менеджера модели, вызывает <i>Http404</i> если получен пустой список.
--	---

7.5.7. Шаблоны

Передавать весь текст страницы через *HttpResponse* неудобно. Задачей Django является разделение бизнес-логики и логики отображения. Для этого и применяют шаблоны.

Шаблон это просто текстовый файл. Он позволяет создать любой текстовый формат (HTML, XML, CSV, и др.).

Шаблоны содержат динамические данные, передаваемые из *views.py*.

Переменные выглядят таким образом: *{{ value }}*. Когда шаблон встречает переменную, он вычисляет ее и заменяет результатом.

Также как и в Python, доступ к атрибутам переменной возможен через точечную нотацию.

Встроенные теги

Кроме переменных в шаблоне Django вы можете встретить следующую комбинацию: *{% tag %}*. Это шаблонный тег. Тег предназначен для добавления какой-либо логики в шаблон. Например, если вы встретите *{% for i in task_list %}...{% endfor %}*, значит, здесь присутствует цикл.

Ниже описаны самые часто используемые теги Django.

Таблица 6.8. Встроенные теги

Тег	Описание
-----	----------

<i>{% autoescape %}</i>	<p>Контролирует авто-экранирование. Этот тег принимает <i>on</i> или <i>off</i> аргумент, указывающий должно ли использоваться автоматическое экранирование внутри блока. Блок закрывается закрывающим тегом <i>endautoescape</i>.</p> <div> <pre><i>{% autoescape on %}</i> {{ body }} <i>{% endautoescape %}</i></pre> </div>
<i>{% block %}</i>	<p>Данный блок применяют при наследовании шаблонов, которое будет рассмотрено ниже.</p>
<i>{% csrf_token %}</i>	<p>Этот тег используется для CSRF защиты.</p>
<i>{% extends ... %}</i>	<p>Указывает, что данный шаблон наследуется от родительского. Может использоваться двумя способами:</p> <div> <pre><i>{% extends "base.html" %}</i></pre> </div> <p>или</p> <div> <pre><i>{% extendstemplate_name%}</i></pre> </div> <p>где "base.html" и <i>template_name</i>(переданная переменная) являются названиями родительского шаблона.</p>

<pre>{% for ... empty ... endfor %}</pre>	<p>Обходит все элементы.</p> <pre>{% for t in task_list %} <p>{{ t.task }}</p> {% empty %} <p>Задач не найдено</p> {% endfor %}</pre> <p>Дополнительные переменные в цикле:</p> <ul style="list-style-type: none"> • forloop.counter (номер текущей итерации цикла начиная с 1); • forloop.counter0 (номер текущей итерации цикла с 0); • forloop.revcounter (номер текущей итерации цикла начиная с конца с 1); • forloop.revcounter0 (номер текущей итерации цикла начиная с конца с 0); • forloop.first (<i>True</i>, если это первая итерация); • forloop.last (<i>True</i>, если это последняя итерация); • forloop.parentloop («внешний» цикл для вложенных циклов).
<pre>{% if ... else ... endif %}</pre>	<p>Проверяет истинность того, что находится внутри тега. Если <i>True</i>, то выводит содержимое.</p> <pre>{% ift.type == 2 %} Это важно! {% elif %} Это не очень важная задача {% endif %}</pre>

<i>{% include %}</i>	<p>Загружает («включает») шаблон и выводит его с текущим контекстом. Можно загрузить двумя способами:</p> <pre>{% include "mypersonal/page.html" %}</pre> <p>или</p> <pre>{% include template_name %}</pre> <p>Вы можете передать дополнительные переменные контекста:</p> <pre>{% include "mypersonal/page.html" with role="student" %}</pre>
<i>{% load %}</i>	<p>Загружает все теги из библиотеки:</p> <pre>{% load somelibrary %}</pre> <p>или определённые теги:</p> <pre>{% load sometag from somelibrary %}</pre>
<i>{% now %}</i>	<p>Отображает текущую дату и/или время, может содержать символы форматирования описанные в разделе о фильтре date и time.</p> <pre>Время: {% now "H:i:s" %}</pre>
<i>{% regroup %}</i>	<p>Группирует объекты по общему атрибуту. Давайте сгруппируем модель <i>Task</i> по типам задач.</p> <pre>{% regroup task_list by type as type_list %}</pre> <pre></pre> <pre>{% for t in type_list %}</pre>

	<pre> {{ t.grouper }} {% for i in t.list %} {{ i.task }}: {{ i.text }} {% endfor %} {% endfor %} </pre> <p><code>{% regroup %}</code> принимает три аргумента:</p> <ul style="list-style-type: none"> • список, который вы хотите сгруппировать (<i>task_list</i>); • атрибут, по которому нужно сгруппировать (<i>type</i>); • результат (<i>type_list</i>). <p><code>{% regroup %}</code> создает список (<i>type_list</i>). Каждый объект группы содержит два атрибута:</p> <ul style="list-style-type: none"> • <i>grouper</i> (значение, по которому происходила группировка (например, строка «Обычная задача» или «Срочная задача»)). • <i>list</i> (список объектов в группе (например, список всех задач по каждому типу)).
<p><code>{% url %}</code></p>	<p>Возвращает абсолютную ссылку с необязательными аргументами.</p> <pre> {% url 'some_url_name' arg1=var1 var2 %} </pre>

<i>{% with %}</i>	Кэширует переменные под простым названием. <div> <i>{% with var = 1 cnt = task_list.count %}</i> <i>{{ cnt }}</i> <i>{% endwith %}</i> </div>
-------------------	--

Шаблонные фильтры

Для форматирования переменных используются шаблонные фильтры. Фильтры присоединяются при помощи вертикальной черты.

Таблица 6.9. Шаблонные фильтры

Фильтр	Описание
<i>add</i>	Суммирует аргумент и значение. <div><i>{{ value add:"2" }}</i></div>
<i>capfirst</i>	Делает заглавным первый символ значения. <div><i>{{ value capfirst }}</i></div>
<i>center</i>	Центрирует значение в поле заданной ширины. <div><i>{{ value center:"10" }}</i></div>
<i>ljust</i>	Выравнивает значение влево в поле указанной ширины. <div><i>{{ value ljust:"10" }}</i></div>
<i>rjust</i>	Выравнивает значение вправо в поле указанной ширины.

<i>cut</i>	<p>Удаляет значение аргумента из строки, к которой применяется фильтр.</p> <pre>{{ value/<i>cut</i>:" " }}</pre>
<i>date</i>	<p>Форматирует дату в соответствии с указанным форматом.</p> <p>Некоторые символы форматирования даты:</p> <ul style="list-style-type: none"> • <i>d</i>(день месяца); • <i>D</i> (день недели); • <i>F</i> (полное название месяца); • <i>m</i> (месяц из двух цифр); • <i>M</i> (3-х буквенное название месяца); • <i>n</i> (номер месяца без ведущего нуля); • <i>t</i> (кол-во дней в месяце); • <i>w</i> (номер дня недели без ведущих нулей); • <i>y</i> (год из двух цифр); • <i>Y</i> (год из четырёх цифр); • <i>z</i> (номер дня в году). <pre>{{ value/<i>date</i>:" d F Y" }}</pre>

<i>time</i>	<p>Форматирует время в соответствии с указанным форматом.</p> <p>Некоторые символы форматирования:</p> <ul style="list-style-type: none"> • g (час, 12-часовом формате без ведущих нулей); • G (час, от 0 до 23); • h (час, от 01 до 12); • H (час, от 00 до 23); • i (минуты); • s (секунды); • u (микросекунды). <pre>{{ value/time:" H:i" }}</pre>
<i>default</i>	<p>Если значение равно False, будет использовано значение по-умолчанию.</p> <pre>{{ value/default:"nothing" }}</pre>
<i>default_if_none</i>	<p>Если значение равно None, будет использовано значение по-умолчанию.</p> <pre>{{ value/default_if_none:"nothing" }}</pre>
<i>dictsort</i>	<p>Сортировка по указанному ключу.</p> <pre>{{ value/dictsort:"task_date" }}</pre>
<i>dictsortreversed</i>	<p>Сортировка в обратном порядке.</p> <pre>{{ value/dictsortreversed:"task_date" }}</pre>
<i>first</i>	<p>Возвращает первый элемент списка.</p>
<i>last</i>	<p>Возвращает последний элемент списка.</p>

<i>join</i>	Объединяет список. <div> <pre>{{ value join:"," }}</pre> </div> Например, при <i>value</i> = ['дом', 'дерево', 'сын'], ответ будет: "дом, дерево, сын".
<i>length</i>	Возвращает размер у строк и списков.
<i>lower</i>	Конвертирует строку в нижний регистр.
<i>upper</i>	Конвертирует строку в верхний регистр.
<i>make_list</i>	Превращает значение в список.
<i>random</i>	Возвращает случайный элемент из списка.
<i>slugify</i>	Конвертирует в нижний регистр, удаляет нетекстовые символы, преобразует пробелы в дефисы, удаляет пробелы в начале и в конце строки.
<i>timesince</i>	Сколько времени прошло с момента другой даты. <div> <pre>{{ date1 timesince:date2 }}</pre> </div>
<i>timeuntil</i>	Время от текущей даты до указанной. <div> <pre>{{ date timeuntil }}</pre> <pre>{{ date1 timeuntil:date2 }}</pre> </div>
<i>truncatechars</i>	Обрезает строку до указанной длины. <div> <pre>{{ value truncatechars:10 }}</pre> </div>

<i>truncatewords</i>	Обрезает строку после указанного количества слов. <div>{{ value/<i>truncatewords</i>:3 }}</div>
<i>wordcount</i>	Возвращает количество слов.
<i>wordwrap</i>	Делит предложение на строки по количеству букв. <div>{{ value/<i>wordwrap</i>:10 }}</div>

Наследование шаблонов

Процесс написания сайтов упрощается благодаря использованию такого мощного инструмента, как наследование шаблонов. Данное решение является более правильным, чем простое включение кода при помощи тега *{% include %}*.

Смысл наследования состоит в разработке скелетного родительского и дочерних шаблонов. Главная страница содержит общие разделы и в ней описываются переопределяемые в наследованных шаблонах блоки. Переопределение становится возможным при добавлении в дочерние страницы тега *{% extends «index.html» %}*, где *index.html* - главная страница.

Некоторые советы по работе с наследованием:

- Тег *{% extends %}* должен быть первым тегом в шаблоне.
- Если вы дублируете содержимое в нескольких шаблонах, возможно, вы должны перенести его в тег *{% block %}* родительского шаблона.
- Дочерний шаблон может не определять все блоки родительского, вы можете указать значение по умолчанию для всех блоков, а затем определить в дочернем шаблоне только те, которые необходимы.

- Если вам необходимо содержимое блока родительского шаблона, используйте переменную `{{ block.super }}`. Эта полезно, если вам необходимо дополнить содержимое родительского блока, а не полностью переопределить его.
- Вы не можете определить несколько тегов одним названием в одном шаблоне.

Создание шаблонов

Все шаблоны в нашем проекте будут находиться в приложениях. Чтобы шаблоны автоматически находились, в настройках проекта мы указали `APP_DIRS = True`.

Добавим главную страницу `index.html` в папку `mypersonal/templates`.

Внешний вид страниц зависит от ваших желаний и навыков верстальщика и дизайнера.

```
<!DOCTYPE html>
<html lang="ru">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<link rel="stylesheet" href="style.css" />
<title>{% block title %}Мояперсональнаястраница{% endblock %}</title>
</head>

<body>
<div id="nav">
    {% block nav %}
<ul>
<li><a href="/">Домой</a></li>
<li><a href="/tasks/">Моизадачи</a></li>
</ul>
```



```

        {% endblock %}
</div>
<div id="content">
    {% block content %}{% endblock %}
</div>
</body>
</html>

```

Распишем страницу «Мои задачи».

Создадим представление *my_tasks* и добавим шаблон *tasks.html*.

```

from django.shortcuts import render
#моязадачи
def tasks(request):
    task_list = Task.objects.all()
    return render(request, 'tasks.html', {'task_list':task_list})

```

Шаблон *tasks.html* наследуется от *index.html*.

```

{% extends'index.html' %}
{% block title %}Моязадачи{% endblock %}
{% block content %}
<ul>
    {% for t in task_list %}
<li><b>{{ t.task_date }}:</b>{{ t.task }}</li>
    {% empty %}
    {% endfor %}
</ul>
{% endblock %}

```

Не забудьте, что представление необходимо связать с url.

7.5.8. Подключение административного интерфейса

Django предоставляет нам мощный встроенный инструмент администрирования. Админка Django предназначена не для обычных посетителей сайта, а для технических специалистов для управления данными на базе созданных моделей. Любая сущность, описанная в

базе данных, может быть отредактирована, добавлена, удалена. Можно легко настроить внешний вид административного интерфейса, страниц с ошибками (404, 500).

Проверьте файл *mysite/urls.py*. Необходимо раскомментировать строки, связанные с подключением административного приложения.

```
from django.conf.urls import include, url
from django.contrib import admin
urlpatterns = [
    url(r'^mypersonal/', include('mypersonal.urls')),
    url(r'^admin/', include(admin.site.urls)),
]
```

Файл для добавления моделей в административный интерфейс имеет название *admin.py*. Он не предусмотрен по умолчанию в приложении. Вам необходимо его создать.

Откройте данный файл. Для того, чтобы наше приложение было доступно в административном интерфейсе, необходимо импортировать подключаемую модель и зарегистрировать её.

```
from django.contrib import admin
from mypersonal.models import *
admin.site.register(Task)
```

После регистрации модели Django отобразит ее на главной странице под названием приложения.

При необходимости можно настроить интерфейс при помощи ряда параметров.

```
# -*- coding:utf-8 -*-
from django.contrib import admin
from mypersonal.models import *

# Класс с параметрами для настройки интерфейса
class TaskAdmin(admin.ModelAdmin):
    list_display = ['task', 'status'] # отображаемые поля
    ordering = ['task_date'] # сортировка
```

```
admin.site.register(Task, TaskAdmin)
```

Чтобы начать пользоваться административным интерфейсом, переходим на страницу *http://127.0.0.1:8000/admin*, используя логин и пароль суперпользователя.

Если суперпользователь не был создан ранее, то вы можете это сделать, выполнив консоли команду:

```
python manage.py createsuperuser
```

8. Практическое задание «Создание страницы «Мои достижения»

1. Одной из частей нашего персонального веб-сайта будет страница с достижениями.

Примечание: все достижения вбиваются в базу через административный интерфейс. То есть необходимо создать модель Advantage и подключить её к админке.

Мои достижения

- | | |
|---------------------------------|--------------------------------------|
| • ДатаНазвание Начало текста... | Перейти к достижению |
| • ДатаНазвание Начало текста... | Перейти к достижению |
| • ДатаНазвание Начало текста... | Перейти к достижению |

Примечание: используйте шаблонные фильтры и теги для оформления страницы.

2. Добавьте сортировку по полям:

- дата создания;
- заголовок;
- «избранное».

2. При нажатии на ссылку «Перейти к достижению» открывается страница с полным описанием.

Полёт на луну

Полный рассказ о том, как вы слетали на луну.

7.5.9. Формы

Обычные пользователи сайта не могут добавлять данные через административный интерфейс. Необходимо, что на сайте тоже была такая возможность.

Чтобы сайт мог принимать и сохранять данные от пользователей, необходимо создать формы.

Для того чтобы отправить данные из браузера, нужно сформировать POST или GET запрос. Как правило, такие запросы формируются HTML-формами. Форма обозначается тэгом `<form>...</form>`, важно указать два атрибута: ***action***- адрес, куда форма отправит запрос и ***method***- тип запроса. Если в форме присутствует загрузка файлов, то необходим ещё один атрибут: ***enctype='multipart/form-data'***.

!GET и **POST** – единственные **HTTP** методы, которые используются для форм. Любой запрос, который может изменить состояние системы - например, который изменяет данные в базе данных - должен использовать **POST**. **GET** должен использоваться для запросов, которые не влияют на состояние системы. При **GET**, в отличие от **POST**, данные собираются в строку и передаются в URL.

Класс **FORM**

Наверняка, если вы ранее программировали, вам приходилось создавать формы. Для этого вы прописывали каждое поле отдельно в шаблоне, а потом, после отправки данных на сервер, делали проверку и обработку.

Но представьте, если полей будет несколько десятков. Сколько же тогда изменений придется вносить в html-файлы и представление.

Но, к счастью, в Django есть инструменты, которые позволяют создавать формы с существенно меньшим объемом кода и за меньшее количество времени. Речь идёт о классе *Form*.

Чтобы воспользоваться библиотекой, необходимо определить класс *Form*. Обычно данный класс размещают в файле *forms.py* приложения.

Большинство полей форм идентичны с полями моделей. Например, *CharField* модели соответствует *CharField* формы. Но некоторые поля отличаются. Например, поля модели *BigIntegerField*, *PositiveIntegerField*, *PositiveSmallIntegerField*, *SmallIntegerField* и сам *IntegerField* в форме представлены полем *IntegerField*; *CommaSeparatedIntegerField*, *NullBooleanField*, *TextField* в форме значатся, как *CharField*; *ForeignKey* – *ModelChoiceField* и *ManyToManyField* – *ModelMultipleChoiceField*.

Поле *ForeignKey* модели представлено полем формы *ModelChoiceField*, которое является обычным *ChoiceField*. Поле *ManyToManyField* модели представлено полем формы *ModelMultipleChoiceField*, которое является обычным *MultipleChoiceField*. Оба поля получают значения из *QuerySet*.

Каждое поле созданной формы имеет атрибуты.

Таблица 6.10. Основные параметры поля формы

Параметр	Описание
<i>required</i>	Если у поля модели есть <i>blank=True</i> , тогда к полю формы будет добавлено <i>required=False</i> , иначе – <i>required=True</i> .
<i>label</i>	Значением атрибута <i>label</i> поля будет значение поля <i>verbose_name</i> модели, причём первый символ этого значения будет преобразован в верхний регистр.
<i>help_text</i>	Значением атрибута <i>help_text</i> поля формы будет значение атрибута <i>help_text</i> поля модели.

Для лучшего понимания, как работают формы, рассмотрим пример авторизации пользователей на сайте. На самом деле вам не придется заниматься данным вопросом, Django сделал все за вас. Но об этом чуть позже.

```
from django.shortcuts import render, redirect
from django.contrib.auth import authenticate, login
from mypersonal.forms import *
# -*- coding:utf-8 -*-
from django import forms
class LoginForm(forms.Form):
    login = forms.EmailField(label='Логин')
    password = forms.CharField(label='Пароль', min_length=6,
max_length=20, widget=forms.PasswordInput(attrs={'placeholder': 'до
20 знаков', 'class': 'password_style'}))
```

Этот код создает класс *Form* с двумя полями: логин и пароль. Мы добавили русские названия поля в *<label>*. В данном случае логином будет являться email-адрес пользователя, на что указывает тип поля *EmailField*.

Второе поле – для ввода пароля от до 20 символов.

При добавлении поля на форму, Django использует стандартный виджет, наиболее подходящий к отображаемому типу данных. Но можно указать другой виджет для поля, в нашем случае – *PasswordInput*, который указывает на то, что поле предназначено для введения паролей. Также часто используют: *EmailInput*, *NumberInput*, *HiddenInput* для скрытых полей, *DateInput* для дат, *Textarea* для текстовой области, *CheckboxInput*, виджет выбора *Select* и множественного выбора *SelectMultiple*, *RadioSelect* – виджет выбора в виде списка радио кнопок, *CheckboxSelectMultiple* – для множественного выбора с отображением в виде checkbox, *FileInput* и другие.

Иногда хочется разнообразить отображение полей. Вам может потребоваться сделать больше строк для поля ввода комментария или

назначить имя виджету, чтобы он начал реагировать на особый CSS класс. Для этого надо использовать аргумент *Widget.attrs* при создании виджета. В примере выше мы добавили подсказку внутри поля и CSS класс.

Обработка формы

После нажатия на *submit* данные отправятся на url, указанный в *action*, который в свою очередь вызывает соответствующее представление.

```
def mysite_login(request):
    if request.method == 'POST':
        form = LoginForm(request.POST)
        if form.is_valid():
            username = form.cleaned_data['login']
            password = form.cleaned_data['password']
            user = authenticate(username=username, password=password)
            if user and user.is_active:
                login(request, user)
        return redirect('/')
    else:
        form = LoginForm()
    return render(request, 'login.html', {'form': form})
```

Когда мы впервые открываем страницу (без POST запроса), создаётся пустая форма, которая была передана в контекст шаблона для последующего рендеринга.

После нажатия на кнопку на форме, отправляется *POST* запрос, и представление создаёт форму с данными из запроса: *form = LoginForm(request.POST)*, то есть данные «привязываются» к форме. Если к запросу «привязаны» файлы, то необходимо дополнительно указать *request.FILES* в скобках в качестве источника данных файлов.

Далее вызываем метод *is_valid()* формы. Если *is_valid()* вернет *True*, мы можем найти проверенные данные в атрибуте *cleaned_data*. Мы можем сохранить эти данные в базе данных, или выполнить какие-то другие действия над ними, перед тем, как сделать редирект на другую страницу. Если *is_valid()* вернул *False*, снова рендерим шаблон, передав нашу форму. Форма содержит ранее отправленные данные и ошибки. Далее, форму можно повторно отредактировать и отправить.

Форма в шаблоне может выглядеть следующим образом:

```
<form action="/mysite_login/" method="post">
  {% csrf_token %}
  {{ form.as_p }}
  <input type="submit" value="Войти" />
</form>
```

! Django поставляется с защитой против Cross Site Request Forgeries. При отправке формы через POST с включенной защитой от CSRF вы должны использовать шаблонный тег *csrf_token*, как показано в предыдущем примере.

По умолчанию в HTML форма представляется в виде таблицы, т.е. каждое поле обернуто в *<tr></tr>*, здесь используется метод *Form.as_table()*. Но существует ряд других представлений: *Form.as_p()*, который представляет форму в виде последовательности тегов *<p></p>*, *Form.as_ul()*, который представляет форму в виде последовательности тегов **.

Если вы хотите изменить порядок или как-то отредактировать форму, то можете получить каждое поле отдельно через атрибут формы *{{ form.name_of_field }}*.

```
<form action="/mysite_login/" method="post">
  {% csrf_token %}
  <div class="field">
    {{ form.login.errors }}
    {{ form.login.label }}
```



```
        {{ form.login }}
</div>
<div class="field">
    {{ form.password.errors }}
    {{ form.password.label }}
    {{ form.password }}
</div>
<input type="submit" value="Войти" />
</form>
```

Дополнительная проверка полей формы

На данном этапе наша форма подвергается только встроенным проверкам: заполнены ли все поля, соответствуют ли введённые данные типу поля и так далее.

Но нам необходимо проверить, зарегистрирован ли данный пользователь в нашей системе. Для таких дополнительных проверок можно создать свои собственные правила проверки.

Лучше всего проверку делать в самих формах, а не выносить их в представление.

```
from django.contrib.auth import authenticate
...
def clean(self):
    cleaned_data = super(LoginForm, self).clean()
    if not self.errors:
        user = authenticate(username=cleaned_data['login'],
password=cleaned_data['password'])
        if user is None:
            self._errors["password"] = self.error_class(["Проверьте
правильность введённых данных"])
    return cleaned_data
```

Данный код проверяет, есть ли указанная в форме комбинация логина и пароля в базе. Если нет, то выводит сообщение об ошибке.

Теперь у нас есть полностью рабочая форма, созданная классом *Form*, с необходимыми проверками ошибок и обрабатываемая представлением.

Наборы форм

Иногда возникает необходимость в создании нескольких объектов за один раз. Например, вы можете позволить пользователю привязать множество файлов к конкретной задаче.

Для таких целей существуют наборы форм *formset*.

Набор форм — это абстрактный слой для работы с множеством форм на одной странице.

Создадим класс *FilesForm* для добавления файлов.

```
class FilesForm(forms.Form):  
    file = forms.FileField()
```

Для того чтобы создать набор форм вам потребуется импортировать *formset_factory* из *django.forms.formsets*.

```
from django.forms.formsets import formset_factory  
def add_files(request):  
    FormSet = formset_factory(FilesForm, extra = 3)  
    if request.method == "POST":  
        formset = FormSet(request.POST, request.FILES)  
        if formset.is_valid() and formset.has_changed():  
            for form in formset:  
                # какое-либо действие с файлом  
            return redirect('/')  
    else:  
        formset = FormSet()  
        return render(request, 'add_files.html', {'formset': formset})
```

Теперь у вас есть класс набора форм *FormSet*. Набор форм предоставляет возможность последовательно проходить по списку форм и отображать их как обычные формы.

Количество выводимых пустых форм управляется с помощью параметра *extra*. По умолчанию фабрика *formset_factory()* добавляет одну пустую форму. В данном примере *extra = 3* формам.

Шаблон будет выглядеть следующим образом:

```
<form method="post" action="" enctype="multipart/form-data">
    {% csrf_token %}
    {{ formset.management_form }}
    {% for form in formset %}
        {{ form }}
    {% endfor %}
    <input type="submit" value="Добавить" />
</form>
```

или можно сократить код:

```
<form method="post" action="" enctype="multipart/form-data">
    {% csrf_token %}
    {{ formset }}
    <input type="submit" value="Добавить" />
</form>
```

Создание форм из моделей

При разработке приложения, использующего базу данных, чаще всего вы будете работать с формами, введенные данные которых сохраняются в моделях. Например, вам может потребоваться создать форму, которая позволит добавлять задания в базу данных из системы (в модель *Task*).

По этой причине Django предоставляет вспомогательный класс ***ModelForm***, который позволит вам создать класс *Form* по имеющейся модели.

```
# -*- coding:utf-8 -*-
from django.forms import ModelForm, Textarea
from mypersonal.models import Task
class TaskModelForm(ModelForm):
```

```

class Meta:
    model = Task
    fields = ['type', 'task', 'text', 'file']
# переопределение виджета поля
widgets = {
    'name': Textarea(attrs={'cols': 80, 'rows': 20}),
}
# переопределение параметров
labels = {
    'task': ('Тема задачи'),
}
help_texts = {
    'task': ('Опишите задачу вкратце.'),
}
error_messages = {
    'task': {
        'max_length': ("Слишком длинная тема."),
    },
}

```

Рекомендуется явно указывать все поля, отображаемые в форме, используя параметр *fields*.

Самый простой способ указать поля - добавить все или исключить определенные. Данный способ не безопасен. Если вы, например, в дальнейшем добавите новые поля в модель, то эти поля автоматически появятся в шаблоне, даже если они не предназначены для редактирования пользователем.

Если же вы все же решили указать все поля модели, то пропишите в параметре *fields* специальное значение `__all__`.

Для исключения определённых моделей используйте атрибут *exclude*.

Поля, которые не определены в форме, не будут учитываться при вызове метода *save()*. Даже, если вы вручную добавите в форму исключенные поля.

Для обработки формы пропишем представление.

```
# Добавим новую задачу
def add_task(request):
    if request.method == "POST":
        form = TaskModelForm(request.POST, request.FILES)
        if form.is_valid():
            form.save()
            return redirect('/tasks/')
    else:
        form = TaskModelForm()
    return render(request, 'add_task.html', {'form': form})
```

Django будет препятствовать всем попыткам сохранить неполную модель. Таким образом, если модель требует заполнения обязательных полей и для них не предоставлено значение по умолчанию, то сохранить форму для такой модели не получится. Для решения этой проблемы вам потребуется создать экземпляр такой модели, передав ему начальные значения для обязательных, но незаполненных полей.

```
t = Task(status=True)
form = TaskModelForm(request.POST, request.FILES, instance=t)
form.save()
```

В качестве альтернативы, вы можете использовать *save(commit=False)* и вручную определить все необходимые поля. Метод вернёт объект, который ещё не был сохранён в базе данных.

```
if form.is_valid():
    form_for_save = form.save(commit=False)
    form_for_save.status = True
    form_for_save.save()
```

Данный код изменит поле «Статус», увеличив его на 1 день.

Каждая форма, созданная с помощью *ModelForm*, обладает методом *save()*. Этот метод создаёт и сохраняет объект в базе данных, используя для этого данные, введённые в форму. Класс, унаследованный от *ModelForm*, может принимать существующий экземпляр модели через именованный аргумент *instance*. Если такой аргумент указан, то *save()* обновит переданную модель. В противном случае, *save()* создаст новый экземпляр указанной модели.

```
# Изменить уже созданную задачу
def change_task(request, id):
    t = Task.objects.get(pk = id)
    if request.method == "POST":
        form = TaskModelForm(request.POST, request.FILES, instance =
t)
        if form.is_valid():
            form.save()
            return redirect('/tasks/')
    else:
        form = TaskModelForm(instance = t)
    return render(request, 'add_task.html', {'form': form})
```

Аналогично наборам обычных форм, Django представляет ряд расширенных классов наборов форм, которые упрощают взаимодействие с моделями Django. В данном курсе они рассмотрены не будут.

9. Практическое задание «Работа с формами для добавления и изменения достижений»

1. В предыдущем задании вы добавляли достижения через административный интерфейс. Усовершенствуйте систему так, чтобы вы могли добавлять достижения через форму на самом сайте.

Примечание: используйте класс ModelForm.

2. Создайте возможность изменения и удаления ранее добавленного достижения.

7.5.10. Встроенная система аутентификации пользователей

В библиотеке Django вы можете найти встроенную систему аутентификации пользователей, которая содержит функции аутентификации и авторизации.

Одними из главных объектов любого сайта, где существует взаимодействие с клиентом, заказчиком и т.п., являются объекты *User*, которые представляют пользователя сайта и используются для проверки прав доступа, регистрации пользователей, ассоциации данных с пользователями.

Таблица 6.11. Основные поля объектов *User*

Поле	Описание
<i>username</i>	Логин. Обязательное поле. Максимум 30 символов. Можно использовать только буквы, цифры и символ подчёркивания.
<i>first_name</i>	Имя пользователя. Необязательное поле. Максимум 30 символов.
<i>last_name</i>	Фамилия пользователя. Необязательное поле. Максимум 30 символов.
<i>email</i>	Электронная почта. Необязательное поле.
<i>password</i>	Пароль. Обязательное поле. Хэш и метаданные пароля (Django не хранит пароль в открытом виде).
<i>is_staff</i>	Определяет, входит ли пользователь в привилегированную группу. Булево значение.
<i>is_active</i>	Определяет, активен ли данный аккаунт. Булево значение. Установите в <i>False</i> для блокировки аккаунта.
<i>is_superuser</i>	Определяет, входит ли пользователь в

	группу администраторов. Булево значение.
<i>last_login</i>	Дата и время последней аутентификации. По умолчанию устанавливается текущее время.
<i>date_joined</i>	Дата и время создания аккаунта. По умолчанию указывается дата и время создания аккаунта.

. Таблица 6.12. Основные методы объектов *User*

Метод	Описание
<i>get_username()</i>	Возвращает имя пользователя
<i>get_full_name()</i>	Возвращает значения <i>first_name</i> и <i>last_name</i> с пробелом между ними.
<i>check_password(pwd)</i>	Возвращает <i>True</i> при совпадении переданного пароля с имеющимся.
<i>get_all_permissions()</i>	Возвращает в виде строк список всех прав, которыми обладает пользователь.
<i>is_authenticated()</i>	Метод определяет, был ли текущий пользователь авторизован. Метод не назначает никаких прав и не проверяет активность пользователя. Он просто является индикатором аутентификации пользователя.
<i>is_anonymous()</i>	Возвращает <i>True</i> только для объектов <i>AnonymousUser</i> . ! Рекомендуется использовать метод <i>is_authenticated()</i> .
<i>set_password(pwd)</i>	Назначает пароль для пользователя, переданный в виде строки. Метод шифрует пароль. Метод не сохраняет объект <i>User</i> .

Создание пользователей

Самый простой способ создать пользователя — использовать метод `create_user()`.

```
from django.contrib.auth.models import User
def add_user(request):
    user = User.objects.create_user('Иван', 'ivan@gmail.com', 'password')
    return HttpResponse("Добавлен новый пользователь")
```

Django не хранит пароль в открытом виде, только его хеш. Поэтому вы не сможете его изменить обычным способом. Для этого необходимо вызвать специальный метод `set_password()`.

```
user = User.objects.get(username='Иван')
user.set_password('new_password')
user.save()
```

Вход и выход из системы

В Django имеется ряд встроенных функций для обработки входа `login()` и выхода пользователя `logout()`. Для того, чтобы воспользоваться ими, необходимо добавить их в `urls.py` из `django.contrib.auth`.

```
from django.contrib.auth.views import login, logout
url(r'^login/$', login, {'template_name': 'login.html'}),
url(r'^logout/$', logout, {'next_page': '/'})
```

`login()` принимает ряд аргументов:

- **template_name**: имя шаблона с формой для авторизации. По умолчанию: `registration/login.html`;
- **redirect_field_name**: URL-адрес для перенаправления после успешного входа в систему;
- **authentication_form**: форма аутентификации. По умолчанию: `AuthenticationForm`;
- **current_app**: текущее приложение;
- **extra_context**: словарь данных, которые будут переданы в шаблон.

Форма, используемая в шаблоне, должна содержать поля `username` и `password`. Чтобы изменить URL-адрес, укажите его в скрытом поле `next` в атрибуте `value`: `<input type="hidden" name="next" value="{{ next }}" />`. Также это значение можно указать в GET-параметре к представлению `login`, тогда оно будет автоматически добавлено в контекст в виде переменной `next`.

Функции `logout()` принимает те же аргументы, что и `login()` кроме `authentication_form`. Дополнительным аргументом функции является `next_page`, в котором указывается URL страницы, куда нужно перейти после выхода из системы.

10. Практическое задание «Аутентификация пользователей»

Ваш сайт – это не просто сайт-визитка. Это полноценный сайт с базой данных. Но чего-то не хватает... А именно, гостей, которые бы могли оценить ваши достижения.

1. Добавьте возможность регистрации новых пользователей на сайте.
2. Добавьте навигационную панель в шапку сайта со ссылками Вход/Выход (если авторизован, то выход, если не авторизован, то вход, соответственно).

Примечание: Вынесите аутентификацию пользователей в отдельное приложение.

3. Необходимо, что авторизованные пользователи могли оценить ваши достижения (нравится/не нравится).
4. Добавьте страницу для отзывов.

Список использованных источников

1. Лутц М. Программирование на Python / Марк Лутц; пер. с англ. А. Киселева. – М.: Символ-Плюс, 2011. – 992 с.
2. Хахаев И. А. Практикум по алгоритмизации и программированию на Python: / И. А. Хахаев – М.: Альт Линукс, 2011. – 126 с.
3. Доусон М. Програмируем на Python / М.Доусон – СПб.: Питер, 2014. – 416 с.
4. Сайт Django [Электронный ресурс]. – Режим доступа: <https://www.djangoproject.com>, свободный.
5. Головатый А. Django. Подробное руководство / А. Головатый, Дж. Каплан-Мосс, пер. с англ. А. Киселева. – М.: Символ-Плюс, 2010. – 560 с.

Приложение 1. Вопросы для тестирования

1. Какие характеристики можно отнести к языку программирования Python?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. удобен для встраивания в проекты на C/C++
2. многоплатформный
3. большая стандартная библиотека модулей

2. Какие парадигмы и стили программирования поддерживает Python?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. модульное программирование
2. императивное программирование
3. структурный стиль
4. логистическое программирование

3. Что будет выведено следующей программой:

```
a = "A"
```

```
b = "B"
```

```
b = b + a
```

```
print a + b
```

(Отметьте один правильный вариант ответа.)

1. сообщение об ошибке в третьей строке
2. ABA
3. BA
4. AB

4. Что выведет следующая программа:

```
S = 0
```

```
for i in range(10, 2, -1):
```

```
if i % 2 == 0:
```

```
    S = S + i
```

```
printS
```

(Отметьте один правильный вариант ответа.)

1. 30
2. 28
3. 20
4. 0

5. Сколько элементов будет содержать словарь D (то есть, чему будет равно len(D)) после выполнения следующего кода:

```
D = { }
```

```
D[1], D[2], D[3] = "ABB"
```

```
D[0], D[1] = "AB"
```

(Отметьте один правильный вариант ответа.)

1. 2
2. 4
3. 3 и произойдет ошибка в 3-й строке
4. 3

6 .Какого типа значение получится в результате вычисления следующего выражения:

```
(r'\u0432')
```

(Отметьте один правильный вариант ответа.)

1. unicode (Unicode-строка)
2. tuple (кортеж)
3. это синтаксическая ошибка
4. str (строка)

7 .Что будет получено в результате вычисления следующего выражения:

```
0 < [1, 4][1] < 3 or None
```

(Отметьте один правильный вариант ответа.)

1. 1
2. 0
3. синтаксическая ошибка
4. None

8. Какие характеристики можно отнести к языку программирования Python?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. для быстрой разработки приложений
2. интерпретируемый
3. использующий препроцессор для макроподстановок
4. с динамической типизацией

9. Что будет выведено следующей программой:

```
a = "AB"
```

```
b = "BC"
```

```
print "%sa, b" % a, b
```

(Отметьте один правильный вариант ответа.)

1. "AB", "BC"a, b
2. ABa, b BC
3. ('AB', 'BC')a, b
4. (AB, BC)a, b

10. Что выведет следующая программа:

```
S = 0
```

```
for i in range(1, 10):
```

```
    if i % 2 == 0:
```

```
        S = S + i
```

```
print S
```

(Отметьте один правильный вариант ответа.)

1. 14

- 2. 10
- 3. 20
- 4. 12

11. Сколько элементов будет содержать список L (то есть, чему будет равно `len(L)`) после выполнения следующего кода:

```
L = []
```

```
L.append([1,2,3])
```

```
L.insert(1, "abc")
```

```
del L[0][0]
```

(Отметьте один правильный вариант ответа.)

- 1. 3
- 2. произойдет ошибка
- 3. 1
- 4. 2

12. Какого типа значение получится в результате вычисления следующего выражения:

```
(" ")
```

(Отметьте один правильный вариант ответа.)

- 1. str (строка)
- 2. это синтаксическая ошибка
- 3. unicode (Unicode-строка)
- 4. tuple (кортеж)

13. Что будет получено в результате вычисления следующего выражения:

```
(0 < 5 <= 3) and (0 / 0)
```

(Отметьте один правильный вариант ответа.)

- 1. False (или 0)
- 2. True (или 1)
- 3. синтаксическая ошибка

4. будет возбуждено исключение `ZeroDivisionError` (деление на нуль)

14. Что произойдет со старыми объектами модуля, используемыми в программе, при его перезагрузке по `reload()` (после изменения на диске):

```
import mdl
a = mdl.a
b = mdl.b()
reload(mdl)
```

(Отметьте один правильный вариант ответа.)

1. объекты (`a`, `b`) изменятся в соответствии с новыми определениями
2. изменятся только классы, функции и т.п. (`a`)
3. изменится только `mdl`
4. имена из модуля (`mdl.a`, `mdl.b`) будут ссылаться на другие объекты. Старые объекты (`a`, `b`) не изменятся

15. Какие встроенные функции служат для работы с атрибутами объекта?

(Отметьте один правильный вариант ответа.)

1. `type()`, `intern()`, `del`
2. `staticmethod()`, `classmethod()`, `property()`
3. `callable()`, `super()`
4. `hasattr()`, `getattr()`, `setattr()`, `delattr()`

16. Из какого модуля будет работать функция `split()` в следующем примере:

```
from re import *
from string import *
split('a', 'b')
```

(Отметьте один правильный вариант ответа.)

1. из string
2. возникнет ошибка (конфликт имен)
3. из re.string
4. из re

17. В каком модуле нужно искать функции, помогающие тестировать программу?

(Отметьте один правильный вариант ответа.)

1. profile
2. dictutils
3. pdb
4. unittest

18. Как мог бы называться стандартный модуль Python для работы с протоколом IMAP?

(Отметьте один правильный вариант ответа.)

1. libimap
2. imap_module
3. imaplib
4. IMAPLibrary

19. Какими из перечисленных ниже способов можно получить случайный элемент последовательности lst с помощью модуля random?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. random.random(lst)
2. lst[random.randrange(len(lst))]
3. random.choice(lst)
4. random.shuffle(lst); lst[0]

20. Какими операторами можно импортировать модуль?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. `import`
2. `from-import`
3. `imp`
4. `exec`

21. С помощью какой функции можно организовать цикл с параметром (`for`)?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. `reload()`
2. `id()`
3. `range()`
4. `xrange()`

22. Какие новые имена появятся в текущем модуле после выполнения следующего кода:

```
import re
```

```
from re import compile
```

(Отметьте один правильный вариант ответа.)

1. только имена `re` и `compile`
2. все имена из `re` (импорт `compile` был лишним)
3. нельзя одновременно делать `import` и `from-import`
4. только имя `compile`

23. С помощью каких модулей можно загрузить файл с FTP-сервера?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. `cgi`
2. `mimetools`

3. urllib
4. ftplib

24. Для чего нужны функции модуля `gettext`?

(Отметьте один правильный вариант ответа.)

1. для чтения строки со стандартного ввода
2. для обеспечения интернационализации программы
3. для показа строки ввода на экране и ввода текста от пользователя
4. для получения текста от пользователя

25. Что из нижеперечисленного естественно для реализации в функциональном стиле?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. рекурсия
2. циклы
3. итераторы

26. Какие функции Python 2.x позволяют организовать обработку сразу двух и более последовательностей?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. функция `map()`
2. функция `itertools.repeat()`
3. функция `zip()`
4. функция `filter()`

27. Начало определения функции `f` выглядит так:

```
def f(a, b, c=None, d="0"):
```

Какие из следующих вариантов вызова не приведут к ошибке на этапе присваивания фактических параметров формальным?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. Вариант 1 f()
2. Вариант 2 f(1, 2, d=3, c=4)
3. Вариант 3 f(1, 2)
4. Вариант 4 f(1, d=3)
5. Вариант 5 f(1, 2, d=3)
6. Вариант 6 f(1, 2, 3, 4)

28. Сколько списков занимающих много памяти задействовано в следующей программе:

```
for i in itertools.izip(xrange(10**6), xrange(10**6)):  
    pass
```

(Отметьте один правильный вариант ответа.)

1. 3
2. 0
3. 1
4. 2

29. Какая из перечисленных функций имеет побочные эффекты:

```
lst = []
```

```
def A(lst, x):  
    return lst + [x]
```

```
def B(x):  
    lst.append(x)  
    return lst
```

```
def C(lst, x):  
    return lst.count(x)
```

(Отметьте один правильный вариант ответа.)

1. C
2. B
3. A

30. Имеется следующий генератор для слияния двух отсортированных последовательностей:

```
def merge(a1, a2):  
    a1 = list(a1)  
    a2 = list(a2)  
    while a1 or a2:  
        if a1 and (not a2 or a1[0] < a2[0]):  
            r = a1  
        else:  
            r = a2  
        yield r[0]  
    del r[0]
```

Какие ошибки или особенности имеет эта программа?

(Отметьте один правильный вариант ответа.)

1. ошибок нет
2. генератор будет портить переданные ему списки
3. в последней строке каждый раз удаляется элемент из временного списка, а не из a1 или a2: генератор зациклится
4. генератор оставит за собой временный список, так как del происходит после yield

31. Как определить функцию в Python?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. заданием списка строк исходного кода
2. с помощью lambda-выражения
3. с помощью оператора import
4. с помощью оператора def

32. Какая встроенная функция Python лучше всего подходит для цепочечных вычислений (в частности, вычислений значения многочлена по схеме Горнера)?

(Отметьте один правильный вариант ответа.)

1. `reduce()`
2. `chain()`
3. `map()`
4. `filter()`

33. Начало определения функции `f` выглядит так:

```
def f(*p, **k):
```

Какие из следующих вариантов вызова не приведут к ошибке на этапе присваивания фактических параметров формальным?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. Вариант 1 `f(1, 2, d=3, c=4)`
2. Вариант 2 `f(1, d=2, 3)`
3. Вариант 3 `f(1, 2, 3, 4)`
4. Вариант 4 `f(1, 2)`
5. Вариант 5 `f(1, 2, d=3)`

34. Какие из получаемых в следующем фрагменте кода объектов являются итераторами?

```
def gen(N):
```

```
    for i in xrange(N):
```

```
        yield i
```

```
lst = [1, 2, 3, 4]
```

```
xr = xrange(12)
```

```
g = gen(10)
```

```
en = enumerate(lst)
```

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. en
2. gen
3. lst
4. g
5. xr

35. Какие из перечисленных функций имеют побочные эффекты:

```
def A(lst):  
    return lambda x: lst + [x]
```

```
def B(x):  
    return lambda lst: lst + [x]
```

```
def C(x, cache={}):  
    return cache.setdefault(x, lambda lst: lst + [x])
```

(Отметьте один правильный вариант ответа.)

1. только C
2. никакие
3. A, B, C
4. только B и C

36. Имеется следующий генератор для слияния двух отсортированных последовательностей:

```
def merge(a1, a2):  
    i1 = iter(a1)  
    i2 = iter(a2)  
    while i1 or i2:  
        if i1 and (not i2 or i1[0] < i2[0]):  
            r = i1  
    else:
```

```
r = i2
```

```
yieldr.next()
```

Какие ошибки или особенности имеет эта программа?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. генератор будет портить переданные ему списки
2. индексирование (`i1[0]`, `i2[0]`) неприменимо к итераторам
3. ошибок нет
4. длина итератора в общем случае неизвестна: ошибка в строке с условием цикла

37. Что из перечисленного правильно характеризует отличия функций в математике от функций в языках программирования?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. в математике функции имеют строго оговоренные множества определения, в программировании это невозможно
2. в математике функции не имеют побочных эффектов
3. числовые функции языка программирования — часто лишь приближение математической функции
4. в программировании функции всегда имеют побочные эффекты

38. Выберите правильные (с точки зрения теории ООП) утверждения:

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. все объекты одного типа могут принимать одни и те же сообщения
2. каждый объект имеет тип
3. все объекты одного типа принадлежат одному классу

39. Сколько общедоступных методов будет иметь экземпляр класса ABC и что возвратит вызов метода `a()`?


```
class A(object):
    def a(self): return 'a'
class B(object):
    def b(self): return 'b'
class C(object):
    def c(self): return 'c'
```

```
class AB(A, B):
    def a(self): return 'ab'
class BC(B, C):
    def a(self): return 'bc'
class ABC(AB, B, C):
    def a(self): return 'abc'
```

(Отметьте один правильный вариант ответа.)

1. 3, возвратит a
2. 3, возвратит ab
3. 3, возвратит abc
4. 5, возвратит a

40. Имеются следующие определения:

```
class A:
    def am(self):
        print "am"
```

```
class B:
    def bm(self):
        print "bm"
```

```
a = A()
```

```
b = B()
```

Какой из фрагментов кода содержит ошибки?

(Отметьте один правильный вариант ответа.)

1. A.am = b.bm; a.am()
2. A.am = B.bm; a.am()
3. a.am = b.bm; a.bm()
4. a.am = b.bm; a.am()

41. Укажите набор атрибутов, которые считаются общедоступными, для экземпляров следующего класса:

```
class Example:
    def __init__(self, x, y):
        xy = x, y
        self.position = xy
        self._length = self.__len(x, y)
    def __len(self, x, y):
        return abs(x) + abs(y)
    def getlen(self):
        return self._length
```

(Отметьте один правильный вариант ответа.)

1. getlen, position
2. position
3. getlen, _length, position
4. getlen, _length, position, __len, xy

42. Как называется отношение, которое имеют следующие два класса:

```
class A(object):
    def __init__(self, x):
        self._mydata = B(x)
```

```
class B(object):
    def __init__(self, x):
        self._mydata = x
```

(Отметьте один правильный вариант ответа.)

1. метакласс. А является метаклассом для В

2. метакласс. В является метаклассом для А
3. наследование. А получается наследованием В
4. ассоциация. Экземпляр А содержит ссылки на экземпляры В

43. Какую роль играет xx в Python-программе:

```
class A:
```

```
...
```

```
class B:
```

```
...
```

```
...
```

```
a = A()
```

```
b = B()
```

```
c = xx(a, b)
```

```
b1 = B()
```

```
c1 = xx(b1, b)
```

(Отметьте один правильный вариант ответа.)

1. мультиметод
2. класс
3. функция
4. метод

44. Класс имеет методы `__iter__()` и `next()`. О чем это говорит и как пользоваться этим методом?

```
class A:
```

```
#...
```

```
def __iter__(self):
```

```
#...
```

```
def next(self):
```

```
#...
```

```
a = A(1, 2, 3)
```

(Отметьте один правильный вариант ответа.)

1. итератор. Пользоваться можно так: `for i in a: print i`
2. генератор. Пользоваться можно так: `for i in a(): print i`
3. нет особого названия. Пользоваться можно так: `print a.next()`
4. последовательность. Пользоваться можно так: `print a[2]`

45. Начало определения функции `f` выглядит так:

```
def f(*p, **k):
```

Какие из следующих вариантов вызова не приведут к ошибке на этапе присваивания фактических параметров формальным?

(Ответ считается верным, если отмечены все правильные варианты ответов.)

1. Вариант 1 `f(1, 2, d=3, c=4)`
2. Вариант 2 `f(1, d=2, 3)`
3. Вариант 3 `f(1, 2, 3, 4)`
4. Вариант 4 `f(1, 2)`
5. Вариант 5 `f(1, 2, d=3)`