Python. Введение в программирование

Курс

Опубликован: https://younglinux.info/python/course

Автор: Светлана Шапошникова (plustilino)

Версия: февраль 2024 года

PDF-версия содержит текст уроков курса и решения с пояснениями к практическим работам
+
8 дополнительных уроков

Курс **"Python. Введение в программирование"** рассчитан на старшеклассников и всех желающих познакомиться с программированием. В курсе рассматриваются основные типы данных, принципы и понятия структурного программирования.

Выбор Python обусловлен такими его преимуществами как ясность кода и быстрота реализации на нем программ.

Курс рассчитан примерно на 30 часов.

Основной целью курса является знакомство с программированием, формирование базовых понятий структурного программирования, подготовка к последующему изучению объектно-ориентированного программирования.

Текущая версия курса: февраль 2024 г.

Уроки базовой части курса в кратком изложении на YouTube.

Содержание

ДОПОЛНИТЕЛЬНЫЕ УРОКИ

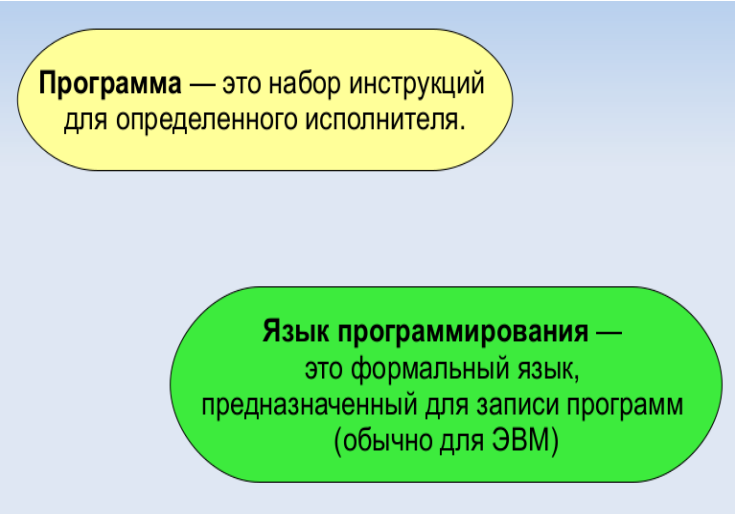
Урок 1. Понятие программы. Краткая история языков программирования. Трансляторы

Программа. Язык программирования

Программу можно представить как набор последовательных команд, то есть *алгоритм*, для объекта, то есть *исполнителя*, который должен их выполнить для достижения определенной цели.

Так можно условно запрограммировать человека, составив для него инструкцию "как приготовить оладьи", а он начнет ей следовать. При этом инструкция, она же программа, для человека будет написана на так называемом естественном языке, например, русском или английском.

Все же программируют не людей, а вычислительные машины, используя при этом специальные языки. Необходимость в особых языках связана с тем, что машины не в состоянии "понимать" наши, то есть человеческие, естественные для нас. Инструкции для машин пишут на *языках программирования*, которые *характеризуются формальностью*, то есть синтаксической однозначностью (например, в них нельзя менять местами определенные слова) и ограниченностью (имеют строго определенный набор слов и символов).



Основные этапы исторического развития языков программирования

Первые программы писались на **машинном языке**, так как для ЭВМ того времени еще не существовало развитого программного обеспечения, а машинный язык – это единственный способ взаимодействия с аппаратным обеспечением компьютера, так называемым "железом".

Каждую команду машинного языка непосредственно выполняет то или иное электронное устройство. Данные и команды записывали в цифровом виде, например, в шестнадцатеричной

или двоичной системах счисления. Человеку воспринимать написанную таким образом программу сложно. Кроме того, даже небольшая программа состояла из множества строк кода. Ситуация осложнялась еще и тем, что каждая вычислительная машина понимает лишь свой машинный язык.

Людям, в отличие от машин, более понятны слова, чем наборы цифр. Стремление человека оперировать словами, а не цифрами привело к появлению **ассемблеров**. Это языки, в которых вместо численного обозначения команд и областей памяти используются словесно-буквенные.

Однако машина по-прежнему не может понимать слова. Необходим какой-нибудь переводчик на ее родной машинный язык. Поэтому, начиная со времен ассемблеров, под каждый язык программирования создаются трансляторы – специальные программы, преобразующие программный код с языка программирования в машинный код. Ассемблеры на сегодняшний день продолжают использоваться. В системном программировании с их помощью создаются низкоуровневые интерфейсы операционных систем, компоненты драйверов.

После ассемблеров наступил расцвет языков так называемого **высокого уровня**. Для них потребовалось разрабатывать более сложные трансляторы, так как языки высокого уровня куда больше удобны для человека, чем для вычислительной машины.

В отличие от ассемблеров, которые остаются привязанными к своим типам машин, языки высоко уровня *обладают переносимостью*. Это значит, что, написав один раз программу, программист без последующего редактирования может выполнить ее на любом компьютере, если на нем установлен соответствующий транслятор. Программа-транслятор для данной ЭВМ при трансляции исходного кода сама адаптирует его под эту ЭВМ.

Следующим значимым шагом было появление **объектно-ориентированных языков**, что в первую очередь связано с усложнением разрабатываемых программ. С помощью таких языков программист как бы управляет виртуальными объектами. Мыслить в рамках объектов-сущностей, описывать их взаимодействие, обобщать объекты в классы и устанавливать между ними наследственные связи, – все это делает программу по-своему похожей на реальный мир, на то, как его воспринимает человек.

На сегодняшний день в большинстве случаев реализация крупных проектов осуществляется с помощью объектно-ориентированных возможностей языков. Хотя существуют и другие современные парадигмы программирования, поддерживаемые другими или теми же языками.

Разнообразие языков программирования

В настоящее время существует множество различающихся и похожих между собой языков программирования. Причина такого явления становится понятна, если представить то количество и разнообразие задач, которые на сегодняшний день решается с помощью вычислительной техники. Для решения разных задач требуются разные инструменты, то есть разные языки и подходы к программированию.

Разработка новых языков программирования, обладающих теми или иными преимуществами, велась как в прошлом, так и ведется сейчас. Эволюционируют, подстраиваясь под запросы нового времени, и старые языки программирования.

Все многообразие языков можно классифицировать по разным критериям. Например, по типу решаемых задач (языки системного или прикладного назначения, языки для web-разработки, организации баз данных, разработки мобильных приложений). Среди наиболее популярных на сегодняшний день можно отметить Java, C, C++, C#, JavaScript, PHP, в том числе Python, изучению базовых основ которого посвящен данный курс.

Трансляция

Ранее было сказано, что для перевода кода с языка программирования высокого уровня на машинный язык требуется специальная программа – транслятор.

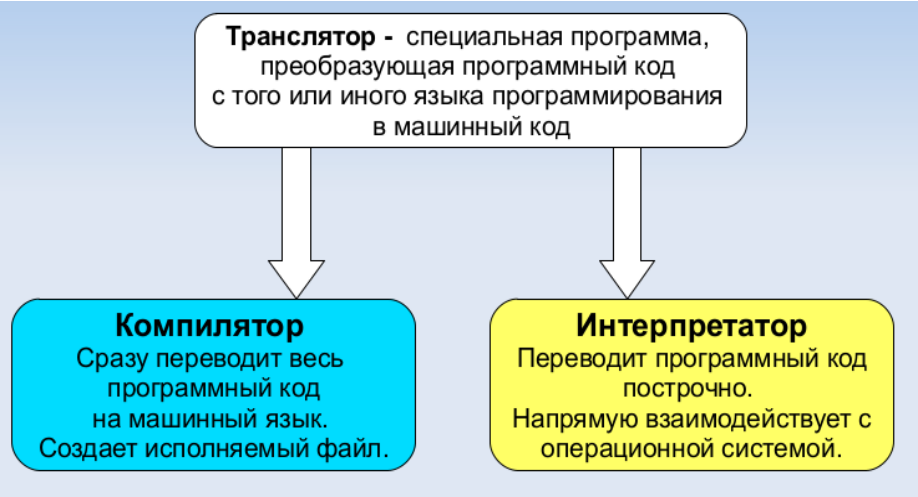
Заложенный в транслятор алгоритм такого перевода сложен. При этом существует два основных способа трансляции – **компиляция** программы или ее **интерпретация**.

При компиляции весь *исходный программный код* (тот, который пишет программист) сразу переводится в машинный. Создается так называемый отдельный *исполняемый файл*, который никак не связан с исходным кодом. Выполнение исполняемого файла обеспечивается операционной системой (ОС). После того как получен исполняемый файл, для его чтения транслятор уже не нужен.

При интерпретации выполнение кода происходит последовательно (условно можно сказать, строка за строкой). Грубо говоря, операционная система взаимодействует с интерпретатором, а не с файлом, содержащим программный код. Интерпретатор же, прочитав очередную часть исходного кода, переводит его в машинный (или не совсем машинный, но "понятный" для ОС) и "отдает" его ОС. ОС исполняет этот код и ждет следующей "подачки" от интерпретатора. Питон именно такой язык. Он интерпретируемый язык программирования.

Выполнение откомпилированной программы происходит быстрее, так как она представляет собой готовый машинный код. Однако на современных компьютерах снижение скорости выполнения при интерпретации обычно не заметно. Кроме того, интерпретируемые языки обладают рядом преимуществ, среди которых отсутствие подготовительных действий для

исполнения программы, что может быть важным для тех, кто только начинает изучать программирование.



Урок 2. Знакомство с Python

Краткая историческая справка

Язык программирования Python был создан к 1991 году голландцем Гвидо ван Россумом.

Свое имя – Пайтон (или Питон) – получил от названия телесериала, а не пресмыкающегося.

После того как Россум разработал язык, он выложил его в Интернет, где сообщество программистов присоединилось к его улучшению.

Python активно развивается и сейчас. Часто выходят новые версии. Раньше поддерживались две отдельные ветки языка: Python 2.x и Python 3.x. Здесь английской буквой "x" обозначается конкретный релиз. Между вторым и третьим Питоном есть небольшая разница. В настоящее время поддержка Python 2 прекращена.

Официальный сайт языка – https://www.python.org.

Особенности языка Python

Python – интерпретируемый язык программирования. Это значит, что исходный код частями преобразуется в машинный в процессе его чтения специальной программой – интерпретатором.

Python характеризуется ясным синтаксисом. Читать код на нем легче, чем на других языках программирования, так как в Питоне мало используются такие вспомогательные синтаксические элементы как скобки, точки с запятыми. С другой стороны, **правила языка заставляют**

программистов делать отступы для обозначения вложенных конструкций. Понятно, что хорошо оформленный текст с малым количеством отвлекающих элементов читать и понимать легче.

Python – это полноценный во многом универсальный язык программирования, используемый в различных сферах. Основная, но не единственная, поддерживаемая им парадигма, – объектно-ориентированное программирование. Однако в данном курсе мы только упомянем об объектах, а будем изучать структурное программирование, так как оно является базой. Без знания основных типов данных, ветвлений, циклов, функций нет смысла изучать более сложные парадигмы, так как в них все это используется.

Интерпретаторы Python распространяется свободно на основании лицензии совместимой с GNU General Public License.

Дзен Питона

Если интерпретатору Питона дать команду `import this`, то выведется так называемый "Дзен Питона", иллюстрирующий идеологию и особенности данного языка. Понимание смысла этих постулатов в приложении к программированию придет тогда, когда вы освоите язык в полной мере и приобретете опыт практического программирования.

1.	Beautiful is better than ugly.	Красивое лучше уродливого.
2.	Explicit is better than implicit.	Явное лучше неявного.
3.	Simple is better than complex.	Простое лучше сложного.
4.	Complex is better than complicated.	Сложное лучше усложнённого.
5.	Flat is better than nested.	Плоское лучше вложенного.
6.	Sparse is better than dense.	Разрежённое лучше плотного.
7.	Readability counts.	Удобочитаемость важна.
8.	Special cases aren't special enough to break the rules.	Частные случаи не настолько существенны, чтобы нарушать правила.
9.	Although practicality beats purity.	Однако практичность важнее чистоты.
10.	Errors should never pass silently.	Ошибки никогда не должны замалчиваться.
11.	Unless explicitly silenced.	За исключением замалчивания, которое задано явно.
12.	In the face of ambiguity, refuse the temptation to guess.	В случае неоднозначности сопротивляйтесь искушению угадать.

13.	There should be one – and preferably only one – obvious way to do it.	Должен существовать один – и, желательно, только один – очевидный способ сделать это.
14.	Although that way may not be obvious at first unless you're Dutch.	Хотя он может быть с первого взгляда не очевиден, если ты не голландец.
15.	Now is better than never.	Сейчас лучше, чем никогда.
16.	Although never is often better than *right* now.	Однако, никогда чаще лучше, чем прямо сейчас.
17.	If the implementation is hard to explain, it's a bad idea.	Если реализацию сложно объяснить – это плохая идея.
18.	If the implementation is easy to explain, it may be a good idea.	Если реализацию легко объяснить – это может быть хорошая идея.
19.	Namespaces are one honking great idea – let's do more of those!	Пространства имён – прекрасная идея, давайте делать их больше!

Как писать программы на Python

Интерактивный режим

Грубо говоря, интерпретатор выполняет команды построчно. Пишешь строку, нажимаешь Enter, интерпретатор выполняет ее, наблюдаешь результат.

Это удобно, когда изучаешь особенности языка или тестируешь какую-нибудь небольшую часть кода. Ведь если работать на компилируемом языке, пришлось бы сначала создать файл с кодом на исходном языке программирования, затем передать его компилятору, получить от него исполняемый файл, только потом выполнить программу и оценить результат. К счастью, даже в случае с компилируемыми языками все эти действия может взять на себя специально предназначенная для того или иного языка среда разработки, если вы используете ее.

В операционных системах на базе ядра Linux можно программировать на Python в интерактивном режиме с помощью приложения «Терминал», в котором работает командная оболочка Bash. Здесь, чтобы запустить интерпретатор, надо выполнить команду `python3` (может быть просто `python`).

```
pl@desk:~$ python3
Python 3.10.12 (main, Nov 20 2023, 15:14:05) [GCC 11.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

В данном случае запустилась версия 3.10.12. Первое число 3 указывает на то, что это интерпретатор для языка программирования Python 3. Последняя строка с тремя угловыми скобками (`>>>`) – это приглашение для ввода команд.

Для операционных систем семейства Windows надо скачать интерпретатор с официального сайта языка (https://www.python.org/downloads/windows/). После установки он будет запускаться по ярлыку. Использовать командную оболочку здесь не требуется.

Возможности Python позволяют использовать его как калькулятор. Поскольку команды языка мы не изучали, это хороший способ протестировать интерактивный ввод команд.

```
>>> 2 + 5
7
>>> 3**2
9
>>> 5 / 4
1.25
>>> (78 - 32) * (21 + 110)
6026
```

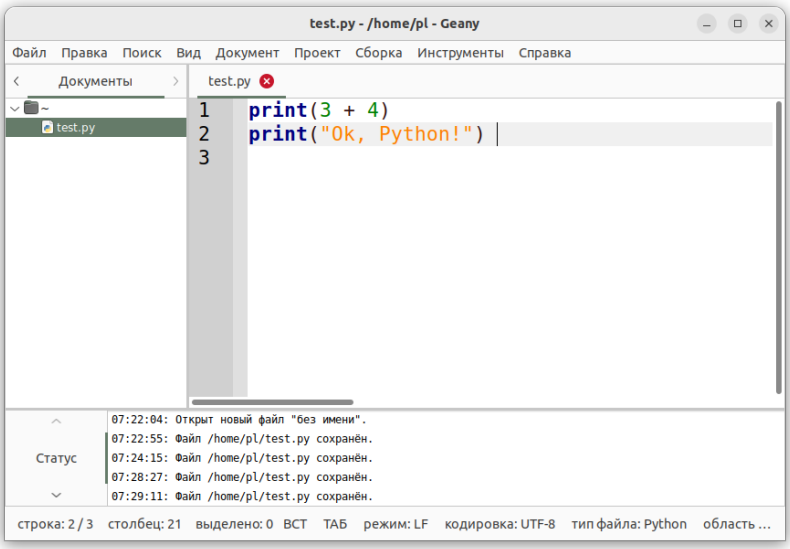
Бывает, что в процессе ввода была допущена ошибка или требуется повторить ранее используемую команду. Чтобы заново не вводить строку, в консоли можно прокручивать историю команд, используя для этого стрелки вверх и вниз на клавиатуре. В среде IDLE (в Windows) для этого используются сочетания клавиш (скорее всего `Alt + N` и `Alt + P`).

Чтобы выйти из интерактивного режима, следует ввести команду `exit()`.

Создание скриптов

Несмотря на удобства интерактивного режима, чаще всего необходимо сохранить исходный программный код для последующего выполнения и использования. В таком случае подготавливаются файлы, которые передаются затем интерпретатору на исполнение. Файлы с кодом на Python обычно имеют расширение **.py*.

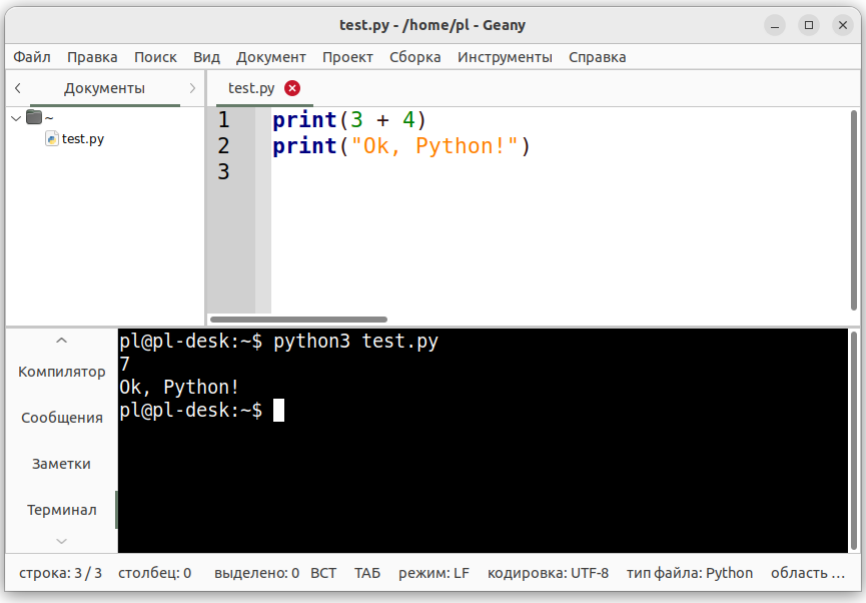
Существует множество сред разработки (IDE), в том числе созданные для программирования почти исключительно на Python. Примером такой среды является PyCharm. Однако подойдет и любой "легковесный" текстовый редактор с подсветкой синтаксиса, например, Geany или Sublime Text.



Здесь создается и сохраняется файл с кодом. Далее его можно запустить на выполнение через терминал. При этом сначала указывается интерпретатор (в данном случае `python3`), потом имя файла (если файл находится в другом каталоге, то указывается с адресом, или надо перейти в этот каталог с помощью команды **`cd`** оболочки Bash).

```
pl@desk:~$ python3 test.py
7
Ok, Python!
```

При этом в редакторе может быть установлен свой встроенный "Терминал", что упрощает работу.



Также в Geany можно просто нажать F5, что отправит файл на исполнение (терминал откроется сам, после выполнения программы и нажатия Enter закроется). Однако при этом должен быть правильно настроен вызываемый интерпретатор (пункт меню Сборка → Установить команды сборки).

В Windows подготовить файлы можно в той же среде IDLE. Для этого в меню следует выбрать команду File → New Window (Crtl + N), откроется чистое (без приглашения >>>) новое окно. Желательно сразу сохранить файл с расширением *.py*, чтобы появилась подсветка синтаксиса. После того как код будет подготовлен, снова сохраните файл. Запуск скрипта выполняется командой Run → Run Module (F5). После этого в окне интерактивного режима появится результат выполнения кода.

Практическая работа

1. Запустите интерпретатор Питона в интерактивном режиме. Выполните несколько команд, например, арифметические примеры.
2. Подготовьте файл с кодом и передайте его на исполнение интерпретатору. Обратите внимание, что если просто записать арифметику, то никакого вывода не последует. Вы увидите пустоту. Это отличается от интерактивного режима. Чтобы увидеть решение, надо "обернуть" пример в функцию `print()`.

Урок 3. PyCharm Community. Основы работы

PyCharm – это одна из наиболее удобных сред разработки на языке Python. Существует в двух версиях:

- PyCharm Community – свободно-распространяемая версия с открытым исходным кодом.
- PyCharm Professional – проприетарная платная версия с триальным периодом.

В версии Community вы сможете программировать в основном на Python, в Professional – также на смежных языках (веб-программирование), использовать множество фреймворков.

В данном уроке мы рассмотрим создание проекта в PyCharm Community, первоначальную настройку среды и некоторые особенности работы в ней. Полную документацию смотрите на сайте разработчика данной IDE: https://www.jetbrains.com/help/pycharm/installation-guide.html

Скачать саму среду можно по адресу https://www.jetbrains.com/ru-ru/pycharm

PyCharm не содержит самого интерпретатора Python, поэтому последний уже должен быть установлен в системе. В дистрибутивах Linux обычно это так и есть: пакет интерпретатора Python устанавливается вместе с операционной системой. Пользователи Windows, если еще не сделали этого, могут скачать интерпретатор Питона с официального сайта: https://www.python.org/downloads/

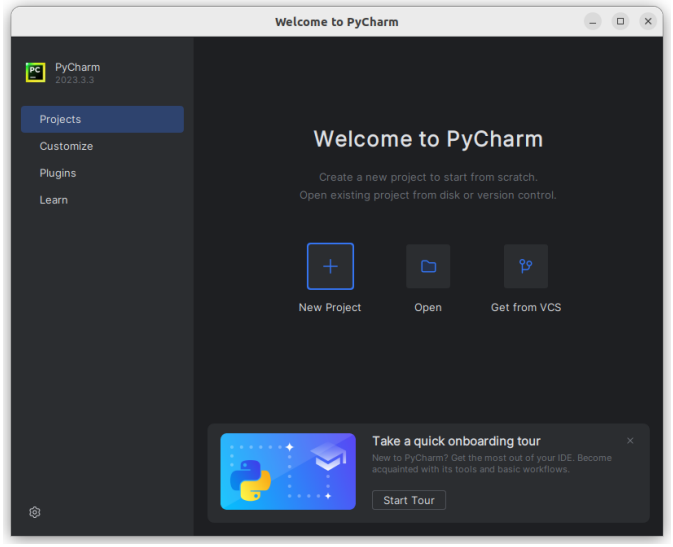
В Linux установку PyCharm Community лучше выполнить с помощью менеджера пакетов вашей операционной системы. Если такой возможности нет или это не ваш способ, то загрузив с официального сайта JetBrains и распаковав установочный пакет, найдите в нем файл *Install***.txt* или подобный, в котором описано, что надо сделать, чтобы установить и запустить среду разработки.

В этом случае процесс может выглядеть следующим образом:

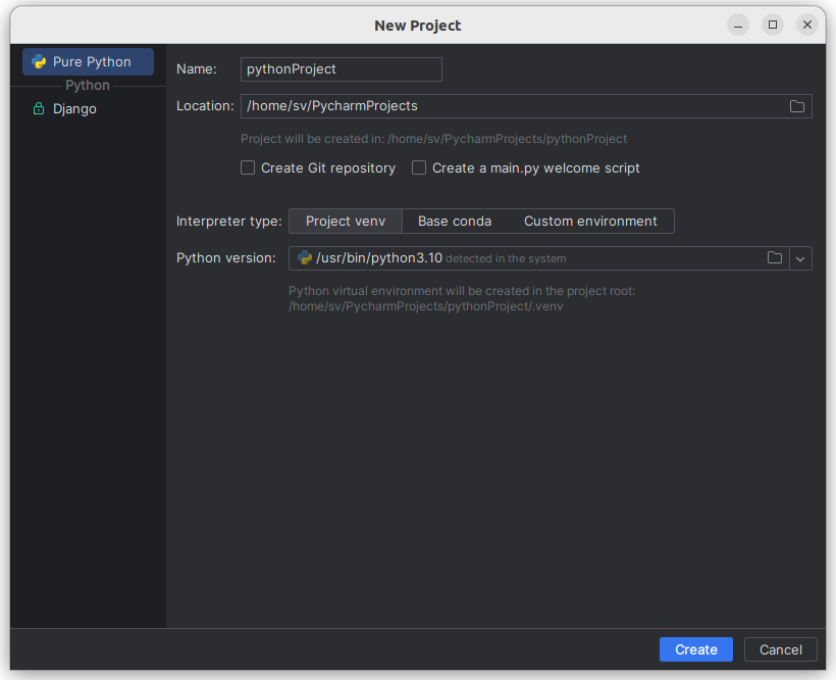
1. Перемещаем каталог с файлами среды разработки в директорию */opt* командой `sudo mv Загрузки/pycharm-community/ /opt/`
2. Переходим в директорию *bin* только что перемещенного каталога: `cd /opt/pycharm-community/bin/`
3. Выполняем файл *pycharm.sh* командой `./pycharm.sh`

При первом запуске PyCharm будет предложено принять пользовательское соглашение, также появится окно с вопросом отправлять или нет анонимные данные о том, как вы используете продукт.

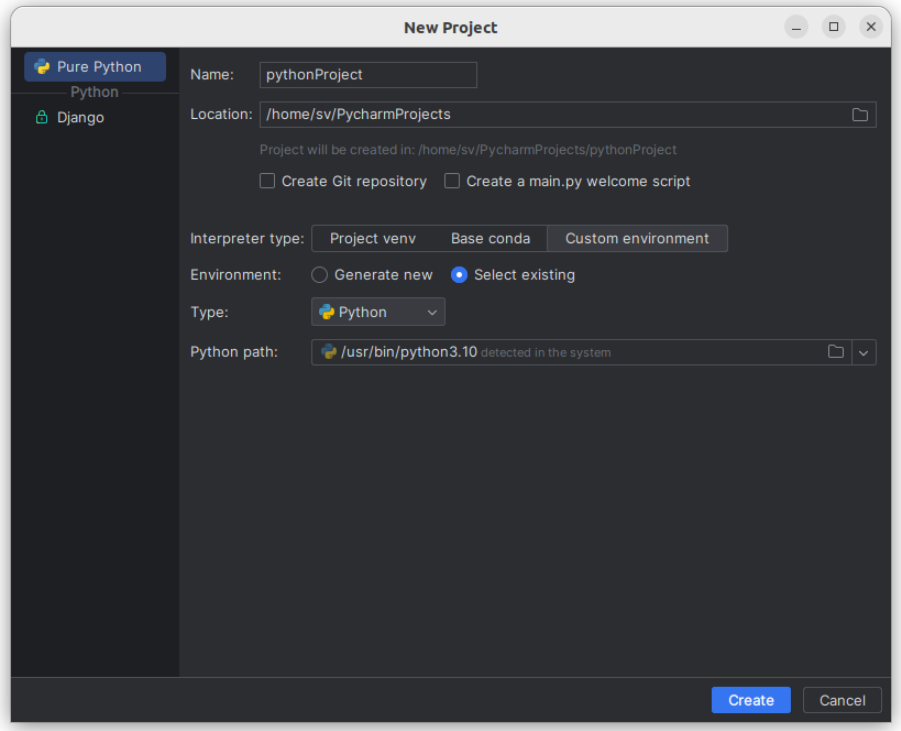
Далее появится приветственное окно, в котором среди прочего предлагается создать новый проект.



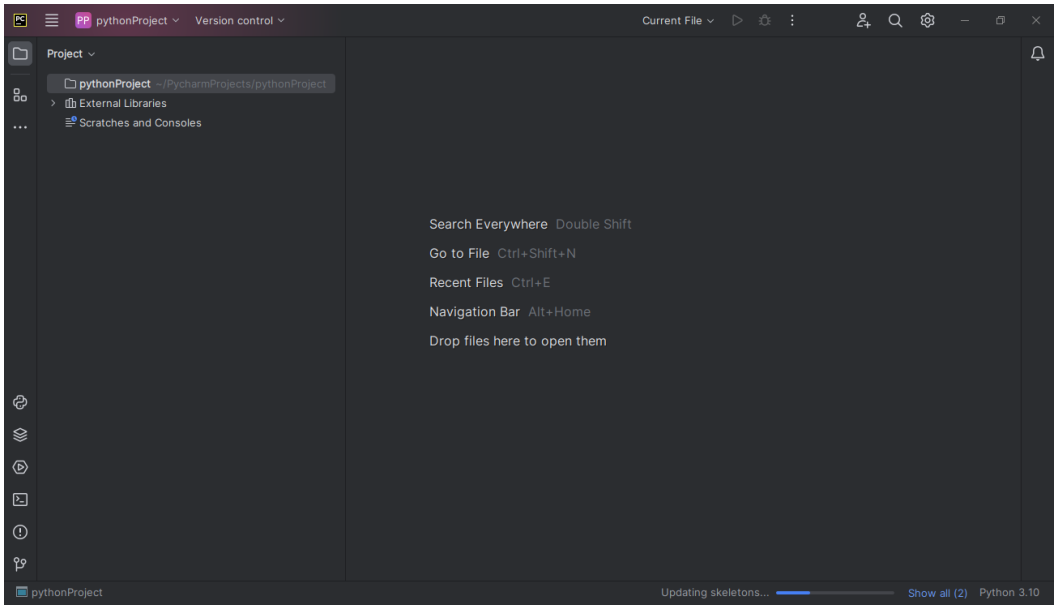
При создании проекта появляется диалоговое окно, в котором следует указать имя проекта и адрес его родительского каталога. Можно согласиться с заданными по умолчанию вариантами.



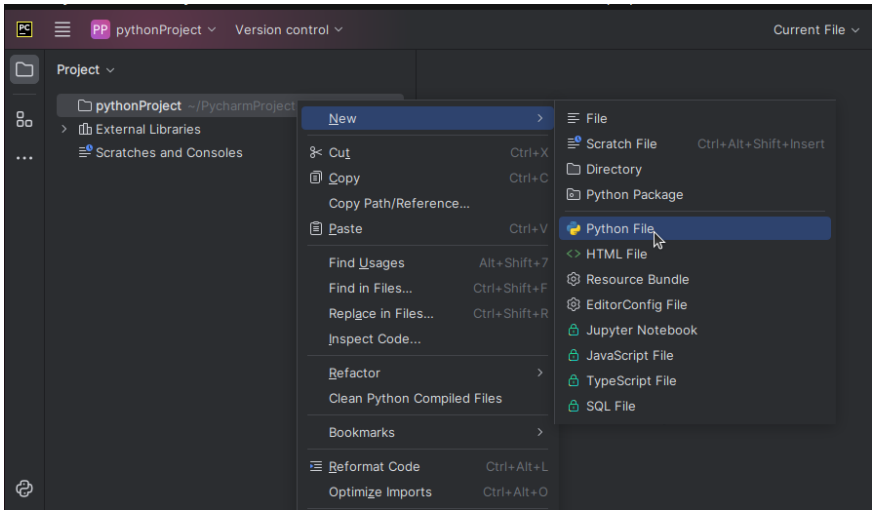
По-умолчанию предлагается использовать виртуальное окружение (выбран вариант Project venv). Однако если вы только учитесь языку Питона и никаких дополнительных библиотек устанавливать не планируете, то во избежание большого количества непонятных файлов в каталоге проекта, может быть целесообразно отказаться от создания виртуальной среды Python. Для этого надо переключиться на Custom environment, там в качестве окружения выбрать Select existing, тип – Python и указать путь до установленного в операционной системе интерпретатора.



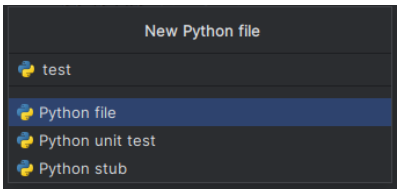
Далее запустится среда разработки, в ней будет открыт только что созданный проект.



Слева на панели Project управляют файлами проекта. На скрине выше в папке *pythonProject* нет ни одного файла (у вас там уже может быть каталог с виртуальным окружением). Чтобы создать файл, в котором будет написана программа на Python, кликнем по этой папке правой кнопкой мыши. В контекстном меню выбираем New → Python File.

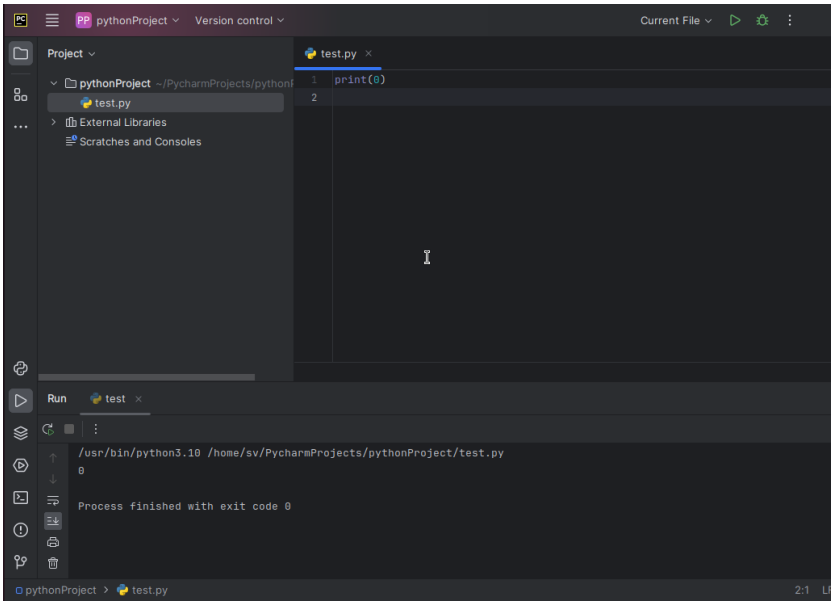


После этого в центральной части среды разработки появится небольшое окно, в которое вписываем имя файла.

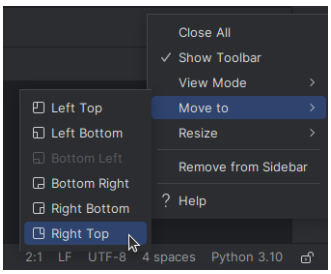


Нажав Enter, вы увидите файл на панели Project. Также он будет открыт в центральной части окна PyCharm.

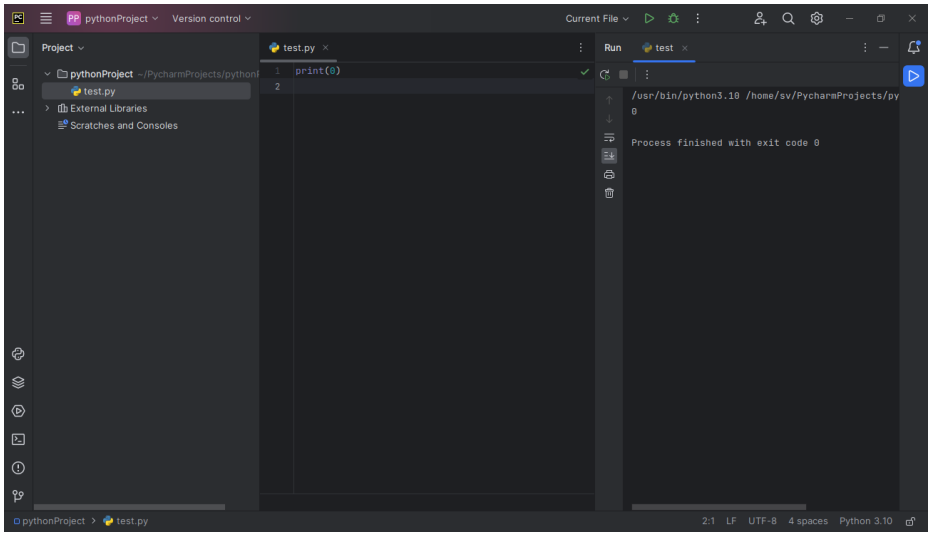
После того как исходный код написан, чтобы запустить программу, надо нажать Shift + F10. (возможно для первого запуска понадобится нажать Ctrl + Shift + F10). Внизу раскроется вкладка Run, в которой отобразиться результат выполнения.



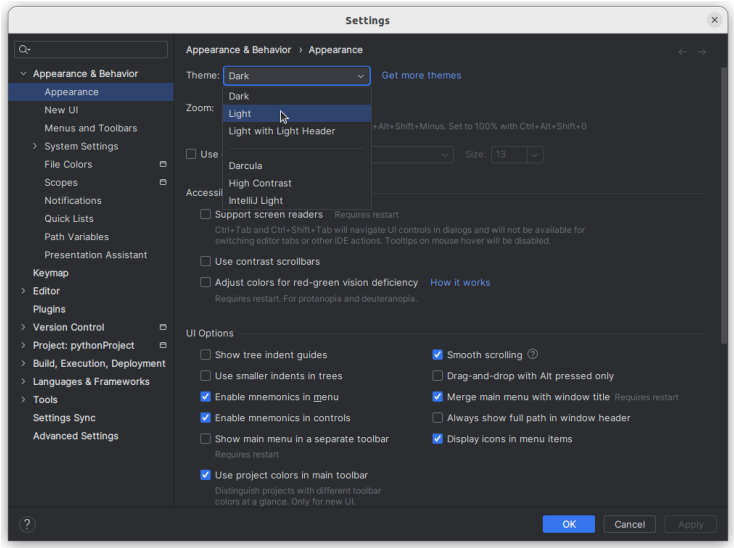
Иногда удобнее, чтобы панель выполнения программы открывалась не снизу, а, например, справа. В этом случае в настройках панели (справа значок с тремя вертикальными точками) следует выбрать Move to → Right Top.



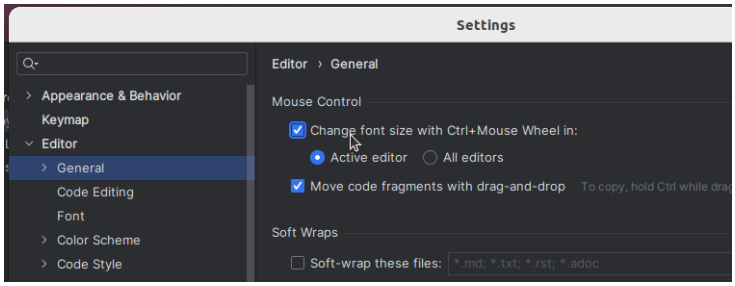
После этого интерфейс среды разработки примет такой вид:



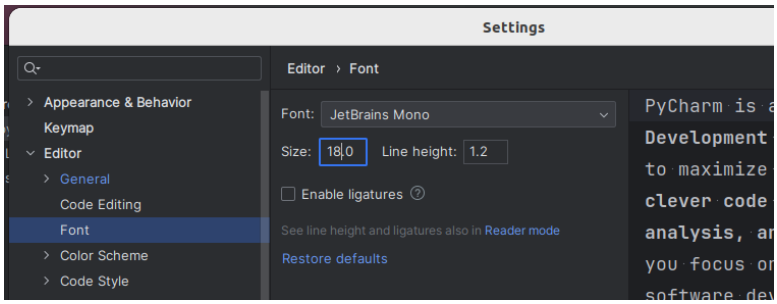
Внешний вид среды и множество других ее свойств, поведение настраиваются в окне Settings (меню File → Settings). На скрине ниже показано, как изменить темную тему оформления PyCharm на светлую.



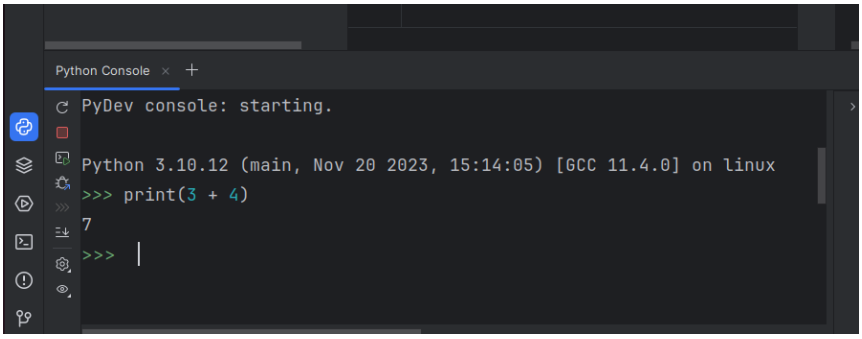
Бывает удобно менять размер шрифта в редакторе кода, зажав Ctrl и прокручивая колесо мыши. Чтобы воспользоваться этой возможностью в PyCharm, надо установить соответствующий флажок в разделе Editor → General окна настроек.



Изменить по-умолчанию заданный размер шрифта можно в разделе Editor → Font.



В PyCharm встроена интерактивная консоль, в которой выполняют небольшие фрагменты кода без создания файлов.



Если в дистрибутивах Linux вы устанавливали среду разработки самостоятельно, а не через менеджер пакетов, то значок PyCharm скорее всего не появится в системном меню. И для последующего запуска среды вам снова надо будет обращаться к файлу *pycharm.sh*. Чтобы создать ярлык на приложение, в меню Tools поищите пункт Create Desktop Entry… .

Теперь рассмотрим некоторые особенности работы в PyCharm, точнее в его редакторе кода. Многие из них универсальны, характерны для других сред разработки. Так нажатие Ctrl + D дублирует строку, в которой находится курсор.

Ctrl + C копирует строку, в которой находится курсор, выделять строку при этом не надо. Потом копию можно вставить в любое место программы командой Ctrl + V.

Если надо скопировать или продублировать участок в несколько строк, его следует выделить.

Выделенный участок можно сдвинуть вправо (сделать вложенным), нажав Tab. Смещение влево (на внешний уровень) выполняется комбинацией Shift + Tab.

Поднять/опустить (поменять местами с предшествующей/нижестоящей) строку или выделенный участок можно с помощью сочетаний Shift + Ctrl + стрелка вверх или стрелка вниз клавиатуры.

Урок 4. Типы данных. Переменные

Данные и их типы

В реальной жизни мы совершаем различные действия над окружающими нас предметами, или объектами. Мы меняем их свойства, наделяем новыми функциями. По аналогии с этим компьютерные программы также управляют объектами, только виртуальными, цифровыми. Пока не дойдем до уровня объектно-ориентированного программирования, будем называть такие объекты **данными**.

Очевидно, данные бывают разными. Часто компьютерной программе приходится работать с числами и строками. Так мы уже имели дело с числами, выполняя над ними арифметические действия. Операция сложения выполняла изменение первого числа на величину второго, а умножение увеличивало одно число в количество раз, соответствующее второму.

Числа в свою очередь также бывают разными: целыми, вещественными, могут иметь огромную величину или очень длинную дробную часть.

При знакомстве с языком программирования Python мы столкнемся с тремя типами данных:

- **целые числа** (тип `int`) – положительные и отрицательные целые числа, а также 0 (например, 4, 687, -45, 0).
- **числа с плавающей точкой** (тип `float`) – дробные, они же вещественные, числа (например, 1.45, -3.789654, 0.00453). Обратите внимание, для отделения дробной части от целой в программировании обычно используется точка, а не привычная нам запятая.
- **строки** (тип `str`) — набор символов, заключенных в кавычки (например, "ball", "What is your name?", 'dkfjUUv', '6589'). Кавычки в Python могут быть одинарными или двойными; одиночный символ в кавычках также является строкой, отдельного символьного типа в Питоне нет.

Операции в программировании

Операция – это выполнение каких-либо действий над данными, которые в данном случае именуют **операндами**. Само действие выполняет **оператор** – специальный инструмент. Если бы вы выполняли операцию постройки стола, то вашими операндами были бы доска и гвоздь, а оператором – молоток.

Так в математике и программировании символ плюса является оператором операции сложения по отношению к числам. В случае строк этот же оператор выполняет операцию *конкатенации*, то есть соединения.

```
>>> 10.25 + 98.36
108.61
>>> 'Hello' + 'World'
'HelloWorld'
```

Здесь следует для себя отметить, что то, что делает оператор в операции, зависит не только от него, но и от типов данных, которыми он оперирует. Молоток в случае нападения на вас крокодила перестанет играть роль строительного инструмента. Однако в большинстве случаев операторы не универсальны. Например, знак плюса неприменим, если операндами являются, с одной стороны, число, а с другой – строка.

```
>>> 1 + 'a'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Здесь в строке `TypeError: unsupported operand type(s) for +: 'int' and 'str'` интерпретатор сообщает, что произошла ошибка типа – неподдерживаемый операнд для типов **`int`** и **`str`**.

Изменение типов данных

Приведенную выше операцию все-таки можно выполнить, если превратить число 1 в строку "1". Для изменения одних типов данных в другие в языке Python предусмотрен ряд встроенных в него функций (что такое функция в принципе, вы узнаете в других уроках). Поскольку мы пока работаем только с тремя типами (**`int`**, **`float`** и **`str`**), рассмотрим вызовы соответствующих им функций – `int()`, `float()`, `str()`.

```
>>> str(1) + 'a'
'1a'
>>> int('3') + 4
7
>>> float('3.2') + int('2')
5.2
>>> str(4) + str(1.2)
'41.2'
```

Эти функции преобразуют то, что помещается в их скобки соответственно в целое число, вещественное число или строку. Однако преобразовать можно не все:

```
>>> int('hi')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'hi'
```

Здесь возникла ошибка значения (`ValueError`), так как передан литерал (в данном случае строка с буквенными символами), который нельзя преобразовать к числу с основанием 10. Однако функция `int` не такая простая:

```
>>> int('101', 2)
5
>>> int('F', 16)
15
```

Если вы знаете о различных системах счисления, то поймете, что здесь произошло.

Обратим внимание еще на одно. Данные могут называться **значениями**, а также **литералами**. Эти три понятия ("данные", "значение", "литерал") не обозначают одно и то же, но близки и нередко употребляются как синонимы. Чтобы понять различие между ними, места их употребления, надо изучить программирование глубже.

Переменные

Данные хранятся в ячейках памяти компьютера. Когда мы вводим число, оно помещается в какую-то область памяти. Но как потом узнать, куда именно? Как впоследствии обращаться к этим данными? Нужно как-то установить связь с местом хранения наших данных, запомнить его и пометить.

Раньше, при написании программ на машинном языке, обращение к ячейкам памяти осуществляли с помощью указания их регистров, то есть конкретно сообщали, куда положить данные и откуда их взять. Однако с появлением ассемблеров при обращении к данным стали использовать словесные **переменные**, что куда удобней для человека.

Механизм связи между переменными и данными может различаться в зависимости от языка программирования и типов данных. Пока достаточно запомнить, что в программе данные связываются с каким-либо именем и в дальнейшем обращение к ним происходит через это имя-переменную.

Слово "переменная" обозначает, что сущность может меняться, в ней что-то непостоянно. Действительно, вы увидите это в дальнейшем, одна и та же переменная может быть связана сначала с одними данными, а потом – с другими. То есть ее значение может меняться, она переменчива.

В программе на языке Python, как и на большинстве других языков, связь между данными и переменными устанавливается с помощью знака `=`. Такая операция называется **присваивание** (также говорят "присвоение"). Например, выражение `sq = 4` означает, что на объект, представляющий собой число 4, находящееся в определенной области памяти, теперь ссылается переменная *sq*, и обращаться к этому объекту следует по имени *sq*.

Имена переменных могут быть любыми. Однако есть несколько общих правил их написания:

1. Желательно давать переменным осмысленные имена, говорящие о назначении данных, на которые они ссылаются.
2. Имя переменной не должно совпадать с командами языка (зарезервированными ключевыми словами).
3. Имя переменной должно начинаться с буквы или символа подчеркивания (_), но не с цифры.
4. Имя переменной не должно содержать пробелы.

Чтобы узнать значение, на которое ссылается переменная, находясь в режиме интерпретатора, достаточно ее вызвать, то есть написать имя и нажать Enter.

```
>>> sq = 4
>>> sq
4
```

Вот более сложный пример работы с переменными в интерактивном режиме:

```
>>> apples = 100
>>> eat_day = 2
>>> days = 7
>>> apples = apples - eat_day * days
>>> apples
86
```

Здесь используются три переменные: *apples*, *eat_day* и *days*. Каждой из них присваивается свое значение. Выражение `apples = apples - eat_day * days` сложное. Сначала выполняется подвыражение, стоящее справа от знака равенства. После этого его результат присваивается переменной *apples*, в результате чего ее старое значение (100) теряется. В подвыражении `apples - eat_day * days` вместо имен переменных на самом деле используются их значения, то есть числа 100, 2 и 7.

Практическая работа

1. Переменной *var_int* присвойте значение 10, *var_float* - значение 8.4, *var_str* - "No".
2. Значение, хранимое в переменной *var_int*, увеличьте в 3.5 раза. Полученный результат свяжите с переменной *var_big*.
3. Измените значение, хранимое в переменной *var_float*, уменьшив его на единицу, результат свяжите с той же переменной.
4. Разделите *var_int* на *var_float*, а затем *var_big* на *var_float*. Результат данных выражений не привязывайте ни к каким переменным.
5. Измените значение переменной *var_str* на "NoNoYesYesYes". При формировании нового значения используйте операции конкатенации (`+`) и повторения строки (`*`).
6. Выведите значения всех переменных.

Урок 5. Ввод и вывод данных

Мы уже встречались с функцией `print()`. Она отвечает за вывод данных, по-умолчанию на экран.

Ввод данных в программу и их вывод важны в программировании. Без ввода программы делали бы одно и то же, исключая случаи, когда в них самих генерируются случайные значения. Вывод позволяет увидеть, использовать, куда-нибудь передать результат работы программы.

Обычно требуется, чтобы программа обрабатывала какой-то диапазон различных входных данных, которые поступают в нее из внешних источников. В качестве последних могут выступать файлы, клавиатура, сеть, выходные данные из другой программы. В свою очередь

вывод данных, например, возможен в файл, по сети, в базу данных, на принтер. Однако нередко информацию просто выводят на экран монитора.

Можно сказать, что программа – это открытая система, которая обменивается чем-либо с внешней для нее средой. Если живой организм в основном обменивается веществом и энергией, то программа – данными, информацией.

Вывод данных. Функция print()

Что такое функция в программировании, узнаем позже. Пока будем считать, что `print()` – это такая команда языка Python, которая выводит то, что в ее скобках на экран.

```
>>> print(1032)
1032
>>> print(2.34)
2.34
>>> print("Hello")
Hello
```

В скобках могут быть любые типы данных. Кроме того, количество данных может быть различным:

```
>>> print("a:", 1)
a: 1
>>> one = 1
>>> two = 2
>>> three = 3
>>> print(one, two, three)
1 2 3
```

Можно передавать в функцию `print()` как непосредственно литералы (в данном случае "a:" и 1), так и переменные, вместо которых будут выведены их значения. Аргументы функции (то, что в скобках), разделяются между собой запятыми. В выводе вместо запятых значения разделены пробелом.

Если в скобках стоит выражение, то сначала оно выполняется, после чего `print()` уже выводит результат данного выражения:

```
>>> print("hello" + " " + "world")
hello world
>>> print(10 - 2.5/2)
8.75
```

В `print()` предусмотрены дополнительные параметры. Например, через параметр **`sep`** можно указать отличный от пробела разделитель строк:

```
>>> print("Mon", "Tue", "Wed", "Thu", "Fri", "Sat", "Sun", sep="-")
Mon-Tue-Wed-Thu-Fri-Sat-Sun
>>> print(1, 2, 3, sep="//")
1//2//3
```

Параметр **`end`** позволяет указывать, что делать, после вывода строки. По-умолчанию происходит переход на новую строку. Однако это действие можно отменить, указав любой другой символ или строку:

```
>>> print(10, end="")
10>>>
```

Обычно **`end`** используется не в интерактивном режиме, а в скриптах, когда несколько выводов подряд надо разделить не переходом на новую строку, а, скажем, запятыми. Сам переход на новую строку обозначается символом '\n'. Если присвоить это значение параметру **`end`**, то никаких изменений в работе функции **`print`** вы не увидите, так как это значение и так присвоено по-умолчанию:

```
>>> print(10, end='\n')
10
>>>
```

Однако, если надо отступить на одну дополнительную строку после вывода, то можно сделать так:

```
>>> print(10, end='\n\n')
10

>>>
```

В функцию **`print`** нередко передаются так называемые форматированные строки, хотя по смыслу их правильнее называть строки-шаблоны. Никакого отношения к самому **`print`** они не имеют. Когда такая строка находится в скобках `print()`, интерпретатор сначала согласно заданному в ней формату преобразует ее к обычной строке, после чего передает результат в `print()`.

Форматирование может выполняться в так называемом старом стиле или с помощью строкового метода **`format`**. Старый стиль также называют Си-стилем, так как он схож с тем, как происходит вывод на экран в языке C. Рассмотрим пример:

```
>>> pupil = "Ben"
>>> old = 16
>>> grade = 9.2
```

```
>>> print("It's %s, %d. Level: %f" % (pupil, old, grade))
It's Ben, 16. Level: 9.200000
```

Здесь вместо трех комбинаций символов `%s`, `%d`, `%f` подставляются значения переменных *pupil*, *old*, *grade*. Буквы **`s`**, **`d`**, **`f`** обозначают типы данных – строку, целое число, вещественное число. Если бы требовалось подставить три строки, то во всех случаях использовалось бы сочетание `%s`.

Хотя в качестве значения переменной *grade* было указано число 9.2, на экран оно вывелось с дополнительными нулями. Чтобы указать, сколько требуется знаков после запятой, надо перед **`f`** поставить точку, после нее указать желаемое количество знаков в дробной части:

```
>>> print("It's %s, %d. Level: %.1f" % (pupil, old, grade))
It's Ben, 16. Level: 9.2
```

Теперь посмотрим на метод `format()`:

```
>>> print("This is a {0}. It's {1}.".format("ball", "red"))
This is a ball. It's red.
>>>
>>> print("This is a {1}. It's {0}.".format("white", "cat"))
This is a cat. It's white.
>>>
>>> print("This is a {2}. It's {0} {1}.".format("a", "number", 1))
This is a 1. It's a number.
```

В строке в фигурных скобках указаны номера данных, которые будут сюда подставлены. Далее к строке применяется метод `format()`. В его скобках указываются сами данные (можно использовать переменные). На нулевое место подставится первый аргумент метода `format()`, на место с номером 1 – второй и т. д. На самом деле возможности метода **`format`** существенно шире, и для их изучения понадобился бы отдельный урок. Нам пока будет достаточно этого.

В новых релизах Питона появился третий способ создания форматированных строк – f-строки. Перед их открывающей кавычкой прописывается буква **`f`**. В самой строке внутри фигурных скобок записываются выражения на Python, которые исполняются, когда интерпретатор преобразует строку-шаблон в обычную.

```
>>> a = 10
>>> b = 1.33
>>> c = 'Box'
>>> print(f'qty - {a:5}, goods - {c}')
qty -    10, goods - Box
>>> print(f'price - {b + 0.2:.1f}')
price - 1.5
```

В примере число 5 после переменной *a* обозначает количество знакомест, отводимых под вывод значения переменной. В выражении `b + 0.2:.1f` сначала выполняется сложение, после этого значение округляется до одного знака после запятой.

Ввод данных. Функция input()

За ввод в программу данных с клавиатуры в Python отвечает функция **`input`**. Когда вызывается эта функция, программа останавливает свое выполнение и ждет, когда пользователь введет текст. После этого, когда он нажмет Enter, функция `input()` заберет введенный текст и передаст его программе, которая уже будет обрабатывать его согласно своим алгоритмам.

Если в интерактивном режиме ввести команду `input()`, то ничего интересного вы не увидите. Компьютер будет ждать, когда вы что-нибудь введете и нажмете Enter или просто нажмете Enter. Если вы что-то ввели, это сразу же отобразится на экране:

```
>>> input()
Yes!
'Yes!'
```

Функция `input()` передает введенные данные в программу. Их можно присвоить переменной. В этом случае интерпретатор не выводит строку сразу же:

```
>>> answer = input()
No, it is not.
```

В данном случае строка сохраняется в переменной *answer*, и при желании мы можем вывести ее значение на экран:

```
>>> answer
'No, it is not.'
```

При использовании функции `print()` кавычки в выводе опускаются:

```
>>> print(answer)
No, it is not.
```

Куда интересней использовать функцию `input()` в скриптах – файлах с кодом. Рассмотрим такую программу:

```
name_user = input()
city_user = input()
print(f'Вас зовут {name_user}. Ваш город {city_user}')
```

При запуске программы, компьютер ждет, когда будет введена сначала одна строка, потом вторая. Они будут присвоены переменным *name_user* и *city_user*. После этого значения этих

переменных выводятся на экран с помощью форматированного вывода. Пример выполнения скрипта:

```
Арнольд
Питонск
Вас зовут Арнольд. Ваш город Питонск
```

Эта программа далека от совершенства. Откуда пользователю знать, что от него хотят? Чтобы не вводить человека в замешательство, для функции **`input`** предусмотрен специальный параметр-приглашение. Это приглашение выводится на экран при вызове `input()`. Усовершенствованная программа может выглядеть так (сразу под ней пример ее выполнения):

```
name_user = input('Ваше имя: ')
city_user = input('Ваш город: ')
print(f'Вас зовут {name_user}. Ваш город {city_user}')
```

```
Ваше имя: Серый
Ваш город: Белый
Вас зовут Серый. Ваш город Белый
```

Обратите внимание, что в программу поступает строка. Даже если ввести число, функция `input()` все равно вернет его строковое представление. Но что делать, если надо получить число? Ответ: использовать функции преобразования типов.

```
qty = input("Сколько апельсинов? ")
price = input("Цена одного? ")

qty = int(qty)
price = float(price)

summa = qty * price

print("Заплатите", summa, "руб.")
```

```
Сколько апельсинов? 5
Цена одного? 35.80
Заплатите 179.0 руб.
```

В данном случае с помощью функций `int()` и `float()` строковые значения переменных *qty* и *price* преобразуются соответственно в целое число и вещественное число. После этого новые численные значения присваиваются тем же переменным.

Программный код можно сократить, если преобразование типов выполнить в тех же строках кода, где вызывается функция `input()`:

```
qty = int(input("Сколько апельсинов? "))
price = float(input("Цена одного апельсина? "))

summa = qty * price

print("Заплатите", summa, "руб.")
```

Сначала выполняется функция `input()`. Она возвращает строку, которую функция `int()` или `float()` сразу преобразует в число. Только после этого происходит присваивание переменной, то есть она сразу получает численное значение.

Практическая работа

1. Напишите программу (файл *user.py*), которая запрашивала бы у пользователя:
   - его имя (например, "What is your name?")
   - возраст ("How old are you?")
   - место жительства ("Where are you live?")

   После этого выводила бы три строки:

   "This is *имя*"

   "It is *возраст*"

   "(S)he live in *место_жительства*"

   Вместо *имя*, *возраст*, *место_жительства* должны быть данные, введенные пользователем. Примечание: можно писать фразы на русском языке, но если вы планируете стать профессиональным программистом, привыкайте к английскому.
2. Напишите программу (файл *arithmetic.py*), которая предлагала бы пользователю решить пример 4 * 100 - 54. Потом выводила бы на экран правильный ответ и ответ пользователя. Подумайте, нужно ли здесь преобразовывать строку в число.
3. Запросите у пользователя четыре числа. Отдельно сложите первые два и отдельно вторые два. Разделите первую сумму на вторую. Выведите результат на экран так, чтобы ответ содержал две цифры после запятой.

Урок 6. Логические выражения и операторы

Логические выражения и логический тип данных

Часто в реальной жизни мы соглашаемся с каким-либо утверждением или отрицаем его. Например, если вам скажут, что сумма чисел 3 и 5 больше 7, вы согласитесь, скажете: "Да, это правда". Если же кто-то будет утверждать, что сумма трех и пяти меньше семи, то вы расцените такое утверждение как ложное.

Подобные фразы предполагают только два возможных ответа – либо "да", когда выражение оценивается как правда/истина, либо "нет", когда утверждение оценивается как ошибочное/ложное. В программировании и математике **если результатом вычисления выражения может быть лишь истина или ложь, то такое выражение называется логическим**.

Например, выражение 4 > 5 является логическим, так как его результатом является либо правда, либо ложь. Выражение 4 + 5 не является логическим, так как результатом его выполнения является число.

На позапрошлом уроке мы познакомились с тремя типами данных – целыми и вещественными числами, а также строками. Сегодня введем четвертый – **логический тип данных** (тип `bool`). Его также называют булевым. У этого типа всего два возможных значения: **True** (правда) и **False** (ложь).

```
>>> a = True
>>> type(a)
<class 'bool'>
>>> b = False
>>> type(b)
<class 'bool'>
```

Здесь переменной *a* было присвоено значение `True`, после чего с помощью встроенной в Python функции `type()` проверен ее тип. Интерпретатор сообщил, что это переменная класса `bool`. Понятия "класс" и "тип данных" в данном случае одно и то же. Переменная *b* также связана с булевым значением.

В программировании `False` обычно приравнивают к нулю, а `True` – к единице. Чтобы в этом убедиться, можно преобразовать булево значение к целочисленному типу:

```
>>> int(True)
1
>>> int(False)
0
```

Возможно и обратное. Можно преобразовать какое-либо значение к булевому типу:

```
>>> bool(3.4)
True
>>> bool(-150)
True
>>> bool(0)
False
>>> bool(' ')
True
```

```
>>> bool('')
False
```

И здесь работает правило: всё, что не 0 и не пустота, является правдой.

Логические операторы

В естественном языке (например, русском), чтобы сравнивать одно с другим, мы используем слова "равно", "больше", "меньше". В языках программирования для этого есть специальные знаки, подобные тем, которые используются в математике: `>` (больше), `<` (меньше), `>=` (больше или равно), `<=` (меньше или равно), `==` (равно), `!=` (не равно).

Не путайте операцию присваивания значения переменной, обозначаемую в языке Python одиночным знаком "равно", и операцию сравнения (два знака "равно"). Присваивание и сравнение – разные операции.

```
>>> a = 10
>>> b = 5
>>> a + b > 14
True
>>> a < 14 - b
False
>>> a <= b + 5
True
>>> a != b
True
>>> a == b
False
>>> c = a == b
>>> a, b, c
(10, 5, False)
```

В данном примере выражение `c = a == b` состоит из двух подвыражений. Сначала происходит сравнение (`==`) переменных *a* и *b*. После этого результат логической операции присваивается переменной *c*. Выражение `a, b, c` просто выводит значения переменных на экран.

Сложные логические выражения

Логические выражения типа `kbyte >= 1023` являются простыми, так как в них выполняется только одна логическая операция. Однако, на практике нередко возникает необходимость в более сложных выражениях. Может понадобиться получить ответа "Да" или "Нет" в зависимости от результата выполнения двух простых выражений. Например, "на улице идет снег или дождь", "переменная *news* больше 12 и меньше 20".

В таких случаях используются специальные операторы, объединяющие два и более простых логических выражения. Широко используются два оператора – так называемые логические И (**and**) и ИЛИ (**or**).

Чтобы получить `True` при использовании оператора `and`, необходимо, чтобы результаты обоих простых выражений, которые связывает данный оператор, были истинными. Если хотя бы в одном случае результатом будет `False`, то и все сложное выражение будет ложным.

Чтобы получить `True` при использовании оператора `or`, необходимо, чтобы результат хотя бы одного простого выражения, входящего в состав сложного, был истинным. В случае оператора `or` сложное выражение становится ложным лишь тогда, когда ложны оба составляющие его простые выражения.

Допустим, переменной *x* было присвоено значение 8 (`x = 8`), переменной *y* присвоили 13 (`y = 13`). Логическое выражение `y < 15 and x > 8` будет выполняться следующим образом. Сначала выполнится выражение `y < 15`. Его результатом будет `True`. Затем выполнится выражение `x > 8`. Его результатом будет `False`. Далее выражение сведется к `True and False`, что вернет `False`.

```
>>> x = 8
>>> y = 13
>>> y < 15 and x > 8
False
```

Если бы мы записали выражение так: `x > 8 and y < 15`, то оно также вернуло бы `False`. Однако сравнение `y < 15` не выполнялось бы интерпретатором, так как его незачем выполнять. Ведь первое простое логическое выражение (`x > 8`) уже вернуло ложь, которая, в случае оператора `and`, превращает все выражение в ложь.

В случае с оператором `or` второе простое выражение проверяется, если первое вернуло ложь, и не проверяется, если уже первое вернуло истину. Так как для истинности всего выражения достаточно единственного `True`, неважно по какую сторону от `or` оно стоит.

```
>>> y < 15 or x > 8
True
```

В языке Python есть еще унарный логический оператор `not`, то есть отрицание. Он превращает правду в ложь, а ложь в правду. Унарный он потому, что применяется к одному выражению, стоящему после него, а не справа и слева от него как в случае бинарных `and` и `or`.

```
>>> not y < 15
False
```

Здесь `y < 15` возвращает `True`. Отрицая это, мы получаем `False`.

```
>>> a = 5
>>> b = 0
>>> not a
False
>>> not b
True
```

Число 5 трактуется как истина, отрицание истины дает ложь. Ноль приравнивается к `False`. Отрицание `False` дает `True`.

Практическая работа

1. Присвойте двум переменным любые числовые значения.
2. Используя переменные из п. 1, с помощью оператора `and` составьте два сложных логических выражения, одно из которых дает истину, другое – ложь.
3. Аналогично выполните п. 2, но уже с оператором `or`.
4. Попробуйте использовать в логических выражениях переменные строкового типа. Объясните результат.
5. Напишите программу, которая запрашивала бы у пользователя два числа и выводила бы `True` или `False` в зависимости от того, больше первое число второго или нет.

Урок 7. Ветвление. Условный оператор

Ход выполнения программы может быть *линейным*, то есть таким, когда выражения выполняются друг за другом, начиная с первого и заканчивая последним. Ни одна строка кода программы не пропускается.

Однако чаще в программах бывает не так. При выполнении кода, в зависимости от тех или иных условий, некоторые его участки могут быть опущены, в то время как другие – выполнены. Иными словами, в программе может присутствовать *ветвление*, которое реализуется **условным оператором – особой конструкцией языка программирования**.

Проведем аналогию с реальностью. Человек живет по расписанию. Можно сказать, расписание – это алгоритм для человека, его программный код, подлежащий выполнению. В расписании на 18.00 стоит поход в бассейн. Однако экземпляр биоробота класса Homo sapiens через свои рецепторы-сенсоры получает информацию, что воду из бассейна слили. Разумно было бы отменить занятие по плаванию, то есть изменить ход выполнения программы-расписания. Одним из условий посещения бассейна должно быть его функционирование, иначе должны выполняться другие действия.

Подобная нелинейность действий может быть реализована в компьютерной программе. Например, часть кода будет выполняться лишь при определенном значении конкретной переменной. В языках программирования используется приблизительно такая конструкция условного оператора:

```
if логическое_выражение {
    выражение 1;
    выражение 2;
    …
}
```

Перевести на человеческий язык можно так: **если логическое выражение возвращает истину, то выполняются выражения внутри фигурных скобок**; если логическое выражение возвращает ложь, то код внутри фигурных скобок не выполняется. С английского "if" переводится как "если".

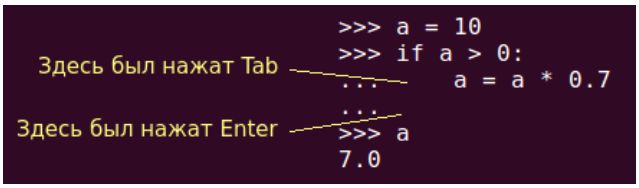
Конструкция `if логическое_выражение` называется **заголовком условного оператора**. Выражения внутри фигурных скобок – **телом условного оператора**. Тело может содержать как множество выражений, так и всего одно.

Пример использования условного оператора в языке программирования Python:

```
if n < 100:
    a = a + b
```

В Питоне вместо фигурных скобок используется двоеточие. Обособление вложенного кода, то есть тела оператора, достигается за счет отступов. В программировании принято делать отступ равным четырем пробелам. Можно использовать клавишу табуляции (Tab) на клавиатуре.

Большинство сред программирования автоматически создают отступ, как только вы поставите двоеточие и перейдете на новую строку. Однако при работе в интерактивном режиме отступы надо добавлять вручную.



Нахождение в теле условного оператора здесь обозначается тремя точками. При создании файла со скриптом таких точек быть не должно, как и приглашения `>>>`.

Python считается языком с ясным синтаксисом и легко читаемым кодом. Это достигается сведением к минимуму таких вспомогательных элементов как различные скобки и точка с запятой. Для разделения выражений используется переход на новую строку, а для обозначения вложенных выражений – отступы от начала строки. В других языках данный стиль

программирования также используется, но лишь для удобочитаемости кода человеком. В Питоне же такой стиль возведен в ранг синтаксического правила.

В примере выше логическим выражением является `n < 100`. Если оно возвращает истину, то выполнится строчка кода `a = a + b`. Если логическое выражение ложно, то выражение `a = a + b` не выполнится.

Данный пример вырван из контекста и сам по-себе не является рабочим. Полная версия программы могла бы выглядеть так:

```
a = 50
b = 10
n = 98

if n < 100:
    a = a + b

print(a)
```

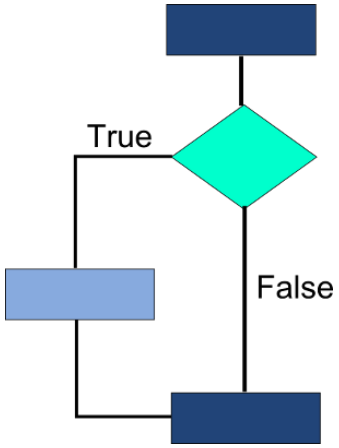
Последняя строчка кода `print(a)` уже не относится к условному оператору, что обозначено отсутствием перед ней отступа. Она не является вложенной в условный оператор, значит, не принадлежит ему.

Поскольку переменная *n* равна 98, а это меньше 100, то *a* станет равной 60. Это значение будет выведено на экран. Если переменная *n* изначально была бы связана, например, со значением 101, то на экран было бы выведено 50. Потому что при *n*, равной 101, логическое выражение в заголовке условного оператора вернуло бы ложь. Значит, тело не было бы выполнено, и переменная *a* не изменилась бы.

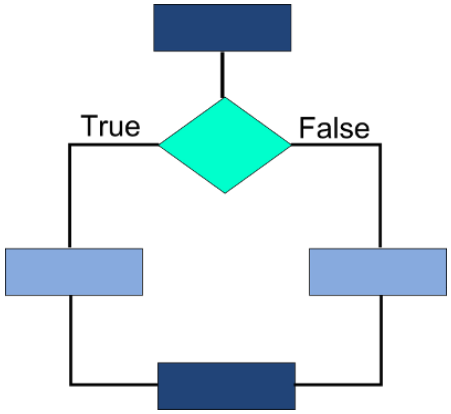
Структуру программы можно изобразить следующим образом:

Основная ветка программы выполняется всегда, а вложенный код лишь тогда, когда в темно-зеленой строчке, обозначающей заголовок условного оператора, случается истина.

Для небольших программ иногда чертят так называемые блок-схемы, отражающие алгоритм выполнения. В языке блок-схем различные части кода обозначаются своими фигурами. Так блоку последовательно выполняемых действий соответствует прямоугольник, ветвлению – ромб. Для кода выше блок-схема может выглядеть так:



Условный оператор может включать не одну ветку, а две, реализуя тем самым полноценное ветвление.



В случае возврата логическим выражением `False` поток выполнения программы не возвращается сразу в основную ветку. На случай `False` существует другой вложенный код, отличный от случая `True`. Другими словами, встретившись с расширенной версией условного оператора, поток выполнения программы не вернется в основную ветку, не выполнив хоть какой-нибудь вложенный код.

В языках программирования разделение на две ветви достигается с помощью добавления блока **else**, получается так называемое **if–else** (если-иначе). Синтаксис выглядит примерно так:

```
if логическое_выражение {
    выражение 1;
    выражение 2;
    …
}
else {
    выражение 3;
    …
}
```

Если условие при инструкции `if` оказывается ложным, то выполняется блок кода при инструкции `else`. **Ситуация, при которой бы выполнились обе ветви, невозможна.** Либо код, принадлежащий `if`, либо код, принадлежащий `else`. Никак иначе. **В заголовке `else` никогда не бывает логического выражения.**

Пример программы с веткой `else` на языке Python (под ним показаны варианты выполнения):

```
your_money = int(input('Сколько у вас монет? '))

sword = 50
helmet = 32

if sword + helmet > your_money:
    print('Вы не можете купить меч и шлем')
else:
    print('Удачный апгрейд!')
```

```
Сколько у вас монет? 85
Удачный апгрейд!
```

```
Сколько у вас монет? 63
Вы не можете купить меч и шлем
```

Следует иметь в виду, что логическое выражение при `if` может выглядеть нестандартно, то есть не так как `a > b` и тому подобное. Там может стоять просто одна переменная, число, слово `True` или `False`, а также сложное логическое выражение, когда два простых соединяются через логически `and` или `or`.

```
a = ?
if a:
   a = 1
```

Если вместо знака вопроса будет стоять 0, то с логической точки зрения это `False`, значит выражение в `if` не будет выполнено. Если *a* будет связано с любым другим числом, то оно будет расцениваться как `True`, и тело условного оператора выполнится. Другой пример:

```
a = 5 > 0
if a:
    print(a)
```

Здесь *a* уже связана с булевым значением. В данном случае это `True`. Отметим, что в выражении `a = 5 > 0` присваивание выполняется после оператора сравнения, так что подвыражение `5 > 0` выполнится первым, после чего его результат будет присвоен переменной *a*. На будущее, если

вы сомневаетесь в последовательности выполнения операторов, используйте скобки, например так: `a = (5 > 0)`.

Третий пример:

```
if a > 0 and a < b:
    print(b - a)
```

Тут, чтобы вложенный код выполнился, *a* должно быть больше нуля и одновременно меньше *b*. Также в Питоне, в отличие от других языков программирования, позволительна такая сокращенная запись сложного логического выражения:

```
if 0 < a < b:
    print(b - a)
```

Практическая работа

1. Напишите программу, которая просит пользователя что-нибудь ввести с клавиатуры. Если он вводит какие-нибудь данные, то на экране должно выводиться сообщение "OK". Если он не вводит данные, а просто нажимает Enter, то программа ничего не выводит на экран.
2. Напишите программу, которая запрашивает у пользователя число. Если оно больше нуля, то в ответ на экран выводится число 1. Если введенное число не является положительным, то на экран должно выводиться -1.

Урок 8. Исключения и их обработка в Python

Ошибки и исключения

В любой, особенно большой, программе могут возникать ошибки, приводящие к ее неработоспособности или к тому, что программа делает не то что должна. Причин возникновения ошибок много.

Программист может сделать ошибку в употреблении самого языка программирования. Другими словами, выразиться так, как выражаться не положено. Например, начать имя переменной с цифры или забыть поставить двоеточие в заголовке сложной инструкции. Подобные ошибки называют **синтаксическими**, они нарушают синтаксис и пунктуацию языка. Интерпретатор Питона, встретив ошибочное выражение, не знает как его интерпретировать. Поэтому останавливает выполнение программы и выводит соответствующее сообщение, указав на место возникновения ошибки:

```
>>> 1a = 10
  File "<stdin>", line 1
    1a = 10
     ^
SyntaxError: invalid syntax
```

В терминологии языка Python здесь возникло исключение, принадлежащее классу `SyntaxError`. Согласно документации Python синтаксические ошибки все-таки принято относить к ошибкам, а все остальные – к исключениям. В некоторых языках программирования не используется слово "исключение", а ошибки делят на синтаксические и семантические. Нарушение семантики обычно означает, что, хотя выражения написаны верно с точки зрения синтаксиса языка, программа не работает так, как от нее ожидалось. Для сравнения. Вы можете грамотным русским языком сказать несколько предложений, но по смыслу это будет белиберда, или вас поймут не так, как вы думали.

В Python не говорят о семантических ошибках, говорят об **исключениях**. Их множество. В этом уроке мы рассмотрим некоторые из них, в последующих встретимся еще с несколькими.

Если вы попытаетесь обратиться к переменной, которой не было присвоено значение, что в случае Python означает, что переменная вообще не была объявлена, она не существует, то возникнет исключение `NameError`.

```
>>> a = 0
>>> print(a + b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'b' is not defined
```

Последнюю строку сообщения можно перевести как "Ошибка имени: имя 'b' не определено".

Если исключение возникает при выполнении кода из файла, то вместо `line 1` будет указана строка, в которой оно возникло, например, `line 24`. Вместо `<stdin>` будет указано имя файла, например, *test.py*. В данном же случае `stdin` обозначает стандартный поток ввода. По-умолчанию это поток ввода с клавиатуры. Строка 1 – потому что в интерактивном режиме каждое выражение интерпретируется отдельно, как обособленная программка. Если написать выражение, состоящее из нескольких строк, то линия возникновения ошибки может быть другой:

```
>>> a = 0
>>> if a == 0:
...     print(a)
...     print(a + b)
...
0
Traceback (most recent call last):
```

```
  File "<stdin>", line 3, in <module>
NameError: name 'b' is not defined
```

Следующие два исключения, о которых следует упомянуть, и с которыми вы уже могли встретиться в предыдущих уроках, это **`ValueError`** и **`TypeError`** – ошибка значения и ошибка типа.

```
>>> int("Hi")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'Hi'
>>>
>>> 8 + "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

В примере строку "Hi" нельзя преобразовать к целому числу. Возникает исключение **`ValueError`**, потому что функция `int()` не может преобразовать такое значение.

Число 8 и строка "3" принадлежат разным типам, операнд сложения между которыми не поддерживается. При попытке их сложить возникает исключение **`TypeError`**.

Деление на ноль вызывает исключение **`ZeroDivisionError`**:

```
>>> 1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Обработка исключений. Оператор try-except

Когда ошибки фиксируются в процессе написания программы, то программист вынужден исправить код так, чтобы их не было. Однако исключительные ситуации могут возникать уже при использовании программы. Например, ожидается ввод числа, но человек вводит букву. Попытка преобразовать ее к числу приведет к возбуждению исключения **`ValueError`**, и программа аварийно завершится.

На этот случай в языках программирования, в том числе Python, существует специальный оператор, позволяющий перехватывать возникающие исключения и обрабатывать их так, чтобы программа продолжала работать или корректно завершала свою работу.

В Питоне такой перехват выполняет оператор **try-except**. "Try" переводится как "попытаться", "except" – как исключение. Словами описать его работу можно так: "Попытаться сделать вот это.

Если при этом возникло исключение, то сделать что-то другое." Его конструкция (но не логика работы) похожа на условный оператор с веткой `else`. Рассмотрим пример:

```
n = input("Введите целое число: ")
try:
    n = int(n)
    print("Удачно")
except:
    print("Что-то пошло не так")
```

Исключительная ситуация может возникнуть в третьей строчке кода, когда значение переменной *n* преобразуется к целому числу. Если это невозможно, то дальнейшее выполнение выражений в теле `try` прекращается. В данном случае выражение `print("Удачно")` выполнено не будет. При этом поток выполнения программы перейдет на ветку `except` и выполнит ее тело.

Если в теле `try` исключения не возникает, то тело ветки `except` не выполняется.

Вот пример вывода программы, когда пользователь вводит целое число:

```
Введите целое число: 100
Удачно
```

А здесь – когда вводит не то, что ожидалось:

```
Введите целое число: AA
Что-то пошло не так
```

Есть одна проблема. Код выше обработает любое исключение. Однако в теле `try` могут возникать разные исключения, и у каждого из них должен быть свой обработчик. Поэтому более правильным является указание типа исключения после ключевого слова `except`.

```
try:
    n = input("Введите целое число: ")
    n = int(n)
    print("Все нормально. Вы ввели число", n)
except ValueError:
    print("Вы ввели не целое число")
```

Теперь если сработает тело `except` мы точно знаем, из-за чего возникла ошибка. Но если в теле `try` возникнет еще какое-нибудь исключение, то оно не будет обработано. Для него надо написать отдельную ветку `except`. Рассмотрим программу:

```
try:
    a = float(input("Введите делимое: "))
    b = float(input("Введите делитель: "))
```

```
    c = a / b
    print("Частное: %.2f" % c)
except ValueError:
    print("Нельзя вводить строки")
except ZeroDivisionError:
    print("Нельзя делить на ноль")
```

При ее выполнении исключения могут возникнуть в трех строчках кода: где происходит преобразование введенных значений к вещественным числам и в месте, где происходит деление. В первом случае может возникнуть `ValueError`, во втором – `ZeroDivisionError`. Каждый тип исключения обрабатывается своей веткой `except`.

Несколько исключений можно сгруппировать в одну ветку и обработать совместно:

```
try:
    a = float(input("Введите делимое: "))
    b = float(input("Введите делитель: "))
    c = a / b
    print("Частное: %.2f" % c)
except (ValueError, ZeroDivisionError):
    print("Нельзя вводить строки или делить на ноль")
```

У оператора обработки исключений, кроме `except`, могут быть еще ветки `finally` и `else` (не обязательно обе сразу). Тело `finally` **выполняется всегда**, независимо от того, выполнялись ли блоки `except` в ответ на возникшие исключения или нет. Тело `else` **сработает, если исключений в** `try` **не было**, то есть не было переходов на блоки `except`.

```
try:
    n = input('Введите целое число: ')
    n = int(n)
except ValueError:
    print("Неверный ввод")
else:  # выполняется, когда в блоке try не возникло исключений
    print("Все нормально. Вы ввели число", n)
finally:  # выполняется в любом случае
    print("Конец программы")
```

Примечание. В данном коде используются комментарии. В языке Python перед ними ставится знак решетки `#`. Комментарии в программном коде пишутся исключительно для человека и игнорируются интерпретатором или компилятором.

Посмотрите, как выполняется программа в случае возникновения исключения и без этого:

```
Введите целое число: 4.3
Неверный ввод
Конец программы
```

```
Введите целое число: 4
Все нормально. Вы ввели число 4
Конец программы
```

В данном уроке изложены не все особенности обработки исключений. Так в более крупных программах, содержащих несколько уровней вложенности кода, функции, модули и классы, исключения могут обрабатываться не по месту их возникновения, а передаваться дальше по иерархии вызовов.

Также исключение может возникнуть в блоке **`except`**, **`else`** или **`finally`**, и тогда им нужен собственный обработчик. Модифицируем немного предыдущую программу и специально сгенерируем исключение в теле **`except`**:

```
try:
    n = input('Введите целое число: ')
    n = int(n)
except ValueError:
    print("Неверный ввод")
    3 / 0
except ZeroDivisionError:
    print("Деление на ноль")
else:
    print("Все нормально. Вы ввели число", n)
finally:
    print("Конец программы")
```

Поначалу может показаться, что все нормально. Исключение, выбрасываемое выражением `3 / 0` будет обработано веткой `except ZeroDivisionError`. Однако это не так. Эта ветка обрабатывает только исключения, возникающие в блоке **`try`**, к которому она сама относится. Вот вывод программы, если ввести не целое число:

```
Введите целое число: a
Неверный ввод
Конец программы
Traceback (most recent call last):
  File "test.py", line 15, in <module>
    n = int(n)
ValueError: invalid literal for int() with base 10: 'a'

During handling of the above exception, another exception occurred:
```

```
Traceback (most recent call last):
  File "test.py", line 6, in <module>
    3 / 0
ZeroDivisionError: division by zero
```

Мало того, что не было обработано деление на ноль, поскольку тело `except ValueError` неудачно завершилось, само исключение **`ValueError`** посчиталось необработанным. Решение проблемы может быть, например, таким:

```
...
except ValueError:
    print("Неверный ввод")
    try:
        3 / 0
    except ZeroDivisionError:
        print("Деление на ноль")
...
```

Здесь в тело **`except`** вложен свой внутренний обработчик исключений.

В конце отметим, что в программах, имеющих практическое значение, обязанность предвидеть возможность возникновения исключительных ситуаций и создавать код их обработки лежит на программисте. Тестировщик также должен понимать, в каких случаях могут возникать исключения. Его задача – написать код, который проверяет работу программы в разных ситуациях, в том числе крайних.

Практическая работа

Напишите программу, которая запрашивает ввод двух значений. Если хотя бы одно из них не является числом, то должна выполняться конкатенация, то есть соединение, строк. В остальных случаях введенные числа суммируются.

Примеры выполнения программы:

```
Первое значение: 4
Второе значение: 5
Результат: 9.0
```

```
Первое значение: a
Второе значение: 9
Результат: a9
```

Урок 9. Множественное ветвление: if-elif-else. Оператор match в Python

Ранее мы рассмотрели работу условного оператора `if`. С помощью его расширенной версии **if-else** можно реализовать две отдельные ветви выполнения. Однако алгоритм программы может предполагать выбор больше, чем из двух путей, например, из трех, четырех или даже пяти. В данном случае следует говорить о необходимости множественного ветвления.

Рассмотрим конкретный пример. Допустим, в зависимости от возраста пользователя, ему рекомендуется определенный видеоконтент. При этом выделяют группы от 3 до 6 лет, от 6 до 12, от 12 до 16, 16+. Итого 4 диапазона. Как бы мы стали реализовывать задачу, имея в наборе инструментов только конструкцию **if-else**?

Самый простой ответ – последовательно проверять вхождение введенного числа-возраста в определенный диапазон с помощью следующих друг за другом условных операторов:

```
old = int(input('Ваш возраст: '))

print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')

if 6 <= old < 12:
    print('"Марсианин"')

if 12 <= old < 16:
    print('"Загадочный остров"')

if 16 <= old:
    print('"Поток сознания"')
```

Примечание. Названия фильмов выводятся на экран в двойных кавычках. Поэтому в программе для определения строк используются одинарные.

Предложенный код прекрасно работает, но есть одно существенное "но". Он не эффективен, так как каждый `if` в нем – это отдельно взятый оператор, никак не связанный с другими `if`. Процессор тратит время и "нервы" на обработку каждого из них, даже если в этом уже нет необходимости. Например, введено число 10. В первом `if` логическое выражение возвращает ложь, и поток выполнения переходит ко второму `if`. Логическое выражение в его заголовке возвращает истину, и его тело выполняется. Всё, на этом программа должна была остановиться.

Однако следующий `if` никак не связан с предыдущим, поэтому далее будет проверяться вхождение значения переменной *old* в диапазон от 12 до 16, в чем необходимости нет. И далее

будет обрабатываться логическое выражение в последнем `if`, хотя уже понятно, что и там будет `False`.

Решить проблему избыточности проверок можно, вкладывая условные операторы друг в друга:

```
old = int(input('Ваш возраст: '))

print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')
else:
    if 6 <= old < 12:
        print('"Марсианин"')
    else:
        if 12 <= old < 16:
            print('"Загадочный остров"')
        else:
            if 16 <= old:
                print('"Поток сознания"')
```

Рассмотрим поток выполнения этого варианта кода. Сначала проверяется условие в первом `if`. Если здесь было получено `True`, то тело этого `if` выполняется, а в ветку `else` мы даже не заходим, так как она срабатывает только тогда, когда в условии `if` возникает ложь.

Если внешний `if` вернул `False`, поток выполнения программы заходит в соответствующий ему внешний `else`. В его теле находится другой `if` со своим `else`. Если введенное число попадает в диапазон от 6 до 12, выполнится тело вложенного `if`, после чего программа завершается. Если же число не попадает в диапазон от 6 до 12, то произойдет переход к ветке `else`. В ее теле находится свой условный оператор, имеющий уже третий уровень вложенности.

Таким образом до последней проверки (`16 <= old`) интерпретатор доходит только тогда, когда все предыдущие возвращают `False`. Если же по ходу выполнения программы возникает `True`, то все последующие проверки опускаются, что экономит ресурсы процессора. Кроме того, такая логика выполнения программы более правильная.

Можно ли как-то оптимизировать код множественного ветвления и не строить лестницу из вложенных друг в друга условных операторов? Во многих языках программирования, где отступы используются только для удобства чтения программистом, но не имеют никакого синтаксического значения, часто используется подобный стиль:

```
if логическое_выражение {
    … ;
}
else if логическое_выражение {
```

```
    … ;
}
else if логическое_выражение {
    … ;
}
else {
    … ;
}
```

Может показаться, что имеется только один уровень вложенности, и появляется новое расширение для `if`, выглядящее как `else if`. Но это только кажется. На самом деле `if`, стоящее сразу после `else`, является вложенным в это `else`. Выше приведенная схема – то же самое, что

```
if логическое_выражение {
    … ;
}
else
    if логическое_выражение {
        … ;
    }
    else
        if логическое_выражение {
            … ;
        }
        else {
            … ;
        }
```

Именно так ее "понимает" интерпретатор или компилятор. Однако считается, что человеку проще воспринимать первый вариант.

В Питоне такое поднятие вложенного `if` к внешнему `else` невозможно, потому что здесь отступы и переходы на новую строку имеют синтаксическое значение. Поэтому в язык Python встроена возможность настоящего **множественного ветвления на одном уровне вложенности, которое реализуется с помощью веток elif**.

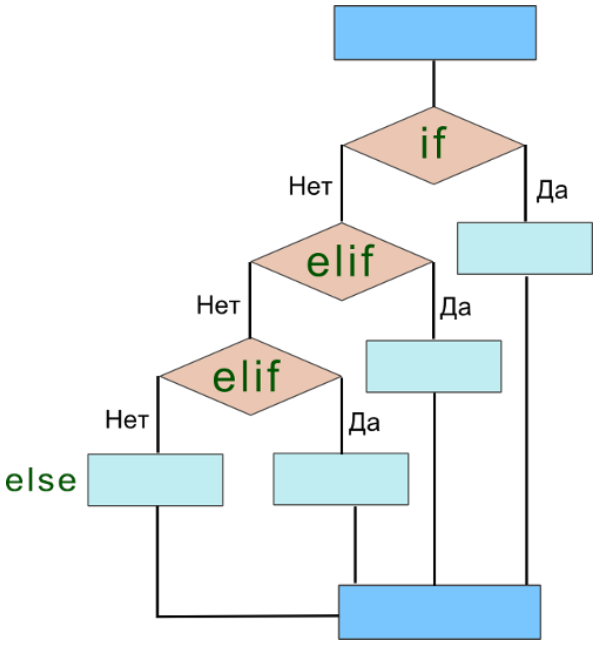
Слово "elif" образовано от двух первых букв слова "else", к которым присоединено слово "if". Это можно перевести как "иначе если".

В отличие от `else`, **в заголовке elif обязательно должно быть логическое выражение** также, как в заголовке `if`. Перепишем нашу программу, используя конструкцию множественного ветвления:

```
old = int(input('Ваш возраст: '))
```

```
print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')
elif 6 <= old < 12:
    print('"Марсианин"')
elif 12 <= old < 16:
    print('"Загадочный остров"')
elif 16 <= old:
    print('"Поток сознания"')
```

Обратите внимание, в конце, после всех `elif`, может использоваться одна ветка `else` для обработки случаев, не попавших в условия ветки `if` и всех `elif`. Блок-схему полной конструкции **if-elif-…-elif-else** можно изобразить так:



Как только тело `if` или какого-нибудь `elif` выполняется, программа сразу же возвращается в основную ветку (нижний ярко-голубой прямоугольник), а все нижеследующие `elif`, а также `else` пропускаются.

Оператор match-case в Python

Начиная с версии 3.10 в Питоне появился оператор `match`, который можно использовать как аналог оператора `switch`, который есть в других языках. На самом деле возможности `match` немного шире.

В `match` множественное ветвление организуется с помощью веток `case`:

```
match имя_переменной:
    case значение_1:
         действия
    case значение_2:
        действия
    …
```

Слова match-case можно перевести как "соответствовать случаю". То есть, если значение переменной или выражения при **`match`** соответствует значению при каком-либо **`case`**, то выполнятся действия, вложенные в этот **`case`**.

В отличие от if-elif здесь нельзя использовать логические выражения. После **`case`** должен находится литерал, конкретное значение, выражение, возвращающее однозначный результат.

Рассмотрим программу, в которой реализовать множественное ветвление с помощью match-case удобнее, чем через if-elif-else:

```
sign = input('Знак операции: ')
a = int(input('Число 1: '))
b = int(input('Число 2: '))

match sign:
    case '+':
        print(a + b)
    case '-':
        print(a - b)
    case '/':
        if b != 0:
            print(round(a / b, 2))
    case '*':
        print(a * b)
    case _:
        print('Неверный знак операции')
```

Здесь значение переменной *sign* проверяется не на вхождение в какой-либо диапазон, а на точное соответствие заданным строковым литералам. При этом в ветках **`case`** уже не надо писать `sign == '+'` или `sign == '-'`, как это пришлось бы делать в программе с if-elif:

```
if sign == '+':
    print(a + b)
elif sign == '-':
    print(a - b)
elif sign == '/':
    if b != 0:
        print(round(a / b, 2))
```

```
elif sign == '*':
    print(a * b)
else:
    print('Неверный знак операции')
```

Код с `match` выглядит более ясным.

Оператор `match` языка Python не имеет ветки `else`. Вместо нее используется ветка `case _`.

При одном `case` через оператор `|` можно перечислять несколько значений. Если значение переменной соответствует хотя бы одному из них, тело этого `case` выполнится.

```
sign = input('Знак операции: ')

match sign:
    case '+' | '-' | '*':
        a = int(input('Число 1: '))
        b = int(input('Число 2: '))
        print(eval(f'{a} {sign} {b}'))
    case _:
        print('Неверный знак операции')
```

В коде выше с помощью функции `eval()` переданная ей строка выполняется как выражение. Например, если были введены числа 3, 5 и знак *, то получится строка "3 * 5". Вызов `eval("3 * 5")` возвращает число 15.

Практическая работа

1. Спишите вариант кода программы "про возраст" с `if` и тремя ветками `elif` из урока. Дополните его веткой `else`, обрабатывающие случаи, когда пользователь вводит числа не входящие в заданные четыре диапазона. Подумайте, почему в первой версии программы (когда использовались не связанные друг с другом условные операторы) нельзя было использовать `else`, а для обработки не входящих в диапазоны случаев пришлось бы писать еще один `if`?
2. Усовершенствуйте предыдущую программу, обработав исключение `ValueError`, возникающее, когда вводится не целое число.
3. Напишите программу, которая запрашивает на ввод число. Если оно положительное, то на экран выводится цифра 1. Если число отрицательное, выводится -1. Если введенное число – это 0, то на экран выводится 0. Используйте в коде условный оператор множественного ветвления.

Урок 10. Циклы в программировании. Цикл while

Циклы являются такой же важной частью структурного программирования, как условные операторы. С помощью циклов можно организовать повторение выполнения участков кода. Потребность в этом возникает довольно часто. Например, пользователь последовательно вводит числа, и каждое из них требуется добавлять к общей сумме. Или нужно вывести на экран квадраты ряда натуральных чисел и тому подобные задачи.

"While" переводится с английского как "пока". Но не в смысле "до свидания", а в смысле "пока имеем это, делаем то".

Можно сказать, **while** является универсальным циклом. Он присутствует во всех языках, поддерживающих структурное программирование, в том числе в Python. Его синтаксис обобщенно для всех языков можно выразить так:

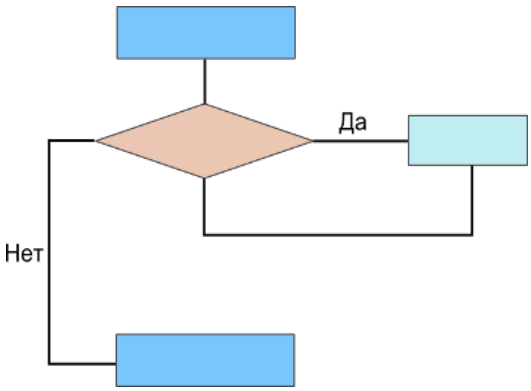
```
while логическое_выражение {
    выражение 1;
    …
    выражение n;
}
```

Это похоже на условный оператор `if`. Однако в случае циклических операторов их тела могут выполняться далеко не один раз. В случае `if`, если логическое выражение в заголовке возвращает истину, то тело выполняется единожды. После этого поток выполнения программы возвращается в основную ветку и выполняет следующие выражения, расположенные ниже всей конструкции условного оператора.

В случае `while`, после того как его тело выполнено, поток возвращается к заголовку цикла и снова проверяет условие. Если логическое выражение возвращает истину, то тело снова выполняется. Потом снова возвращаемся к заголовку и так далее.

Цикл завершает свою работу только тогда, когда логическое выражение в заголовке возвращает ложь, то есть условие выполнения цикла больше не соблюдается. После этого поток выполнения перемещается к выражениям, расположенным ниже всего цикла. Говорят, "происходит выход из цикла".

Рассмотрите блок-схему цикла `while`.



На ней ярко-голубыми прямоугольниками обозначена основная ветка программы, ромбом – заголовок цикла с логическим выражением, бирюзовым прямоугольником – тело цикла.

С циклом **while** возможны две исключительные ситуации:

- Если при первом заходе в цикл логическое выражение возвращает `False`, то тело цикла не выполняется ни разу. Эту ситуацию можно считать нормальной, так как при определенных условиях логика программы может предполагать отсутствие необходимости в выполнении выражений тела цикла.
- Если логическое выражение в заголовке `while` никогда не возвращает `False`, а всегда остается равным `True`, то цикл никогда не завершится, если только в его теле нет оператора принудительного выхода из цикла (`break`) или вызовов функций выхода из программы – `quit()`, `exit()` в случае Python. Если цикл повторяется и повторяется бесконечное количество раз, то в программе происходит **зацикливание**. В это время она зависает и самостоятельно завершиться не может.

Вспомним наш пример из урока про исключения. Пользователь должен ввести целое число. Поскольку функция `input()` возвращает строку, то программный код должен преобразовать введенное к целочисленному типу с помощью функции `int()`. Однако, если были введены символы, не являющиеся цифрами, то возникает исключение `ValueError`, которое обрабатывается веткой `except`. На этом программа завершается. Другими словами, если бы программа предполагала дальнейшие действия с числом (например, проверку на четность), а она его не получила, то единственное, что программа могла сделать, это закончить свою работу досрочно.

Но ведь можно просить и просить пользователя корректно вести число, пока он его не введет. Вот как может выглядеть реализующий это код:

```
n = input("Введите целое число: ")

while type(n) != int:
    try:
        n = int(n)
    except ValueError:
```

```
        print("Неправильно ввели!")
        n = input("Введите целое число: ")

if n % 2 == 0:
    print("Четное")
else:
    print("Нечетное")
```

Примечание 1. Не забываем, в языке программирования Python в конце заголовков сложных инструкций ставится двоеточие.

Примечание 2. В выражении `type(n) != int` с помощью функции `type()` проверяется тип переменной *n*. Если он не равен **`int`**, то есть значение *n* не является целым числом, а является в данном случае строкой, то выражение возвращает истину. Если же тип *n* равен **`int`**, то данное логическое выражение возвращает ложь.

Примечание 3. Оператор **`%`** в языке Python используется для нахождения остатка от деления. Так, если число четное, то оно без остатка делится на 2, то есть остаток будет равен нулю. Если число нечетное, то остаток будет равен единице.

Проследим алгоритм выполнения этого кода. Пользователь вводит данные, они имеют строковый тип и присваиваются переменной *n*. В заголовке **`while`** проверяется тип *n*. При первом входе в цикл тип *n* всегда строковый, то есть он не равен **`int`**. Следовательно, логическое выражение возвращает истину, что позволяет зайти в тело цикла.

Здесь в ветке **`try`** совершается попытка преобразования строки к целочисленному типу. Если она была удачной, то ветка **`except`** пропускается, и поток выполнения снова возвращается к заголовку **`while`**.

Теперь *n* связана с целым числом, следовательно, ее тип **`int`**, который не может быть не равен **`int`**. Он ему равен. Таким образом логическое выражение `type(n) != int` возвращает **`False`**, и весь цикл завершает свою работу. Далее поток выполнения переходит к оператору **if-else**, находящемуся в основной ветке программы. Здесь могло бы находиться что угодно, не обязательно условный оператор.

Вернемся назад. Если в теле **`try`** попытка преобразования к числу была неудачной, и было выброшено исключение **`ValueError`**, то поток выполнения программы отправляется в ветку **`except`** и выполняет находящиеся здесь выражения, последнее из которых просит пользователя снова ввести данные. Переменная *n* теперь имеет новое значение.

После завершения **`except`** снова проверяется логическое выражение в заголовке цикла. Оно даст **`True`**, так как значение *n* по-прежнему строка.

Выход из цикла возможен только тогда, когда значение *n* будет успешно конвертировано в число.

Рассмотрим следующий пример:

```
total = 100

i = 0
while i < 5:
    n = int(input())
    total = total - n
    i = i + 1

print("Осталось", total)
```

Сколько раз прокрутится цикл в этой программе, то есть сколько итераций он сделает? Ответ: 5.

1. Сначала переменная *i* равна 0. В заголовке цикла проверяется условие `i < 5`, и оно истинно. Тело цикла выполняется. В нем меняется значение *i*, путем добавления к нему единицы.
2. Теперь переменная *i* равна 1. Это меньше пяти, и тело цикла выполняется второй раз. В нем *i* меняется, ее новое значение 2.
3. Два меньше пяти. Тело цикла выполняется третий раз. Значение *i* становится равным трем.
4. Три меньше пяти. На этой итерации *i* присваивается 4.
5. Четыре по-прежнему меньше пяти. К *i* добавляется единица, и теперь ее значение равно пяти.

Далее начинается шестая итерация цикла. Происходит проверка условия `i < 5`. Но поскольку теперь оно возвращает ложь, то выполнение цикла прерывается, и его тело не выполняется.

"Смысловая нагрузка" данного цикла – это последовательное вычитание из переменной *total* вводимых чисел. Переменная *i* в данном случае играет только роль счетчика итераций цикла. В других языках программирования для таких случаев предусмотрен цикл **`for`**, который так и называется: "цикл со счетчиком". Его преимущество заключается в том, что в теле цикла не надо изменять переменную-счетчик, ее значение меняется автоматически в заголовке **`for`**.

В языке Python тоже есть цикл **`for`**. Но это не цикл со счетчиком. В Питоне он предназначен для перебора элементов последовательностей и других сложных объектов. Данный цикл и последовательности будут изучены в последующих уроках.

Для **`while`** наличие счетчика не обязательно. Представим, что надо вводить числа, пока переменная *total* больше нуля. Тогда код будет выглядеть так:

```
total = 100

while total > 0:
    n = int(input())
    total = total - n
```

```
print("Ресурс исчерпан")
```

Сколько раз здесь выполнится цикл? Неизвестно, все зависит от вводимых значений. Поэтому у цикла со счетчиком известно количество итераций, а у цикла без счетчика – нет.

Самое главное для цикла `while` – чтобы в его теле происходили изменения значений переменных, которые проверяются в его заголовке, и чтобы хоть когда-нибудь наступил случай, когда логическое выражение в заголовке возвращает `False`. Иначе произойдет зацикливание.

Примечание 1. Не обязательно в выражениях `total = total - n` и `i = i + 1` повторять одну и ту же переменную. В Python допустим сокращенный способ записи подобных выражений: `total -= n` и `i += 1`.

Примечание 2. При использовании счетчика он не обязательно должен увеличиваться на единицу, а может изменяться в любую сторону на любое значение. Например, если надо вывести числа кратные пяти от 100 до 0, то изменение счетчика будет таким `i = i - 5`, или `i -= 5`.

Примечание 3. Для счетчика не обязательно использовать переменную с идентификатором *i*. Можно назвать переменную-счетчик как угодно. Однако так принято в программировании, что счетчики обозначают именами *i* и *j* (иногда одновременно требуются два счетчика).

Практическая работа

1. Измените последний код из урока так, чтобы переменная *total* не могла уйти в минус. Например, после предыдущих вычитаний ее значение стало равным 25. Пользователь вводит число 30. Однако программа не выполняет вычитание, а выводит сообщение о недопустимости операции, после чего осуществляет выход из цикла.
2. Используя цикл `while`, выведите на экран для числа 2 его степени от 0 до 20. Возведение в степень в Python обозначается как `**`. Фрагмент вывода:

```
...
  32
  64
 128
 256
 512
1024
...
```

Урок 11. Функции в программировании

Функция в программировании представляет собой обособленный участок кода, который можно вызывать. Для этого к нему обращаются по имени, которым он был назван. При вызове происходит выполнение команд тела функции.

Функции можно сравнить с небольшими программками, которые сами по себе, то есть автономно, не исполняются, а встраиваются в обычную программу. Нередко их так и называют – подпрограммы. Других ключевых отличий функций от программ нет. Функции также при необходимости могут получать и возвращать данные. Только обычно они их получают не с ввода (клавиатуры, файла и др.), а из вызывающей программы. Сюда же они возвращают результат своей работы.

Существует множество встроенных в язык программирования функций. С некоторыми такими в Python мы уже сталкивались. Это `print()`, `input()`, `int()`, `float()`, `str()`, `type()`. Код их тела нам не виден, он где-то "спрятан внутри языка". Нам же предоставляется только интерфейс – имя функции, что она возвращает, информация об особенностях передачи в нее данных-аргументов.

С другой стороны, программист всегда может определять свои функции. Их называют пользовательскими. В данном случае под "пользователем" понимают программиста, а не того, кто использует программу. Разберемся, зачем нам эти функции, и как их создавать.

Предположим, надо три раза подряд запрашивать на ввод пару чисел и складывать их. С этой целью можно использовать цикл:

```
i = 0
while i < 3:
    a = int(input())
    b = int(input())
    print(a + b)
    i += 1
```

Однако, что если перед каждым запросом чисел, надо выводить надпись, зачем они нужны, и каждый раз эта надпись разная. Мы не можем прервать цикл, а затем вернуться к тому же циклу обратно. Придется отказаться от него, и тогда получится длинный код, содержащий в разных местах одинаковые участки:

```
print("Сколько бананов и ананасов для обезьян?")
a = int(input())
b = int(input())
print("Всего", a + b, "шт.")

print("Сколько жуков и червей для ежей?")
a = int(input())
b = int(input())
```

```
print("Всего", a + b, "шт.")

print("Сколько рыб и моллюсков для выдр?")
a = int(input())
b = int(input())
print("Всего", a + b, "шт.")
```

Пример исполнения программы:

```
Сколько бананов и ананасов для обезьян?
15
5
Всего 20 шт.
Сколько жуков и червей для ежей?
50
12
Всего 62 шт.
Сколько рыб и моллюсков для выдр?
16
8
Всего 24 шт.
```

Определение функций позволяет решить проблему дублирования кода в разных местах программы. Благодаря функциям можно исполнять один и тот же участок кода не сразу, а только тогда, когда он понадобится.

Определение функции. Оператор def

В языке программирования Python функции определяются с помощью оператора `def`.

Рассмотрим код:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a + b, "шт.")
```

Это пример определения функции. Как и другие сложные инструкции вроде условного оператора и цикла функция состоит из заголовка и тела. Заголовок оканчивается двоеточием и переходом на новую строку. Тело имеет отступ.

Ключевое слово `def` сообщает интерпретатору, что перед ним определение функции. За `def` следует имя функции. Оно может быть любым, также как и всякий идентификатор, например, переменная. В программировании весьма желательно давать всему осмысленные имена. Так в данном случае функция названа "посчитать_еду" в переводе на русский.

После имени функции ставятся скобки. В приведенном примере они пустые. Это значит, что функция не принимает никакие данные из вызывающей ее программы. Однако она могла бы их принимать, и тогда в скобках были бы указаны так называемые параметры.

После двоеточия следует тело, содержащее инструкции, которые выполняются при вызове функции. Следует различать определение функции и ее вызов. В программном коде они не рядом и не вместе. Можно определить функцию, но ни разу ее не вызвать. Нельзя вызвать функцию, которая не была определена. Определив функцию, но ни разу не вызвав ее, вы никогда не выполните ее тела.

Вызов функции

Рассмотрим полную версию программы с функцией:

```
def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a+b, "шт.")


print("Сколько бананов и ананасов для обезьян?")
count_food()

print("Сколько жуков и червей для ежей?")
count_food()

print("Сколько рыб и моллюсков для выдр?")
count_food()
```

После вывода на экран каждого информационного сообщения осуществляется вызов функции, который выглядит просто как упоминание ее имени со скобками. Поскольку в функцию мы ничего не передаем скобки опять же пустые. В приведенном коде функция вызывается три раза.

Когда функция вызывается, поток выполнения программы переходит к ее определению и начинает исполнять ее тело. После того, как тело функции исполнено, поток выполнения возвращается в основной код в то место, где функция вызывалась. Далее исполняется следующее за вызовом выражение.

В языке Python определение функции должно предшествовать ее вызовам. Это связано с тем, что интерпретатор читает код строка за строкой и о том, что находится ниже по течению, ему еще неизвестно. Поэтому если вызов функции предшествует ее определению, то возникает ошибка (выбрасывается исключение `NameError`):

```
print("Сколько бананов и ананасов для обезьян?")
count_food()
```

```
print("Сколько жуков и червей для ежей?")
count_food()

print("Сколько рыб и моллюсков для выдр?")
count_food()

def count_food():
    a = int(input())
    b = int(input())
    print("Всего", a + b, "шт.")
```

Результат:

```
Сколько бананов и ананасов для обезьян?
Traceback (most recent call last):
  File "test.py", line 2, in <module>
    count_food()
NameError: name 'count_food' is not defined
```

Для многих компилируемых языков это не обязательное условие. Там можно определять и вызывать функцию в произвольных местах программы.

Функции придают программе структуру

Польза функций не только в возможности многократного вызова одного и того же кода из разных мест программы. Не менее важно, что благодаря им программа обретает истинную структуру. Функции как бы разделяют ее на обособленные части, каждая из которых выполняет свою конкретную задачу.

Пусть надо написать программу, вычисляющую площади разных фигур. Пользователь указывает, площадь какой фигуры он хочет вычислить. После этого вводит исходные данные. Например, длину и ширину в случае прямоугольника. Чтобы разделить поток выполнения на несколько ветвей, будем использовать оператор **if-elif-else**:

```
figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")

if figure == '1':
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))
elif figure == '2':
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))
elif figure == '3':
```

```
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r ** 2))
else:
    print("Ошибка ввода")
```

Здесь нет никаких функций, и все прекрасно. Но напишем вариант с функциями:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))


def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))


def circle():
    r = float(input("Радиус: "))
    print("Площадь: %.2f" % (3.14 * r ** 2))


figure = input("1-прямоугольник, 2-треугольник, 3-круг: ")

if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
elif figure == '3':
    circle()
else:
    print("Ошибка ввода")
```

Он кажется сложнее, а каждая из трех функций вызывается всего один раз. Однако из общей логики программы как бы убраны и обособлены инструкции для нахождения площадей. Программа теперь состоит из отдельных "кирпичиков Лего". В основной ветке мы можем комбинировать их как угодно. Она играет роль управляющего механизма.

Если нам когда-нибудь захочется вычислять площадь треугольника по формуле Герона, а не через высоту, то не придется искать код во всей программе (представьте, что она состоит из тысяч строк кода как реальные программы). Мы пойдем к месту определения функций и изменим тело одной из них.

Если понадобиться использовать эти функции в какой-нибудь другой программе, то мы сможем импортировать их туда, сославшись на данный файл с кодом (как это делается в Python, будет рассмотрено позже).

Практическая работа

В программировании можно из одной функции вызывать другую. Для иллюстрации этой возможности напишите программу по следующему описанию.

Основная ветка программы, не считая заголовков функций, состоит из одной строки кода. Это вызов функции `test()`. В ней запрашивается на ввод целое число. Если оно положительное, то вызывается функция `positive()`, тело которой содержит команду вывода на экран слова "Положительное". Если число отрицательное, то вызывается функция `negative()`, ее тело содержит выражение вывода на экран слова "Отрицательное".

Понятно, что вызов `test()` должен следовать после определения функций. Однако имеет ли значение порядок определения самих функций? То есть должны ли определения `positive()` и `negative()` предшествовать `test()` или могут следовать после него? Проверьте вашу гипотезу, поменяв объявления функций местами. Попробуйте объяснить результат.

Урок 12. Локальные и глобальные переменные

В программировании особое внимание уделяется концепции о локальных и глобальных переменных, а также связанное с ними представление об областях видимости. Соответственно, локальные переменные видны только в локальной области видимости, которой может выступать отдельно взятая функция. Глобальные переменные видны во всей программе. "Видны" – значит, известны, доступны. К ним можно обратиться по имени и получить связанное с ними значение.

К глобальной переменной можно обратиться из локальной области видимости. К локальной переменной нельзя обратиться из глобальной области видимости, потому что локальная переменная существует только в момент выполнения тела функции. При выходе из нее, локальные переменные исчезают. Компьютерная память, которая под них отводилась, освобождается. Когда функция будет снова вызвана, локальные переменные будут созданы заново.

Вернемся к нашей программе из прошлого урока, немного упростив ее для удобства:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    print("Площадь: %.2f" % (a * b))
```

```
def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    print("Площадь: %.2f" % (0.5 * a * h))


figure = input("1-прямоугольник, 2-треугольник: ")

if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

Сколько здесь переменных? Какие из них являются глобальными, а какие – локальными?

Здесь пять переменных. Глобальной является только *figure*. Переменные *a* и *b* из функции `rectangle`, а также *a* и *h* из `triangle` – локальные. При этом локальные переменные с одним и тем же идентификатором *a*, но объявленные в разных функциях, – разные переменные.

Следует отметить, что идентификаторы `rectangle` и `triangle`, хотя и не являются именами переменных, а представляют собой имена функций, также имеют область видимости. В данном случае она глобальная, так как функции объявлены непосредственно в основной ветке программы.

В приведенной программе к глобальной области видимости относятся заголовки объявлений функций, объявление и присваивание переменной *figure*, конструкция условного оператора.

К локальной области относятся тела функций. Если, находясь в глобальной области видимости, мы попытаемся обратиться к локальной переменной, то возникнет ошибка:

```
...
elif figure == '2':
    triangle()

print(a)
```

Пример выполнения:

```
1-прямоугольник, 2-треугольник: 2
Основание: 4
Высота: 5
Площадь: 10.00
Traceback (most recent call last):
  File "test.py", line 17, in <module>
    print(a)
NameError: name 'a' is not defined
```

Однако мы можем обращаться из функций к глобальным переменным:

```
def rectangle():
    a = float(input("Ширина %s: " % figure))
    b = float(input("Высота %s: " % figure))
    print("Площадь: %.2f" % (a * b))

def triangle():
    a = float(input("Основание %s: " % figure))
    h = float(input("Высота %s: " % figure))
    print("Площадь: %.2f" % (0.5 * a * h))


figure = input("1-прямоугольник, 2-треугольник: ")

if figure == '1':
    rectangle()
elif figure == '2':
    triangle()
```

```
1-прямоугольник, 2-треугольник: 1
Ширина 1: 6.35
Высота 1: 2.75
Площадь: 17.46
```

В данном случае из тел функций происходит обращение к имени *figure*, которое, из-за того, что было объявлено в глобальной области видимости, видимо во всей программе.

Наши функции не совсем идеальны. Они должны вычислять площади фигур, но выводить результат на экран им не следовало бы. Вполне вероятна ситуация, когда результат нужен в программе для каких-то дальнейших вычислений, а выводить ли его на экран – вопрос второстепенный.

Если функции не будут выводить, а только вычислять результат, то его надо где-то сохранить для дальнейшего использования. Для этого подошли бы глобальные переменные. В них можно записать результат. Напишем программу вот так:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    result = a * b

def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    result = 0.5 * a * h
```

```
result = 0

figure = input("1-прямоугольник, 2-треугольник: ")

if figure == '1':
    rectangle()
elif figure == '2':
    triangle()

print("Площадь: %.2f" % result)
```

Итак, мы ввели в программу глобальную переменную *result* и инициировали ее нулем. В функциях ей присваивается результат вычислений. В конце программы ее значение выводится на экран. Мы ожидаем, что программа будет прекрасно работать. Однако…

```
1-прямоугольник, 2-треугольник: 2
Основание: 6
Высота: 4.5
Площадь: 0.00
```

… что-то пошло не так.

Дело в том, что в Python присвоение значения переменной совмещено с ее объявлением. (Во многих других языках это не так.) Поэтому, когда имя *result* впервые упоминается в локальной области видимости, и при этом происходит присваивание ей значения, то создается локальная переменная *result*. Это другая переменная, никак не связанная с глобальной *result*.

Когда функция завершает свою работу, то значение локальной *result* теряется, а глобальная не была изменена.

Когда мы вызывали внутри функции переменную *figure*, то ничего ей не присваивали. Наоборот, мы запрашивали ее значение. Интерпретатор Питона искал такую переменную сначала в локальной области видимости и не находил. После этого шел в глобальную и находил.

В случае с *result* он ничего не ищет. Он выполняет вычисления справа от знака присваивания, создает локальную переменную *result*, связывает ее с полученным значением.

На самом деле можно принудительно обратиться к глобальной переменной. Для этого существует команда `global`:

```
def rectangle():
    a = float(input("Ширина: "))
    b = float(input("Высота: "))
    global result
    result = a * b
```

```
def triangle():
    a = float(input("Основание: "))
    h = float(input("Высота: "))
    global result
    result = 0.5 * a * h


result = 0

figure = input("1-прямоугольник, 2-треугольник: ")

if figure == '1':
    rectangle()
elif figure == '2':
    triangle()

print("Площадь: %.2f" % result)
```

В таком варианте программа будет работать правильно. Однако менять значения глобальных переменных в теле функции – плохая практика. В больших программах трудно отследить, где, какая функция и почему изменила их значение. Программист смотрит на исходное значение глобальной переменной и может подумать, что оно остается таким же. Сложно заметить, что какая-то функция поменяла его. Подобное может привести к логическим ошибкам.

Чтобы избавиться от необходимости использовать глобальные переменные, для функций существует возможность возврата результата своей работы в основную ветку программы. И уже это полученное из функции значение можно присвоить глобальной переменной в глобальной области видимости. То есть программист видит изменение значения переменной в основной ветке программы. Это делает программу более понятной.

Как функция принимает и возвращает данные, будет рассмотрено в следующих уроках.

Практическая работа

В языке Python можно внутри одной функции определять другую. Напишите программу по следующему описанию. В основной ветке программы вызывается функция `cylinder()`, которая вычисляет площадь цилиндра. В теле `cylinder` определена функция `circle`, вычисляющая площадь круга по формуле πr^2. В теле `cylinder` у пользователя спрашивается, хочет ли он получить только площадь боковой поверхности цилиндра, которая вычисляется по формуле $2\pi rh$, или полную площадь цилиндра. В последнем случае к площади боковой поверхности цилиндра должен добавляться удвоенный результат вычислений функции `circle()`.

Как вы думаете, можно ли из основной ветки программы вызвать функцию, вложенную в другую функцию? Почему?

Урок 13. Возврат значений из функции. Оператор return

Функции могут передавать какие-либо данные из своих тел в основную ветку программы. Говорят, что функция возвращает значение. В большинстве языков программирования, в том числе Python, выход из функции и передача данных в то место, откуда она была вызвана, выполняется оператором `return`.

Если интерпретатор Питона, выполняя тело функции, встречает `return`, то он "забирает" значение, указанное после этой команды, и "уходит" из функции.

```
def cylinder():
    r = float(input())
    h = float(input())
    # площадь боковой поверхности цилиндра:
    side = 2 * 3.14 * r * h
    # площадь одного основания цилиндра:
    circle = 3.14 * r**2
    # полная площадь цилиндра:
    full = side + 2 * circle
    return full


square = cylinder()
print(square)
```

Пример выполнения:

```
3
7
188.4
```

В данной программе в основную ветку из функции возвращается значение локальной переменной *full*. Не сама переменная, а ее значение, в данном случае – какое-либо число, полученное в результате вычисления площади цилиндра.

В основной ветке программы это значение присваивается глобальной переменной *square*. То есть выражение `square = cylinder()` выполняется так:

1. Вызывается функция `cylinder()`.
2. Из нее возвращается значение.
3. Это значение присваивается переменной *square*.

Не обязательно присваивать результат переменной, его можно сразу вывести на экран:

```
...
print(cylinder())
```

Здесь число, полученное из `cylinder()`, непосредственно передается функции `print()`. Если мы в программе просто напишем `cylinder()`, не присвоив полученные данные переменной или не передав их куда-либо дальше, то эти данные будут потеряны. Но синтаксической ошибки не будет.

В функции может быть несколько операторов **`return`**. Однако всегда выполняется только один из них. Тот, которого первым достигнет поток выполнения. Допустим, мы решили обработать исключение, возникающее на некорректный ввод. Пусть тогда в ветке **`except`** обработчика исключений происходит выход из функции без всяких вычислений и передачи значения:

```
def cylinder():
    try:
        r = float(input())
        h = float(input())
    except ValueError:
        return
    side = 2 * 3.14 * r * h
    circle = 3.14 * r**2
    full = side + 2 * circle
    return full


print(cylinder())
```

Если попытаться вместо цифр ввести буквы, то сработает **`return`**, вложенный в **`except`**. Он завершит выполнение функции, так что все нижеследующие вычисления, в том числе `return full`, будут опущены. Пример выполнения:

```
r
None
```

Но постойте! Что это за слово **`None`**, которое нам вернул "пустой" **`return`**? Это ничего, такой объект – "ничто". Он принадлежит классу **`NoneType`**. До этого мы знали четыре типа данных, они же классы: **`int`**, **`float`**, **`str`**, **`bool`**. Пришло время пятого.

Когда после **`return`** ничего не указывается, то по умолчанию считается, что там стоит объект **`None`**. При желании мы можете явно писать `return None`.

Более того. Ранее мы рассматривали функции, которые вроде бы не возвращали никакого значения, потому что в них не было оператора **`return`**. На самом деле возвращали, просто мы не обращали на него внимание, не присваивали никакой переменной и не выводили на экран. В

Python всякая функция что-либо возвращает. Если в ней нет оператора `return`, то она возвращает `None`. То же самое, как если в ней имеется "пустой" `return`.

Возврат нескольких значений

В Питоне позволительно возвращать из функции несколько объектов, перечислив их через запятую после команды `return`:

```
def cylinder():
    r = float(input())
    h = float(input())
    side = 2 * 3.14 * r * h
    circle = 3.14 * r ** 2
    full = side + 2 * circle
    return side, full


s_cyl, f_cyl = cylinder()
print("Площадь боковой поверхности %.2f" % s_cyl)
print("Полная площадь %.2f" % f_cyl)
```

Из функции `cylinder()` возвращаются два значения. Первое из них присваивается переменной *s_cyl*, второе – *f_cyl*. Возможность такого группового присвоения – особенность Python, обычно не характерная для других языков:

```
>>> a, b, c = 10, 15, 19
>>> a
10
>>> b
15
>>> c
19
```

Фокус здесь в том, что перечисление значений через запятую (например, `10, 15, 19`) создает объект типа `tuple`. На русский переводится как "кортеж". Это разновидность структур данных, которые будут изучены позже.

Когда же кортеж присваивается сразу нескольким переменным, то происходит сопоставление его элементов соответствующим в очереди переменным. Это называется распаковкой.

Таким образом, когда из функции возвращается несколько значений, на самом деле из нее возвращается один объект класса `tuple`. Перед возвратом эти несколько значений упаковываются в кортеж. Если же после оператора `return` стоит только одна переменная или объект, то ее/его тип сохраняется как есть.

Распаковка не является обязательной. Будет работать и так:

```
…
print(cylinder())
```

```
4
3
(75.36, 175.84)
```

На экран выводится кортеж, о чем говорят круглые скобки. Его также можно присвоить одной переменной, а потом вывести ее значение на экран.

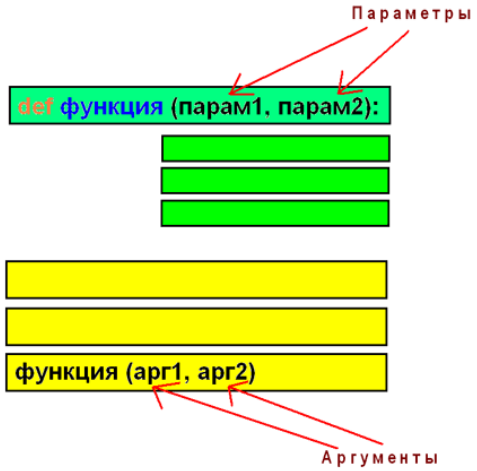
Практическая работа

1. Напишите программу, в которой вызывается функция, запрашивающая с ввода две строки и возвращающая в программу результат их конкатенации. Выведите результат на экран.
2. Напишите функцию, которая считывает с клавиатуры числа и перемножает их до тех пор, пока не будет введен 0. Функция должна возвращать полученное произведение. Вызовите функцию и выведите на экран результат ее работы.

Урок 14. Параметры и аргументы функции

В программировании функции могут не только возвращать данные, но также принимать их, что реализуется с помощью так называемых параметров, которые указываются в скобках в заголовке функции. Количество параметров может быть любым.

Параметры представляют собой локальные переменные, которым присваиваются значения в момент вызова функции. Конкретные значения, которые передаются в функцию при ее вызове, будем называть аргументами. Следует иметь в виду, что встречается иная терминология. Например, формальные параметры и фактические параметры. В Python же обычно все называют аргументами. Рассмотрим схему и поясняющий ее пример:




```
>>> def do_math(a, b):
...     a = (a + 1) / 2
...     b = b + 3
...     print(a * b)
...
>>> num1 = 99
>>> num2 = 4
>>> do_math(num1, num2)
350.0
```

Когда функция вызывается, то ей передаются аргументы. В примере указаны глобальные переменные *num1* и *num2*. Однако на самом деле передаются не эти переменные, а их значения. В данном случае числа 99 и 4. Другими словами, мы могли бы писать `do_math(99, 4)`. Разницы не было бы.

Когда интерпретатор переходит к функции, чтобы начать ее исполнение, он присваивает переменным-параметрам переданные в функцию значения-аргументы. В примере переменной *a* будет присвоено 99, *b* будет присвоено 4.

Изменение значений *a* и *b* в теле функции никак не скажется на значениях переменных *num1* и *num2*. Они останутся прежними. В Python такое поведение характерно для неизменяемых типов данных, к которым относятся, например, числа и строки. Говорят, что в функцию данные передаются по значению. Можно сказать, когда *a* присваивалось число 99, то это было уже другое число, не то, на которое ссылается переменная *num1*. Число 99 было скопировано и помещено в отдельную ячейку памяти для переменной *a*.

На самом деле переменная *a* в момент присваивания значения может указывать на то же число 99, что и переменная *num1*. Однако, когда *a* в результате вычислений в теле функции получает новое значение, то связывается с другой ячейкой памяти, потому что числа относятся к неизменяемым типам данных, то есть нельзя переписать значение содержащей их ячейки. При этом переменная *num1* остается связанной со старым значением.

Существуют изменяемые типы данных. Для Питона, это, например, списки и словари. В этом случае данные передаются по ссылке. В функцию передается ссылка на них, а не сами данные. И эта ссылка связывается с локальной переменной. Изменения таких данных через локальную переменную обнаруживаются при обращении к ним через глобальную. Это есть следствие того, что несколько переменных ссылаются на одни и те же данные, на одну и ту же область памяти.

Необходимость передачи по ссылке связана в первую очередь с экономией памяти. Сложные типы данных, по сути представляющие собой структуры данных, обычно копировать не целесообразно. Однако, если надо, всегда можно сделать это принудительно.

Произвольное количество аргументов

Обратим внимание еще на один момент. Количество аргументов и параметров совпадает. Нельзя передать три аргумента, если функция принимает только два. Нельзя передать один аргумент, если функция требует два обязательных. В рассмотренном примере они обязательные.

Однако в Python у функций бывают параметры, которым уже присвоено значение по-умолчанию. В таком случае, при вызове можно не передавать соответствующие этим параметрам аргументы. Хотя можно и передать. Тогда значение по умолчанию заменится на переданное.

```
def rect_area(a, b=1):
    return a * b
```

```
rect1 = rect_area(4, 3)
rect2 = rect_area(5)

print(rect1)  # 12
print(rect2)  # 5
```

При втором вызове `rect_area()` мы указываем только один аргумент. Он будет присвоен переменной-параметру *a*. Переменная *b* будет равна 1.

Согласно правилам синтаксиса Python при определении функции параметры, которым присваивается значение по-умолчанию должны следовать (находиться сзади) за параметрами, не имеющими значений по умолчанию.

А вот при вызове функции, можно явно указывать, какое значение соответствует какому параметру. В этом случае их порядок не играет роли:

```
...
rect3 = rect_area(10, 2)
rect4 = rect_area(b=2, a=10)
print(rect3)  # 20
print(rect4)  # 20
```

В данном случае оба вызова – это вызовы с одними и теми же аргументами-значениями. Просто в первом случае сопоставление параметрам-переменным идет в порядке следования. Во-втором случаи – по ключам, которыми выступают имена параметров.

В Python определения и вызовы функций имеют и другие нюансы, рассмотрение которых мы пока опустим, так как они требуют более глубоких знаний, чем у нас есть на данный момент. Скажем лишь, что функция может быть определена так, что в нее можно не передавать ни одного аргумента или передать неопределенное множество аргументов:

```
def few_or_many(*a):
    print(a)


few_or_many(1)
few_or_many('1', 1, 2, 'abc')
few_or_many()
```

Результат:

```
(1,)
('1', 1, 2, 'abc')
()
```

Опять же, судя по скобкам, здесь возникает упомянутый в прошлом уроке кортеж.

Практическая работа

Напишите программу, в которой определена функция *int_test*, имеющая один параметр. Функция проверяет, можно ли переданное ей значение преобразовать к целому числу. Если можно, возвращает логическое `True`. Если нельзя – `False`.

В основной ветке программы присвойте переменной *s* то, что пользователь вводит с клавиатуры. Вызовите функцию `int_test()`, передав ей значение *s*. Если функция возвращает истину, преобразуйте строку *s* в число *n* и выведите на экран значение `n + 10`.

Урок 15. Встроенные функции

Язык Python включает много уже определенных, то есть встроенных в него, функций. Программист не видит их определений, они скрыты где-то в "недрах" языка. Достаточно знать, что эти функции принимают и что возвращают, то есть их интерфейс.

Ряд встроенных функций, касающихся ввода-вывода и типов данных, мы уже использовали. Это `print`, `input`, `int`, `float`, `str`, `bool`, `type`. Перечень всех встроенных в Python функций можно найти в официальной документации по языку:
https://docs.python.org/3/library/functions.html .

В этом уроке рассмотрим следующие функции, условно разбив их на группы:

- функции для работы с символами – `ord`, `chr`, `len`
- математические функции – `abs`, `round`, `divmod`, `pow`, `max`, `min`, `sum`

Функция `ord` позволяет получить номер символа по таблице Unicode. Соответственно, принимает она в качестве аргумента одиночный символ, заключенный в кавычки:

```
>>> ord('z')
122
>>> ord('ф')
1092
>>> ord('@')
64
```

Функция `chr` выполняет обратное действие. Она позволяет получить символ по его номеру:

```
>>> chr(87)
'W'
>>> chr(1049)
'Й'
>>> chr(10045)
'❽'
```

Чтобы не путать **ord** и **chr**, помните, что функция – это действие. Ее имя как бы отвечает на вопрос "Что сделать?". Order – это порядок. Значит, мы хотим получить порядковый номер элемента в ряду. А чтобы получить номер, должны передать символ. Character – это символ. Значит, мы хотим получить символ. Поэтому должны передать порядковый номер.

Функция **len** в качестве аргумента принимает сложный объект и подсчитывает в нем количество составляющих его простых элементов. Числа – это простые объекты, их нельзя передавать в **len**. Строки можно:

```
>>> len('abc')
3
>>> s1 = '------'
>>> s2 = '_______'
>>> len(s1) > len(s2)
False
>>> len(s1)
6
>>> len(s2)
7
```

Кроме строк в **len** можно передавать другие, еще не изученные нами, структуры данных.

Функция **abs** возвращает абсолютное значение числа:

```
>>> abs(-2.2)
2.2
>>> abs(9)
9
```

Если требуется округлить вещественное число до определенного знака после запятой, то следует воспользоваться функцией **round**:

```
>>> a = 10/3
>>> a
3.3333333333333335
>>> round(a, 2)
3.33
>>> round(a)
3
```

Если второй аргумент не задан, то округление идет до целого числа. Есть одна специфическая особенность этой функции. Второй аргумент может быть отрицательным числом. В этом случае округляться начинают единицы, десятки, сотни и т. д., то есть целая часть:

```
>>> round(5321, -1)
5320
>>> round(5321, -3)
5000
>>> round(5321, -4)
10000
```

Функция именно округляет согласно правилу округления из математики, а не отбрасывает. Поэтому 5 тысяч неожиданно округляются до десяти.

```
>>> round(3.76, 1)
3.8
>>> round(3.72, 1)
3.7
>>> round(3.72)
4
>>> round(3.22)
3
```

Если нужно просто избавиться от дробной части без округления, следует воспользоваться функцией `int`:

```
>>> int(3.78)
3
```

Нередко функцию `round` используют совместно с функцией `print`, избегая форматирования вывода:

```
>>> a = 3.45673
>>> print("Number: %.2f" % a)
Number: 3.46
>>> print("Number:", round(a, 2))
Number: 3.46
```

В последнем случае код выглядит более ясным.

Функция `divmod` выполняет одновременно деление нацело и нахождение остатка от деления:

```
>>> divmod(10, 3)
(3, 1)
>>> divmod(20, 7)
(2, 6)
```

Возвращает она кортеж. В некоторых других языках встречаются две отдельные функции: `div` и `mod`. Первая делит нацело, вторая находит остаток от целочисленного деления (деления по

модулю). В Python и многих других языках для этого используются специальные символы-операнды:

```
>>> 10 // 3
3
>>> 10 % 3
1
```

Функция `pow` возводит в степень. Первое число – основание, второе – показатель:

```
>>> pow(3, 2)
9
>>> pow(2, 4)
16
```

То же самое можно проделать так:

```
>>> 3**2
9
>>> 2**4
16
```

Однако `pow` может принимать третий необязательный аргумент. Это число, на которое делится по модулю результат возведения в степень:

```
>>> pow(2, 4, 4)
0
>>> 2**4 % 4
0
```

Преимуществом первого способа является его более быстрое выполнение.

Функции `max`, `min` и `sum` находят соответственно максимальный, минимальный элемент и сумму элементов аргумента:

```
>>> max(10, 12, 3)
12
>>> min(10, 12, 3, 9)
3
>>> a = (10, 12, 3, 10)
>>> sum(a)
35
```

В `sum` нельзя передать перечень элементов, должна быть структура данных, например, кортеж. В `min` и `max` также чаще передают один так называемый итерируемый объект:

```
>>> max(a)
12
```

Практическая работа

1. Напишите программу, которая в цикле запрашивает у пользователя номера символов по таблице Unicode и выводит соответствующие им символы. Завершает работу при вводе нуля.
2. Напишите программу, которая измеряет длину введенной строки. Если строка длиннее десяти символов, то выносится предупреждение. Если короче, то к строке добавляется столько символов `*`, чтобы ее длина составляла десять символов, после чего новая строка должна выводиться на экран.
3. Напишите программу, которая запрашивает у пользователя шесть вещественных чисел. На экран выводит минимальное и максимальное из них, округленные до двух знаков после запятой. Выполните задание без использования встроенных функций `min` и `max`.

Урок 16. Модули

Встроенные в язык программирования функции доступны сразу. Чтобы их вызывать, не надо выполнять никаких дополнительных действий. Однако за время существования любого популярного языка на нем было написано столько функций и классов, которые оказались востребованными множеством программистов и в разных областях, что включить весь этот объем кода в сам язык если возможно, то нецелесообразно.

Чтобы разрешить проблему доступа к дополнительным возможностям языка, в программировании стало общепринятой практикой использовать так называемые модули, пакеты и библиотеки. Каждый модуль содержит коллекцию функций и классов, предназначенных для решения задач из определенной области. Так в модуле `math` языка Python содержатся математические функции, модуль `random` позволяет генерировать псевдослучайные числа, в модуле `datetime` содержатся классы для работы с датами и временем, модуль `sys` предоставляет доступ к системным переменным и т. д.

Количество модулей для языка Python огромно, что связано с популярностью языка. Часть модулей собрана в так называемую стандартную библиотеку. Стандартная она потому, что поставляется вместе с установочным пакетом. Однако существуют сторонние библиотеки. Они скачиваются и устанавливаются отдельно.

Для доступа к функционалу модуля, его надо импортировать в программу. После импорта интерпретатор "знает" о существовании дополнительных классов и функций и позволяет ими пользоваться.

В Питоне импорт осуществляется командой **`import`**. При этом существует несколько способов импорта. Рассмотрим работу с модулем на примере **`math`**. Итак,

```
>>> import math
```

Ничего не произошло. Однако в глобальной области видимости появилось имя **`math`**. Если до импорта вы упомянули бы имя **`math`**, то возникла бы ошибка **`NameError`**. Теперь же

```
>>> math
<module 'math' (built-in)>
```

В программе завелся объект **`math`**, относящийся к классу **`module`**.

Чтобы увидеть перечень функций, входящих в этот модуль, воспользуемся встроенной в Python функцией `dir()`, передав ей в качестве аргумента имя модуля:

```
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh',
'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos',
'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'lcm', 'ldexp', 'lgamma',
'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'nextafter', 'perm', 'pi', 'pow',
'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc', 'ulp']
```

Проигнорируем имена с двойными подчеркиваниями. Все остальное – имена функций и констант (переменных, которые не меняют своих значений), включенных в модуль **`math`**. Чтобы вызвать функцию из модуля, надо впереди написать имя модуля, поставить точку, далее указать имя функции, после чего в скобках передать аргументы, если они требуются. Например, чтобы вызвать функцию **`pow`** из **`math`**, надо написать так:

```
>>> math.pow(2, 2)
4.0
```

Обратите внимание, эта другая функция `pow()`, не та, что встроена в сам язык. "Обычная" функция `pow()` возвращает целое, если аргументы целые числа:

```
>>> pow(2, 2)
4
```

Для обращения к константе скобки не нужны:

```
>>> math.pi
3.141592653589793
```

Если мы не знаем, что делает та или иная функция, то можем получить справочную информацию о ней с помощью встроенной в язык Python функции `help()`:

```
>>> help(math.gcd)

Help on built-in function gcd in module math:

gcd(*integers)
    Greatest Common Divisor.
(END)
```

Для выхода из интерактивной справки надо нажать клавишу `q`. В данном случае сообщается, что функция вычисляет наибольший общий делитель. Описание модулей и их содержания также можно посмотреть в официальной документации на сайте python.org.

Второй способ импорта – это когда импортируется не сам модуль, а только необходимые функции из него.

```
>>> from math import gcd, sqrt, hypot
```

Перевести можно как "из модуля **`math`** импортировать функции **`gcd`**, **`sqrt`** и **`hypot`**".

В таком случае при их вызове не надо перед именем функции указывать имя модуля:

```
>>> gcd(100, 150)
50
>>> sqrt(16)
4.0
>>> hypot(3, 4)
5.0
```

Чтобы импортировать сразу все функции из модуля:

```
>>> from math import *
```

Импорт через **`from`** не лишен недостатка. В программе уже может быть идентификатор с таким же именем, как имя одной из импортируемых функций или констант. Ошибки не будет, но одно из них окажется "затерто":

```
>>> pi = 3.14
>>> from math import pi
>>> pi
3.141592653589793
```

Здесь исчезает значение 3.14, присвоенное переменной *pi*. Это имя теперь указывает на число из модуля **`math`**. Если импорт сделать раньше, чем присвоение значения *pi*, то будет все наоборот:

```
>>> from math import pi
>>> pi = 3.14
>>> pi
3.14
```

В связи с этим более опасен именно импорт всех функций. Так как в этом случае очень легко не заметить подмены значений идентификаторов.

Однако можно изменить имя идентификатора из модуля на какое угодно:

```
>>> from math import pi as P
>>> P
3.141592653589793
>>> pi
3.14
```

В данном случае константа **`pi`** из модуля импортируется под именем *P*. Другой смысл подобных импортов – сокращение имен, так как есть модули с длинными именами, а имена функций и классов в них еще длиннее. Если в программу импортируется всего пара сущностей, и они используются в ней часто, то имеет смысл переименовать их на более короткий вариант. Сравните:

```
>>> import calendar
>>> calendar.weekheader(2)
'Mo Tu We Th Fr Sa Su'
```

и

```
>>> from calendar import weekheader as week
>>> week(3)
'Mon Tue Wed Thu Fri Sat Sun'
```

Во всех остальных случаях лучше оставлять идентификаторы содержимого модуля в пространстве имен самого модуля и получать доступ к ним через имя модуля, то есть выполнять импорт командой `import имя_модуля`, а вызывать, например, функции через `имя_модуля.имя_функции()`.

Практическая работа. Создание собственного модуля

Программист на Python всегда может создать собственный модуль, чтобы использовать его в нескольких своих программах или даже предоставить в пользование всему миру. В качестве

тренировки создадим модуль с функциями для вычисления площадей прямоугольника, треугольника и круга:

```
from math import pi, pow

def rectangle(a, b):
    return round(a * b, 2)

def triangle(a, h):
    return round(0.5 * a * h, 2)

def circle(r):
    return round(pi * pow(r, 2), 2)
```

Здесь также иллюстрируется принцип, что один модуль может импортировать другие. В данном случае импортируются функции из модуля `math`.

Поместите данный код в отдельный файл *square.py*. Однако куда поместить сам файл?

Когда интерпретатор Питона встречает команду импорта, то просматривает на наличие файла-модуля определенные каталоги. Их перечень можно увидеть по содержимому `sys.path`:

```
>>> import sys
>>> sys.path
['', '/usr/lib/python310.zip', '/usr/lib/python3.10', '/usr/lib/python3.10/lib-
dynload', '/home/pl/.local/lib/python3.10/site-packages',
'/usr/local/lib/python3.10/dist-packages', '/usr/lib/python3/dist-packages']
```

Это список адресов в Linux. В Windows он будет несколько другим. Первый элемент – пустая строка, что обозначает текущий каталог, т. е. место, где сохранена сама программа, импортирующая модуль. Если вы сохраните файл-модуль и файл-программу в одном каталоге, то интерпретатор без труда найдет модуль.

Также модуль можно положить в любой другой из указанных в списке каталогов. Тогда он будет доступен для всех программ на Python, а также его можно будет импортировать в интерактивном режиме.

Можно добавить в `sys.path` свой каталог. Однако в этом случае либо код программы должен содержать команды изменения значения `sys.path`, либо надо править конфигурационный файл операционной системы. В большинстве случаев лучше так не делать.

Поместите файл *square.py* в тот же каталог, где будет исполняемая программа. Ее код должен включать инструкцию импорта модуля *square* (при импорте расширение файла не указывается) и вызов той функции и с теми аргументами, которые ввел пользователь. То есть у пользователя надо спросить, площадь какой фигуры он хочет вычислить. Далее запросить у него аргументы для соответствующей функции. Передать их в функцию из модуля *square*, а полученный оттуда результат вывести на экран.

Примечание. Исполнение модуля как самостоятельного скрипта, а также создание строк документации, которые отображает встроенная в Python функция `help()`, будут рассмотрены в курсе объектно-ориентированного программирования.

Урок 17. "Случайные" числа в Python – random, randint и randrange

В компьютерных программах нередко требуется эмуляция случайности. Например, при разработке игр. Если в программе имеется некий генератор, то есть производитель, случайного числа, то, используя полученное таким образом число, можно выбирать ту или иную ветку выполнения программы, или произвольный объект из коллекции. Другими словами, главное – сгенерировать число. Эмуляция случайности иного рода основывается на нем.

Мы наверняка не знаем, есть ли в природе случайность, или она нам только кажется из-за ограниченности наших знаний. Мы только знаем, что в программировании настоящей случайности нет. Неоткуда взяться произвольному числу, нельзя запрограммировать его появление из ниоткуда. Можно лишь создать программу, которая в результате применения сложной формулы к "зерну" будет выдавать число, и нам будет казаться, что это число случайно.

"Зерно" – это исходные данные для формулы. Им может быть, например, системное время в миллисекундах, которое постоянно меняется. Следовательно, "зерно" будет постоянно разным. Или программист может задавать его самостоятельно.

Подобную программу (в реальности модуль или функцию) называют генератором псевдослучайных чисел. В состав стандартной библиотеки языка Python входит модуль **`random`**. Он содержит множество функций, связанных с эмуляцией случайности (например, "перемешивание" элементов последовательности), а не только функции генерации псевдослучайных чисел.

В этом уроке будут рассмотрены функции `random()`, `randrange()` и `randint()` из модуля **`random`**. Обратите внимание, что модуль **`random`** содержит одноименную функцию `random()`. Так бывает.

Чтобы обращаться к функциям, надо импортировать модуль **`random`**:

```
>>> import random
```

Или импортировать отдельные функции из него:

```
>>> from random import random, randrange, randint
```

Функции для получения целых "случайных" чисел – randint() и randrange()

Функции `randint()` и `randrange()` генерируют псевдослучайные целые числа. Первая из них наиболее простая и всегда принимает только два аргумента – пределы целочисленного диапазона, из которого выбирается любое число:

```
>>> random.randint(0, 10)
6
```

или (если импортировались отдельные функции):

```
>>> randint(100, 200)
110
```

В случае `randint()` обе границы включаются в диапазон, т. е. на языке математики отрезок описывается как [*a*; *b*].

Числа могут быть отрицательными:

```
>>> random.randint(-100, 10)
-83
>>> random.randint(-100, -10)
-38
```

Но первое число всегда должно быть меньше или равно второму. То есть `a <= b`.

Функция `randrange()` сложнее. Она может принимать один аргумент, два или даже три. Если указан только один, то она возвращает случайное число от 0 до указанного аргумента. Причем сам аргумент в диапазон не входит. На языке математики – это [0; *a*).

```
>>> random.randrange(10)
4
```

Или:

```
>>> randrange(5)
0
```

Если в `randrange()` передается два аргумента, то она работает аналогично `randint()` за одним исключением. Верхняя граница не входит в диапазон, т. е. [*a*; *b*).

```
>>> random.randrange(5, 10)
9
>>> random.randrange(1, 2)
1
```

Здесь результатом второго вызова всегда будет число 1.

Если в `randrange()` передается три аргумента, то первые два – это границы диапазона, как в случае с двумя аргументами, а третий – так называемый шаг. Если, например, функция вызывается как `randrange(10, 20, 3)`, то "случайное" число будет выбираться из чисел 10, 13, 16, 19:

```
>>> random.randrange(10, 20, 3)
13
>>> random.randrange(10, 20, 3)
19
>>> random.randrange(10, 20, 3)
10
```

Функция random() – "случайные" вещественные числа

Чтобы получить случайное вещественное число, или, как говорят, число с плавающей точкой, следует использовать функцию `random()` из одноименного модуля **`random`** языка Python. Она не принимает никаких аргументов и возвращает число от 0 до 1, не включая 1:

```
>>> random.random()
0.17855729241927576
>>> random.random()
0.3310978930421846
```

или

```
>>> random()
0.025328854415995194
```

Результат содержит много знаков после запятой. Чтобы его округлить, можно воспользоваться встроенной в Python функцией `round()`:

```
>>> a = random.random()
>>> a
0.8366142721623201
>>> round(a, 2)
0.84
>>> round(random.random(), 3)
0.629
```

Чтобы получать случайные вещественные числа в иных пределах, отличных от [0; 1), прибегают к математическим приемам. Так если умножить полученное из `random()` число на любое целое, то получится вещественное в диапазоне от 0 до этого целого, не включая его:

```
>>> random.random() * 10
2.510618091637596
>>> random.random() * 10
6.977540211221759
```

Если нижняя граница должна быть отличной от нуля, то число из `random()` надо умножать на разницу между верхней и нижней границами, после чего прибавить нижнюю:

```
>>> random.random() * (10 - 4) + 4
9.517280589233597
>>> random.random() * (10 - 4) + 4
6.4429124181215975
>>> random.random() * (10 - 4) + 4
4.9231983600782385
```

В данном примере число умножается на 6. В результате получается число от 0 до 6. Прибавив 4, получаем число от 4 до 10.

Пример получения случайных чисел от -1 до 1:

```
>>> random.random() * (1 + 1) - 1
-0.673382618351051
>>> random.random() * (1 + 1) - 1
0.34121487148075924
>>> random.random() * (1 + 1) - 1
-0.988751324713907
>>> random.random() * (1 + 1) - 1
0.44137358363477674
```

Нижняя граница равна -1. При вычитании получается +. Когда же добавляется нижняя граница, то плюс заменяется на минус.

Для получения псевдослучайных чисел можно пользоваться исключительно функцией `random()`. Если требуется получить целое, то всегда можно округлить до него с помощью `round()` или отбросить дробную часть с помощью `int()`:

```
>>> int(random.random() * 100)
61
>>> round(random.random() * 100 - 50)
-33
```

Практическая работа

1. Используя функцию `randrange()` получите псевдослучайное четное число в пределах от 6 до 12. Также получите число кратное пяти в пределах от 5 до 100.

2. Напишите программу, которая запрашивает у пользователя границы диапазона и какое (целое или вещественное) число он хочет получить. Выводит на экран подходящее случайное число.

Урок 18. Списки

Список в Python – это встроенный тип (класс) данных, представляющий собой одну из разновидностей структур данных. Структуру данных можно представить как сложную единицу, объединяющую в себе группу более простых. Каждая разновидность структур данных имеет свои особенности. Список – это изменяемая последовательность произвольных элементов.

В большинстве других языков программирования есть такой тип данных как массив. В Питоне такого встроенного типа нет. Однако при знакомстве с программированием списки условно можно считать аналогом массивов за одним исключением. Составляющие массив элементы должны принадлежать одному типу данных, для списков такого ограничения нет.

Например, массив может содержать только целые числа или только вещественные числа или только строки. Список также может содержать элементы исключительно одного типа, что делает его похожим на массив. Но вполне допустимо, чтобы в одном списке содержались как числа, так и строки или иные типы данных.

Создавать списки можно разными способами. Создадим его простым перечисление элементов:

```
>>> a = [12, 3.85, 'black', -4]
>>> a
[12, 3.85, 'black', -4]
```

Итак, у нас имеется список, присвоенный переменной *a*. В Python список определяется квадратными скобками. Он содержит четыре элемента. Если где-то в программе нам понадобится весь этот список, мы получим доступ к нему, указав всего лишь одну переменную – *a*.

Элементы в списке упорядочены, то есть имеет значение, в каком порядке они расположены. Каждому элементу соответствует свой индекс, или номер. Индексация начинается с нуля. В данном случае число 12 имеет индекс 0, строка "black" – индекс 2. Чтобы извлечь определенный элемент, надо после имени переменной указать в квадратных скобках его индекс:

```
>>> a = [12, 3.85, 'black', -4]
>>> a[0]
12
>>> a[3]
-4
```

В Python существует также индексация с конца. Она начинается с -1:

```
>>> a[-1]
-4
>>> a[-2]
'black'
>>> a[-3], a[-4]
(3.85, 12)
```

Часто требуется извлечь не один элемент, а так называемый срез – часть списка. В этом случае указывается индекс первого элемента среза и индекс следующего за последним элементом среза:

```
>>> a[0:2]
[12, 3.85]
```

В данном случае извлекаются первые два элемента с индексами 0 и 1. Элемент с индексом 2 в срез уже не входит. В таком случае возникает вопрос, как извлечь срез, включающий в себя последний элемент? Если какой-либо индекс не указан, то считается, что имеется в виду начало или конец:

```
>>> a = [12, 3.85, 'black', -4]
>>> a[:3]
[12, 3.85, 'black']
>>> a[2:]
['black', -4]
>>> a[:]
[12, 3.85, 'black', -4]
```

Списки – изменяемые объекты. Это значит, что в них можно добавлять элементы, удалять их, изменять существующие. Проще всего изменить значение элемента. Для этого надо обратиться к нему по индексу и перезаписать значение в заданной позиции:

```
>>> a[1] = 4
>>> a
[12, 4, 'black', -4]
```

Добавлять и удалять лучше с помощью специальных встроенных методов списка:

```
>>> a.append('wood')
>>> a
[12, 4, 'black', -4, 'wood']
>>> a.insert(1, 'circle')
>>> a
[12, 'circle', 4, 'black', -4, 'wood']
>>> a.remove(4)
>>> a
```

```
[12, 'circle', 'black', -4, 'wood']
>>> a.pop()
'wood'
>>> a
[12, 'circle', 'black', -4]
>>> a.pop(2)
'black'
>>> a
[12, 'circle', -4]
```

Перечень всех методов списка можно узнать с помощью встроенной в Python функции `dir()`, передав в качестве аргумента переменную, связанную со списком, или название класса (в данном случае – **`list`**). В полученном из `dir()` списке надо смотреть имена без двойных подчеркиваний.

Для получения информации о конкретном методе следует воспользоваться встроенной функцией `help()`, передав ей в качестве аргумента имя метода, связанное с объектом или классом. Например, `help(a.pop)` или `help(list.index)`. Выход из справки – `q`.

Можно изменять списки не используя методы, а с помощью взятия и объединения срезов:

```
>>> b = [1, 2, 3, 4, 5, 6]
>>> b = b[:2] + b[3:]
>>> b
[1, 2, 4, 5, 6]
```

Здесь берется срез из первых двух элементов и срез, начиная с четвертого элемента (индекс 3) и до конца. После чего срезы объединяются с помощью оператора "сложения".

Можно изменить не один элемент, а целый срез:

```
>>> mylist = ['ab','ra','ka','da','bra']
>>> mylist[0:2] = [10, 20]
>>> mylist
[10, 20, 'ka', 'da', 'bra']
```

Пример создания пустого списка с последующим заполнением его в цикле случайными числами:

```
>>> import random
>>> c = []
>>> i = 0
>>> while i < 10:
...     c.append(random.randint(0, 100))
...     i += 1
...
```

```
>>> c
[30, 44, 35, 77, 53, 44, 49, 17, 61, 82]
```

Практическая работа

1. Напишите программу, которая запрашивает с ввода восемь чисел, добавляет их в список. На экран выводит их сумму, максимальное и минимальное из них. Для нахождения суммы, максимума и минимума воспользуйтесь встроенными в Python функциями `sum()`, `max()` и `min()`.
2. Напишите программу, которая генерирует сто случайных вещественных чисел и заполняет ими список. Выводит получившийся список на экран по десять элементов в ряд. Далее сортирует список с помощью метода `sort()` и снова выводит его на экран по десять элементов в строке. Для вывода списка напишите отдельную функцию, в качестве аргумента она должна принимать список.

Урок 19. Цикл for

Цикл **`for`** в языке программирования Python предназначен для перебора элементов структур данных и других составных объектов. Это не цикл со счетчиком, каковым является **`for`** во многих других языках.

Что значит перебор элементов? Например, у нас есть список, состоящий из ряда элементов. Сначала берем из него первый элемент, затем второй, потом третий и так далее. С каждым элементом мы выполняем одни и те же действия в теле **`for`**. Нам не надо извлекать элементы по их индексам, заботиться, на каком из них список заканчивается, и следующая итерация бессмысленна. Цикл **`for`** сам переберет и определит конец.

```
>>> numbers = [10, 40, 20, 30]
>>> for item in numbers:
...     print(item + 2)
...
12
42
22
32
```

После ключевого слова **`for`** используется переменная под именем *item*. Имя здесь может быть любым. Нередко используют *i*. На каждой итерации цикла **`for`** ей будет присвоен очередной элемент из списка *numbers*. Так при первой прокрутке цикла идентификатор *item* связан с

числом 10, на второй – с числом 40, и так далее. Когда элементы в *numbers* заканчиваются, цикл `for` завершает свою работу.

С английского "for" переводится как "для", "in" как "в". Перевести конструкцию с языка программирования на человеческий можно так: **для** каждого элемента **в** списке делать следующее (то, что в теле цикла).

В примере мы увеличивали каждый элемент на 2 и выводили его на экран. При этом сам список конечно же не изменялся:

```
>>> numbers
[10, 40, 20, 30]
```

Нигде не шла речь о перезаписи его элементов, они просто извлекались и использовались. Однако бывает необходимо изменить сам список, например, изменить значение каждого элемента в нем или только определенных, удовлетворяющих определенному условию. И тут без переменной, обозначающей индекс элемента, не обойтись:

```
>>> i = 0
>>> for item in numbers:
...     numbers[i] = item + 2
...     i += 1
...
>>> numbers
[12, 42, 22, 32]
```

Но если мы вынуждены использовать счетчик, то выгода от использования цикла `for` не очевидна. Если знать длину списка, то почему бы не воспользоваться `while`. Длину можно измерить с помощью встроенной в Python функции `len()`.

```
>>> i = 0
>>> while i < len(numbers):
...     numbers[i] = numbers[i] + 2
...     i += 1
...
>>> numbers
[14, 44, 24, 34]
```

Кроме того, с циклом `while` мы избавились от переменной *item*.

Функция range()

Теперь пришло время познакомиться со встроенной в Python функцией `range()`. "Range" переводится как "диапазон". Она может принимать один, два или три аргумента. Их назначение такое же как у функции `randrange()` из модуля `random`. Если задан только один, то

генерируются числа от 0 до указанного числа, не включая его. Если заданы два, то числа генерируются от первого до второго, не включая его. Если заданы три, то третье число – это шаг.

Однако, в отличие от `randrange()`, функция `range()` генерирует не одно случайное число в указанном диапазоне. Она вообще не генерирует случайные числа. Она создает последовательность чисел в указанном диапазоне. Так, `range(5, 11)` сгенерирует последовательность 5, 6, 7, 8, 9, 10. Однако это будет не список. Функция `range()` производит объекты своего класса – диапазоны:

```
>>> a = range(-10, 10)
>>> a
range(-10, 10)
>>> type(a)
<class 'range'>
```

Несмотря на то, что мы не видим последовательности чисел, она есть, и мы можем обращаться к ее элементам:

```
>>> a[0]
-10
>>> a[5]
-5
>>> a[15]
5
>>> a[-1]
9
```

Хотя изменять их нельзя, так как, в отличие от списков, объекты `range()` относятся к группе неизменяемых:

```
>>> a[10] = 100
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'range' object does not support item assignment
```

Цикл for и range()

Итак, зачем нам понадобилась функций `range()` в теме про цикл **`for`**? Дело в том, что вместе они образуют неплохой тандем. **`For`** как цикл перебора элементов, в отличие от **`while`**, позволяет не следить за тем, достигнут ли конец структуры. Не надо вводить счетчик для этого, изменять его и проверять условие в заголовке. С другой стороны, `range()` дает последовательность целых чисел, которые можно использовать как индексы для элементов того же списка.

```
>>> range(len(numbers))
range(0, 4)
```

Здесь с помощью функции `len()` измеряется длина списка. В данном случае она равна четырем. После этого число 4 передается в функцию `range()`, и она генерирует последовательность чисел от 0 до 3 включительно. Это как раз индексы элементов нашего списка.

Теперь "соединим" **`for`** и `range()`:

```
>>> numbers = [14, 44, 24, 34]
>>> for i in range(len(numbers)):
...     numbers[i] += 2
...
>>> numbers
[16, 46, 26, 36]
```

Еще раз обратим внимание, в заголовке цикла **`for`** берутся элементы вовсе не списка, а объекта **`range`**.

Практическая работа

1. Заполните список случайными числами. Используйте в коде цикл **`for`**, функции `range()` и `randint()`.
2. Если объект **`range`** (диапазон) передать встроенной в Python функции `list()`, то она преобразует его к списку. Создайте таким образом список с элементами от 0 до 100 и шагом 17.
3. В заданном списке, состоящем из положительных и отрицательных чисел, посчитайте количество отрицательных элементов. Выведите результат на экран.
4. Напишите программу, которая заполняет список пятью словами, введенными с клавиатуры, измеряет длину каждого слова и добавляет полученное значение в другой список. Например, список слов – ['yes', 'no', 'maybe', 'ok', 'what'], список длин – [3, 2, 5, 2, 4]. Оба списка должны выводиться на экран.

Урок 20. Функция enumerate

В Python есть еще одна встроенная функция, которая часто используется в заголовке **`for`**. Это функция `enumerate()`. Если `range()` позволяет получить только индексы элементов списка, то `enumerate()` – сразу индекс элемента и его значение.

Функция `enumerate()` применяется для так называемых итерируемых объектов (список относится к таковым) и создает объект-генератор, который генерирует кортежи, состоящие из двух элементов – индекса элемента и самого элемента.

```
>>> numbers = [16, 46, 26, 36]
>>> for i in enumerate(numbers):
...     print(i)
...
(0, 16)
(1, 46)
(2, 26)
(3, 36)
```

```
>>> b = "hello"
>>> for i in enumerate(b):
...     print(i)
...
(0, 'h')
(1, 'e')
(2, 'l')
(3, 'l')
(4, 'o')
```

Эти кортежи можно распаковывать, то есть извлекать индекс и значение, в теле цикла:

```
>>> for item in enumerate(numbers):
...     print(item[0], item[1])
...
0 16
1 46
2 26
3 36
```

Однако чаще это делают еще в заголовке `for`, используя две переменные перед `in`:

```
>>> for i, v in enumerate(numbers):
...     print(i, v)
...
0 16
1 46
2 26
3 36
```

Функция `enumerate` используется для упрощения прохода по коллекциям в цикле, когда кроме самих элементов требуется их индекс:

```
>>> a = [10, 20, 30, 40]
>>> for index, item in enumerate(a):
...     a[index] = item + 5
...
>>> a
[15, 25, 35, 45]
```

В данном случае на каждой итерации цикла из объекта, полученного от вызова функции `enumerate`, извлекается очередной кортеж. Этот кортеж состоит из индекса очередного элемента списка и значения этого элемента. Элементы кортежа связываются с идентификаторами *index* и *item*.

Без использования `enumerate` в цикл пришлось бы вводить переменную-счетчик:

```
>>> a = [10, 20, 30, 40]
>>> i = 0 # используется счетчик
>>> for num in a:
...     a[i] = num + 5
...     i += 1
...
>>> a
[15, 25, 35, 45]
```

Или извлекать элементы по индексу:

```
>>> a = [10, 20, 30, 40]
>>> for i in range(len(a)): # перебор по индексам
...     a[i] += 5
...
>>> a
[15, 25, 35, 45]
```

Другими словами, без функции `enumerate` можно обойтись. Однако в ряде случаев она может быть удобней.

Если `enumerate()` так хороша, зачем использовать `range()` в заголовке `for`? На самом деле незачем, если только вам так не проще. Кроме того, бывают ситуации, когда значения не нужны, нужны только индексы. Однако следует учитывать один нюанс. Функция `enumerate()` возвращает так называемый **объект-итератор**. Когда такие объекты сгенерировали значения, то становятся "пустыми". Второй раз по ним пройтись нельзя.

Функция `range()` возвращает **итерируемый объект**. Хотя такой объект может быть превращен в объект-итератор, сам по себе таковым не является.

Когда `range()` и `enumerate()` используются в заголовке `for`, то разницы нет, так как range- и enumerate-объекты не присваиваются переменным и после завершения работы цикла теряются.

Но если мы присваиваем эти объекты переменным, их отличия могут сказаться на выполнении программы:

```
>>> nums = [34, 15, 124, 800, 339]
>>>
>>> r_obj = range(len(nums))
>>> e_obj = enumerate(nums)
>>>
>>> for i in r_obj:
...     print(nums[i])
...     if i == 1:
...             break
...
34
15
>>> for i, v in e_obj:
...     print(v)
...     if i == 1:
...             break
...
34
15
>>> for i in r_obj:
...     print(nums[i])
...
34
15
124
800
339
>>> for i, v in e_obj:
...     print(v)
...
124
800
339
```

Сначала мы прерываем извлечение элементов из объектов на элементе с индексом 1. Когда снова прогоняем объекты через цикл `for`, то в случае *r_obj* обход начинается сначала, а в случае *e_obj* продолжается после места останова. Объект *e_obj* уже не содержит извлеченных ранее элементов.

Практическая работа

Дан список чисел. Используя функцию `enumerate()` в заголовке цикла `for`, создайте второй список, в котором каждый элемент должен быть строкой, включающей через пробел индекс и значение соответствующего элемента первого списка.

Урок 21. Строки

Мы уже рассматривали строки как простой тип данных наряду с целыми и вещественными числами и знаем, что строка – это последовательность символов, заключенных в одинарные или двойные кавычки.

В Python нет символьного типа – типа данных, объектами которого являются одиночные символы. Однако язык позволяет рассматривать строки как объекты, состоящие из подстрок длинной в один и более символов. При этом, в отличие от списков, строки не принято относить к структурам данных. Видимо потому, что структуры данных состоят из более простых типов данных, а для строк в Python нет более простого (символьного) типа.

С другой стороны, строка, как и список, – это упорядоченная последовательность элементов. Следовательно, из нее можно извлекать отдельные символы и срезы.

```
>>> s = "Hello, World!"
>>> s[0]
'H'
>>> s[7:]
'World!'
>>> s[::2]
'Hlo ol!'
```

В последнем случае извлечение идет с шагом, равным двум, то есть извлекается каждый второй символ. Извлекать срезы с шагом также можно из списков.

Важным отличием от списков является неизменяемость строк в Python. Нельзя перезаписать какой-то отдельный символ или срез в строке:

```
>>> s[-1] = '.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Интерпретатор сообщает, что объект типа `str` не поддерживает присвоение элементам.

Если требуется изменить строку, то можно создать новую из срезов старой:

```
>>> s = s[0:-1] + '.'
>>> s
'Hello, World.'
```

В примере берется срез из исходной строки, соединяется с другой строкой. Получается новая строка, которая присваивается переменной *s*. Ее старое значение при этом теряется.

Методы строк

В Python для строк есть множество методов. Посмотреть их можно по команде `dir(str)`, получить информацию по каждому – `help(str.имя_метода)`. Рассмотрим наиболее интересные из них.

Методы split и join

Метод **`split`** позволяет разбить строку по пробелам. В результате получается список слов. Если пользователь вводит в одной строке ряд слов или чисел, каждое из которых должно в программе обрабатываться отдельно, то без **`split`** не обойтись.

```
>>> s = input()
red blue orange white
>>> s
'red blue orange white'
>>> sl = s.split()
>>> sl
['red', 'blue', 'orange', 'white']
>>> s
'red blue orange white'
```

Список, возвращенный методом `split()`, мы могли бы присвоить той же переменной *s*, то есть `s = s.split()`. Тогда исходная строка была бы потеряна. Если она не нужна, то лучше не вводить дополнительную переменную.

Метод `split()` может принимать необязательный аргумент-строку, указывающей по какому символу или подстроке следует выполнить разделение:

```
>>> s.split('e')
['r', 'd blu', ' orang', ' whit', '']
>>> '40030023'.split('00')
['4', '3', '23']
```

Метод строк **`join`** выполняет обратное действие. Он формирует из списка строку. Поскольку это метод строки, то впереди ставится строка-разделитель, а в скобках – передается список:

```
>>> '-'.join(sl)
'red-blue-orange-white'
```

Если разделитель не нужен, то метод применяется к пустой строке:

```
>>> ''.join(sl)
'redblueorangewhite'
```

Методы find и replace

Данные методы строк работают с подстроками. Методы **`find`** ищет подстроку в строке и возвращает индекс первого элемента найденной подстроки. Если подстрока не найдена, то возвращает -1.

```
>>> s
'red blue orange white'
>>> s.find('blue')
4
>>> s.find('green')
-1
```

Поиск может производиться не во всей строке, а лишь на каком-то ее отрезке. В этом случае указывается первый и последний индексы отрезка. Если последний не указан, то ищется до конца строки:

```
>>> letters = 'ABCDACFDA'
>>> letters.find('A', 3)
4
>>> letters.find('DA', 0, 6)
3
```

Здесь мы ищем с третьего индекса и до конца, а также с первого и до шестого. Обратите внимания, что метод `find()` возвращает только первое вхождение. Так выражение `letters.find('A', 3)` последнюю букву 'A' не находит, так как 'A' ему уже встретилась под индексом 4.

Метод **`replace`** заменяет одну подстроку на другую:

```
>>> letters.replace('DA', 'NET')
'ABCNETCFNET'
```

Исходная строка, конечно, не меняется:

```
>>> letters
'ABCDACFDA'
```

Так что если результат надо сохранить, то его надо присвоить переменной:

```
>>> new_letters = letters.replace('DA', 'NET')
>>> new_letters
'ABCNETCFNET'
```

Метод format

Строковый метод **`format`** уже упоминался при рассмотрении вывода на экран с помощью функции `print()`:

```
>>> print("This is a {0}. It's {1}.".format("ball", "red"))
This is a ball. It's red.
```

Однако к **`print`** он никакого отношения не имеет, а применяется к строкам. Лишь потом заново сформированная строка передается в функцию вывода.

Возможности **`format`** широкие, рассмотрим основные.

```
>>> size1 = "length - {}, width - {}, height - {}"
>>> size1.format(3, 6, 2.3)
'length - 3, width - 6, height - 2.3'
```

Если фигурные скобки исходной строки пусты, то подстановка аргументов идет согласно порядку их следования. Если в фигурных скобках строки указаны индексы аргументов, порядок подстановки может быть изменен:

```
>>> size2 = "height - {2}, length - {0}, width - {1}"
>>> size2.format(3, 6, 2.3)
'height - 2.3, length - 3, width - 6'
```

Кроме того, аргументы могут передаваться по слову-ключу:

```
>>> info = "This is a {subj}. It's {prop}."
>>> info.format(subj="table", prop="small")
"This is a table. It's small."
```

Пример форматирования вещественных чисел:

```
>>> "{1:.2f} {0:.3f}".format(3.33333, 10/6)
'1.67 3.333'
```

Практическая работа

1. Вводится строка, включающая строчные и прописные буквы. Требуется вывести ту же строку в одном регистре, который зависит от того, каких букв больше. При равном

количестве преобразовать в нижний регистр. Например, вводится строка "HeLLo World", она должна быть преобразована в "hello world", потому что в исходной строке малых букв больше. В коде используйте цикл `for`, строковые методы `upper()` (преобразование к верхнему регистру) и `lower()` (преобразование к нижнему регистру), а также методы `isupper()` и `islower()`, проверяющие регистр строки или символа.

2. Строковый метод `isdigit()` проверяет, состоит ли строка только из цифр. Напишите программу, которая запрашивает с ввода два целых числа и выводит их сумму. В случае некорректного ввода программа не должна завершаться с ошибкой, а должна продолжать запрашивать числа. Обработчик исключений try-except использовать нельзя.

Урок 22. Кортежи

Кортежи (tuple) в Python – это те же списки за одним исключением. Кортежи неизменяемые структуры данных. Так же как списки они могут состоять из элементов разных типов, перечисленных через запятую. Кортежи заключаются в круглые, а не квадратные скобки.

```
>>> a = (10, 2.13, "square", 89, 'C')
>>> a
(10, 2.13, 'square', 89, 'C')
```

Из кортежа можно извлекать элементы и брать срезы:

```
>>> a[3]
89
>>> a[1:3]
(2.13, 'square')
```

Однако изменять его элементы нельзя:

```
>>> a[0] = 11
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Также у типа `tuple` нет методов для добавления и удаления элементов.

Возникает резонный вопрос. Зачем в язык программирования был введен этот тип данных, по-сути представляющий собой неизменяемый список? Дело в том, что иногда надо защитить список от изменений. Преобразовать же кортеж в список, если это потребуется, как и выполнить обратную операцию легко с помощью встроенных в Python функций `list()` и `tuple()`:

```
>>> a = (10, 2.13, "square", 89, 'C')
>>> b = [1, 2, 3]
>>> c = list(a)
>>> d = tuple(b)
>>> c
[10, 2.13, 'square', 89, 'C']
>>> d
(1, 2, 3)
```

Рассмотрим случай, когда уместно использовать кортежи. В Python изменяемые объекты передаются в функцию по ссылке. Это значит, что не создается копия объекта, а переменной-параметру присваивается ссылка на уже существующий объект. В итоге, если в теле функции объект изменяется, то эти изменения касаются глобального объекта.

```
def add_num(seq, num):
    for i in range(len(seq)):
        seq[i] += num
    return seq

origin = [3, 6, 2, 6]
changed = add_num(origin, 3)

print(origin)
print(changed)
```

Данная программа неправильная. Хотя никаких выбросов исключений не произойдет, она содержит логическую ошибку. На выводе получаем:

```
[6, 9, 5, 9]
[6, 9, 5, 9]
```

То есть исходный список был также изменен. Параметр *seq* содержал ссылку не на свой локальный список, а на список-оригинал. Таким образом, в операторе `return` здесь нет смыла. Если планировалось, что функция будет изменять глобальный список, то программа должна была бы выглядеть так:

```
def add_num(seq, num):
    for i in range(len(seq)):
        seq[i] += num

origin = [3, 6, 2, 6]
add_num(origin, 3)
print(origin)
```

Что делать, если все же требуется не изменять исходный список, а сформировать по нему новый. Задачу можно решить несколькими способами. Во первых, в функции можно создать локальный список, после чего возвращать его:

```
def add_num(seq, num):
    new_seq = []
    for i in seq:
        new_seq.append(i + num)
    return new_seq

origin = [3, 6, 2, 6]
changed = add_num(origin, 3)

print(origin)
print(changed)
```

Результат:

```
[3, 6, 2, 6]
[6, 9, 5, 9]
```

Исходный список в функции не меняется. Его элементы лишь перебираются.

Второй способ защитить список-оригинал – использовать кортеж. Этот способ более надежный, так как в больших программах трудно отследить, что ни одна функция не содержит команд изменения глобальных данных.

Хотя преобразовывать к кортежу можно как при передаче в функцию, так и в самой функции, лучше сразу делать глобальный список кортежем. Поскольку неизменяемые объекты передаются по значению, а не по ссылке, то в функцию будет поступать копия структуры, а не оригинал. Даже если туда передается оригинал, изменить его невозможно. Можно лишь, как вариант, скопировать его и/или изменить тип, создав тем самым локальную структуру, и делать с ней все, что заблагорассудится.

```
def add_num(seq, num):
    seq = list(seq)
    for i in range(len(seq)):
        seq[i] += num
    return seq

origin = (3, 6, 2, 6)
changed = add_num(origin, 3)

print(origin)
print(changed)
```

Списки в кортежах

Кортежи могут содержать списки, также как списки быть вложенными в другие списки.

```
>>> nested = (1, "do", ["param", 10, 20])
```

Как вы думаете, можем ли мы изменить список `["param", 10, 20]` вложенный в кортеж *nested*? Список изменяем, кортеж – нет. Если вам кажется, что нельзя, то вам кажется неправильно. На самом деле можно:

```
>>> nested[2][1] = 15
>>> nested
(1, 'do', ['param', 15, 20])
```

Примечание. Выражения типа `nested[2][1]` используются для обращения к вложенным объектам. Первый индекс указывает на позицию вложенного объекта, второй – индекс элемента внутри вложенного объекта. Так в данном случае сам список внутри кортежа имеет индекс 2, а элемент списка 10 – индекс 1 в списке.

Странная ситуация. Кортеж неизменяем, но мы все-таки можем изменить его. На самом деле кортеж остается неизменяемым. Просто в нем содержится не сам список, а ссылка на него. Ее изменить нельзя. Но менять сам список можно.

Чтобы было проще понять, перепишем кортеж так:

```
>>> l = ["param", 10, 20]
>>> t = (1, "do", l)
>>> t
(1, 'do', ['param', 10, 20])
```

Кортеж содержит переменную-ссылку. Поменять ее на другую ссылку нельзя. Но кортеж не содержит самого списка. Поэтому его можно менять как угодно:

```
>>> l.pop(0)
'param'
>>> t
(1, 'do', [10, 20])
```

Однако такой номер не проходит с неизменяемыми типами:

```
>>> a = "Kat"
>>> t = (a, l)
>>> t
('Kat', [10, 20])
>>> a = "Bat"
>>> t
('Kat', [10, 20])
```

Они передаются в кортеж, как и в функцию – по значению. То есть их значение копируется в момент передачи.

Практическая работа

1. Чтобы избежать изменения исходного списка, не обязательно использовать кортеж. Можно создать его копию с помощью метода списка `copy()` или взять срез от начала до конца `[:]`. Скопируйте список первым и вторым способом и убедитесь, что изменение копий никак не отражается на оригинале.
2. Заполните один кортеж десятью случайными целыми числами от 0 до 5 включительно. Также заполните второй кортеж числами от -5 до 0. Для заполнения кортежей числами напишите одну функцию. Объедините два кортежа с помощью оператора `+`, создав тем самым третий кортеж. С помощью метода кортежа `count()` определите в нем количество нулей. Выведите на экран третий кортеж и количество нулей в нем.

Урок 23. Словари

В языке программирования Python словари (тип `dict`) представляют собой еще одну разновидность структур данных наряду со списками и кортежами. *Словарь – это **изменяемый** (как список) **неупорядоченный** (в отличие от строк, списков и кортежей) набор элементов "ключ:значение"*. Такие элементы-пары будем называть записями.

"Неупорядоченный" – значит, что последовательность расположения пар не важна, вследствие чего обращение к элементам по индексам невозможно.

В других языках структуры, схожие со словарями, называются по-другому. Например, в Java подобный тип данных называется отображением.

Чтобы представление о словаре стало более понятным, проведем аналогию с обычным словарем, например, англо-русским. На каждое английское слово в таком словаре есть русское слово-перевод: cat – кошка, dog – собака, table – стол и т. д. Если англо-русский словарь описать с помощью Python, то английские слова можно сделать ключами, а русские – их значениями:

```
a = {'cat': 'кошка', 'dog': 'собака', 'bird': 'птица'}
```

Обратите внимание, что для определения словаря используются **фигурные скобки**. Синтаксис словаря на Питоне описывается такой схемой:

В словаре не может быть двух элементов с одинаковыми ключами. Однако могут быть одинаковые значения у разных ключей. Например, такой вариант словаря корректен:

```
>>> a = {'cat': 'кошка', 'dog': 'собака', 'bird': 'птица', 'fowl': 'птица'}
>>> a
{'cat': 'кошка', 'dog': 'собака', 'bird': 'птица', 'fowl': 'птица'}
```

Мы получили то, что определили. В случае повторения ключей ошибки тоже не будет:

```
>>> b = {'cat': 'кошка', 'dog': 'собака', 'bird': 'птица', 'dog': 'пёс'}
>>> b
{'cat': 'кошка', 'dog': 'пёс', 'bird': 'птица'}
```

Однако добавление записи с ключом, который уже был в словаре, приводит к обновлению его значения.

В нашем примере, если мы хотим решить проблему перевода одного английского слова несколькими русскими, то вместо строкового значения можем использовать список или любой другой структурный тип данных:

```
>>> b = {'cat': 'кошка', 'dog': ['собака', 'пёс'], 'bird': 'птица'}
>>> b
{'cat': 'кошка', 'dog': ['собака', 'пёс'], 'bird': 'птица'}
```

С другой стороны, в качестве ключей нельзя использовать изменяемые типы данных. Это связано с особенностями хранения словаря в памяти компьютера.

```
>>> a = {'cat': 'кошка', 'dog': 'собака', ['bird', 'fowl']: 'птица'}
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

Однако возможно такое:

```
>>> a = {'cat': 'кошка', 'dog': 'собака', ('bird', 'fowl'): 'птица'}
>>> a
{'cat': 'кошка', 'dog': 'собака', ('bird', 'fowl'): 'птица'}
```

Потому что кортеж – это неизменяемый тип данных (и в данном случае не должен содержать элементов изменяемых типов внутри себя). Другое дело, что в большинстве случаев для удобства пользования словарем ключи не следует делать составными.

В словаре доступ к значениям осуществляется не по индексам, а **по ключам**, которые заключаются в квадратные скобки (по аналогии с индексами списков):

```
>>> a['cat']
'кошка'
```

```
>>> a['bird']
'птица'
```

Но если таким способом попытаться получить значение ключа, которого в словаре нет, то возникнет ошибка:

```
>>> a
{'cat': 'кошка', 'dog': 'собака', 'bird': 'птица'}
>>> a['pig']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'pig'
```

Исключение можно обработать с помощью инструкции try-except:

```
animals = {'cat': 'кошка', 'dog': 'собака', 'bird': 'птица'}

a_key = input('Введите слово на английском: ')

try:
    print('Его перевод:', animals[a_key])
except KeyError:
    print('В словаре нет такого слова')
```

С другой стороны, у словаря как типа данных есть метод `get()`, с помощью которого также можно получать значения ключей:

```
>>> a.get('cat')
'кошка'
>>> a.get('bird')
'птица'
```

Однако когда ключа в словаре нет, метод `get()` ведет себя не так, как это было в случае получения значения с помощью квадратных скобок. Вместо выброса исключения, метод возвращает объект `None`. Тогда в программе выше вместо конструкции try-except достаточно написать всего одну строчку кода:

```
print('Его перевод:', animals.get(a_key))
```

Пример выполнения при вводе отсутствующего ключа:

```
Введите слово на английском: pig
Его перевод: None
```

В метод `get()` можно передать второй аргумент, который при отсутствии ключа в словаре будет подставляться вместо `None`.

```
print('Его перевод:', animals.get(a_key, '-'))
```

Словари, как и списки, являются изменяемым типом данных: позволительно изменять, добавлять и удалять записи (пары "ключ:значение"). Изначально словарь можно создать пустым (например, `d = {}`) и потом заполнить его элементами.

```
>>> fruits = {}
>>> fruits['apple'] = 4
>>> fruits['kiwi'] = 3
>>> fruits
{'apple': 4, 'kiwi': 3}
```

Добавление и изменение имеет одинаковый синтаксис: `словарь[ключ] = значение`. Ключ может быть как уже существующим (тогда происходит изменение значения), так и новым (происходит добавление элемента словаря).

```
>>> fruits['orange'] = 2
>>> fruits['apple'] = 5
>>> fruits
{'apple': 5, 'kiwi': 3, 'orange': 2}
```

Представим, что мы хотим написать программу, которая запрашивает у пользователя ключ и значение. Если ключа нет в словаре, то происходит добавление новой записи в словарь. Если же полученный ключ там уже есть, надо добавить введенное значение к имеющемуся у данного ключа. То есть, перед тем, как выполнять ту или иную операцию, необходимо проверить, есть ли ключ в словаре. В Python это делается с помощью оператора **`in`**:

```
>>> 'apple' in fruits
True
>>> 'banana' in fruits
False
```

Тогда программа может выглядеть так:

```
fruits = {'apple': 5, 'kiwi': 3, 'orange': 2}

k = input('Fruit name: ')
v = int(input('Number: '))

if k in fruits:
    fruits[k] += v
else:
    fruits[k] = v

print(fruits)
```

Также как списки словари можно перебирать в цикле `for`:

```
>>> fruits = {'apple': 5, 'kiwi': 3, 'orange': 2}
>>> for k in fruits:
...     print(k)
...
apple
kiwi
orange
```

Здесь выражение `k in fruits` в заголовке цикла обозначает не то, что в заголовке `if`. Там мы проверяли один элемент на вхождение в структуру, а в цикле происходит перебор всех элементов с присваиванием каждого переменной, указанной перед `in`. Но и там, и здесь обращение к переменной, связанной со словарем, как бы извлекает из него только ключи (дает нам перечень ключей), но не значения. Конечно, по ключам всегда можно получить значения:

```
>>> fruits = {'apple': 5, 'kiwi': 3, 'orange': 2}
>>> for k in fruits:
...     print(fruits[k])
...
5
3
2
```

Однако в Python есть другие способы получения перечней ключей, значений, а также их пар – методы `keys()`, `values()` и `items()`. Со значениями все понятно. Они могут понадобиться отдельно от ключей:

```
>>> nums = {1: 'one', 2: 'two', 3: 'three', 4: 'four'}
>>> for v in nums.values():
...     print(v)
...
one
two
three
four
>>>
>>> fruits = {'apple': 5, 'kiwi': 3, 'orange': 2}
>>> sum(fruits.values())
10
```

Возникает вопрос: зачем нужны наборы ключей или пар, если и так ключи можно получить из имени словаря, а сам словарь и так состоит из пар? На самом деле ключи просто так получить нельзя. То, как используется переменная-словарь в контексте оператора `in`, не имеет отношения

к обычному обращению к переменной, когда возвращается ее значение. В данном случае это будет целый словарь:

```
>>> nums
{1: 'one', 2: 'two', 3: 'three', 4: 'four'}
>>> fruits
{'apple': 5, 'kiwi': 3, 'orange': 2}
```

Теперь посмотрим, что нам вернет метод `keys()`:

```
>>> nums.keys()
dict_keys([1, 2, 3, 4])
>>> fruits.keys()
dict_keys(['apple', 'kiwi', 'orange'])
```

Это перечень только ключей словаря, упакованный в особый тип объекта. В случае `items()` получаем подобный объект, но состоящий из кортежей `(ключ, значение)`.

```
>>> nums.items()
dict_items([(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')])
```

Объекты типов **`dict_items`**, **`dict_values`**, **`dict_keys`** могут перебираться с помощью цикла **`for`**.

```
>>> for i in nums.items():
...     print(i)
...
(1, 'one')
(2, 'two')
(3, 'three')
(4, 'four')
```

Обратите внимание, что на каждой итерации цикла из **`dict_items`** извлекается кортеж. Его можно распаковывать сразу в заголовке цикла **`for`**, отделяя таким образом ключ от значения.

```
>>> for k, v in nums.items():
...     print(k, 'is', v)
...
1 is one
2 is two
3 is three
4 is four
```

Удаление записи из словаря выполняется с помощью встроенного оператора **`del`** языка Python.

```
>>> e = {1.2: 5, 1.3: 10, 1.4: 8}
>>> del e[1.2]
>>> e
{1.3: 10, 1.4: 8}
```

Если указанного ключа нет, то возникнет исключение `KeyError`. В связи с этим интересен метод `pop()`, который также **удаляет элемент из словаря** и также возбуждает `KeyError`, если не передавать второй аргумент. Но если он указан, то в случае отсутствия ключа `pop()` вернет этот дефолтный аргумент. Если же ключ есть, вернется его значение.

```
colors = {'white': 'ffffff', 'black': '000000',
          'red': 'ff0000', 'green': '00ff00'}

while colors:
    u = input('Цвет: ')
    print(colors.pop(u, '-'))

print('Все цвета были использованы!')
```

Пример выполнения:

```
Цвет: white
ffffff
Цвет: black
000000
Цвет: green
00ff00
Цвет: dfdf
-
Цвет: red
ff0000
Все цвета были использованы!
```

Примечание: при преобразовании пустого словаря к булевому значению возвращается `False`. Поэтому цикл `while` завершается, когда из словаря исчезают все записи.

Почитать о всех методах словаря можно в этой статье, также о разных способах создания словарей.

Практическая работа

1. Создайте словарь, связав его с переменной *school*, и наполните данными, которые бы отражали количество учащихся в разных классах (1а, 1б, 2б, 6а, 7в и т. п.). Внесите изменения в словарь согласно следующему: а) в одном из классов изменилось количество

учащихся, б) в школе появился новый класс, с) в школе был расформирован (удален) другой класс. Вычислите общее количество учащихся в школе.

2. В Python ключи словаря можно получить, воспользовавшись встроенной функцией `list()`. Присвойте одной переменной произвольный словарь, второй – список его ключей, полученный из `list()`, третьей – результат выполнения словарного метода `keys()`. После этого внесите изменения в словарь, например, добавив в него еще одну запись. Выведите значения переменных на экран. Сделайте вывод об особенностях объектов типа `dict_keys`.

Урок 24. Функция open. Чтение и запись файлов

Большие объемы данных хранят не в списках или словарях, а в файлах и базах данных. В этом уроке изучим особенности работы с текстовыми файлами в Python. Такие файлы рассматриваются как содержащие символы и строки.

Бывают еще байтовые (бинарные) файлы, которые рассматриваются как потоки байтов. По байтам считываются, например, файлы изображений. Работа с бинарными файлами несколько сложнее. Нередко их обрабатывают с помощью специальных модулей Python (pickle, struct).

Функция open

Связь с файлом на жестком диске выполняется с помощью встроенной в Python функции `open()`. Обычно ей передают один или два аргумента. Первый – имя файла или имя с адресом, если файл находится не в том каталоге, где находится сама программа. Второй аргумент – режим, в котором открывается файл.

Обычно используются режимы чтения (`'r'`) и записи (`'w'`). Если файл открыт в режиме чтения, то запись в него невозможна. Можно только считывать данные. Если файл открыт в режиме записи, то в него можно только записывать данные, считывать нельзя.

Если файл открывается в режиме `'w'`, то все данные, которые в нем были до этого, стираются. Файл становится пустым. Если не надо удалять существующие в файле данные, тогда следует использовать вместо режима записи, режим дозаписи (`'a'`).

Если файл отсутствует, то открытие его в режиме `'w'` создаст новый файл. Бывают ситуации, когда надо гарантировано создать новый файл, избежав случайной перезаписи данных существующего. В этом случае вместо режима `'w'` используется режим `'x'`. В нем всегда создается новый файл для записи. Если указано имя существующего файла, то будет выброшено исключение. Потери данных в уже имеющемся файле не произойдет.

Если при вызове `open()` второй аргумент не указан, то файл открывается в режиме чтения как текстовый файл. Чтобы открыть файл как байтовый, дополнительно к букве режима

чтения/записи добавляется символ `'b'`. Буква `'t'` обозначает текстовый файл. Поскольку это тип файла по умолчанию, то обычно ее не указывают.

Нельзя указывать только тип файла, то есть `open("имя_файла", 'b')` есть ошибка, даже если файл открывается на чтение. Правильно – `open("имя_файла", 'rb')`. Только текстовые файлы мы можем открыть командой `open("имя_файла")`, потому что и 'r' и 't' подразумеваются по-умолчанию.

Функция `open()` возвращает объект файлового типа. Его надо либо сразу связать с переменной, чтобы не потерять, либо сразу прочитать.

Чтение файла

С помощью файлового метода `read()` можно прочитать файл целиком или только определенное количество байт. Пусть у нас имеется файл *data.txt* с таким содержимым:

```
one - 1 - I
two - 2 - II
three - 3 - III
four - 4 - IV
five - 5 - V
```

Откроем его и почитаем:

```
>>> f1 = open('data.txt')
>>> f1.read(10)
'one - 1 - '
>>> f1.read()
'I\ntwo - 2 - II\nthree - 3 - III\nfour - 4 - IV\nfive - 5 - V\n'
>>> f1.read()
''
>>> type(f1.read())
<class 'str'>
```

Сначала считываются первые десять символов. Последующий вызов `read()` считывает весь оставшийся текст. После этого объект файлового типа *f1* становится пустым.

Заметим, что метод `read()` возвращает строку.

Для того чтобы читать файл построчно, существует метод `readline()`:

```
>>> f1 = open('data.txt')
>>> f1.readline()
'one - 1 - I\n'
>>> f1.readline()
'two - 2 - II\n'
>>> f1.readline()
```

```
'three - 3 — III\n'
```

Метод `readlines()` считывает сразу все строки и создает список:

```
>>> f1 = open('data.txt')
>>> f1.readlines()
['one - 1 - I\n', 'two - 2 - II\n', 'three - 3 - III\n', 'four - 4 - IV\n', 'five
- 5 - V\n']
```

Объект файлового типа относится к итераторам. Из таких объектов происходит последовательное извлечение элементов. Элементами в данном случае являются строки-линии файла. Поэтому считывать данные из файла можно сразу в цикле без использования методов чтения:

```
>>> for i in open('data.txt'):
...     print(i)
...
one - 1 - I

two - 2 - II

three - 3 - III

four - 4 - IV

five - 5 - V

>>>
```

Здесь выводятся лишние пустые строки, потому что функция `print()` преобразует `'\n'` в переход на новую строку. К этому добавляет свой переход на новую строку. Создадим список строк файла без `'\n'`:

```
>>> nums = []
>>> for i in open('data.txt'):
...     nums.append(i[:-1])
...
>>> nums
['one - 1 - I', 'two - 2 - II', 'three - 3 - III', 'four - 4 - IV', 'five - 5 - V']
```

Переменной *i* присваивается очередная строка файла. Мы берем ее срез от начала до последнего символа, не включая его. Следует иметь в виду, что `'\n'` это один символ, а не два.

Запись в файл

Запись в файл выполняется с помощью методов `write()` и `writelines()`. Во второй можно передать структуру данных:

```
>>> l = ['three', 'four']
>>> f2 = open('newdata.txt', 'w')
>>> f2.write('one')
3
>>> f2.write(' two')
4
>>> f2.writelines(l)
```

Метод `write()` возвращает количество записанных символов.

Закрытие файла

После того как работа с файлом закончена, важно не забывать его закрыть, чтобы освободить место в памяти. Делается это с помощью файлового метода `close()`. Свойство файлового объекта `closed` позволяет проверить закрыт ли файл.

```
>>> f1.close()
>>> f1.closed
True
>>> f2.closed
False
```

Если файл открывается в заголовке цикла (`for i in open('fname')`), он автоматически не закрывается при завершении работы цикла, а остается открытым на неопределенное количество времени. Поэтому в больших программах лучше так не делать. Если файл открывается для одномоментного чтения или записи, и вы не хотите захламлять код строчками кода его открытия и закрытия, то следует использовать оператор **`with`**. Он отслеживает, чтобы при выходе из его тела файл был закрыт. Пример использования:

```
>>> with open('test.txt') as f:
...     for line in f:
...             print(line, end='')
...
one two
three
four
```

Практическая работа

1. Создайте файл *data.txt* по образцу урока. Напишите программу, которая открывает этот файл на чтение, построчно считывает из него данные и записывает строки в другой файл (*dataRu.txt*), заменяя английские числительные русскими, которые содержатся в списке (`["один", "два", "три", "четыре", "пять"]`), определенном до открытия файлов.
2. Создайте файл *nums.txt*, содержащий несколько чисел, записанных через пробел. Напишите программу, которая подсчитывает и выводит на экран общую сумму чисел, хранящихся в этом файле.

Итоги курса "Python. Введение в программирование"

В этом курсе мы изучили основы структурного программирования, а в качестве инструмента использовали современный и популярный язык Python. Также были изучены его некоторые специфические особенности, что неизбежно, так как любой язык программирования чем-то отличается от всех остальных. Однако базовые принципы зачастую остаются общими.

Подводя итоги вспомним и обобщим изученный материал.

В курсе были рассмотрены следующие встроенные типы данных:

1. `int` – целые числа
2. `float` – числа с плавающей точкой (дробные)
3. `str` – строки
4. `list` – списки
5. `tuple` – кортежи
6. `dict` – словари
7. файловые типы

В Python у модулей и функций также имеется свой тип данных. Хотя их скорее следует рассматривать как важный элемент структурного программирования.

Функции бывают встроенными в язык, из модулей стандартной библиотеки, из сторонних библиотек, а также пользовательскими, то есть определяемыми самим прикладным программистом. То же самое касается и классов (типов данных). Функции, определенные в классах, называются методами.

Функции позволяют разделить программу на логически завершенные части, каждая из которых выполняет свою подзадачу. Организация функций в модули делает их в своем роде мобильными, то есть дает возможность использовать их в других программах и другими людьми.

Существуют две ключевые управляющие конструкции – это ветвление и цикл. В Python ветвление реализуется условным оператором `if` и его расширенными версиями **if-else** и **if-elif-…-else**. В других языках, кроме `if`, встречается оператор **switch-case**. Его принято называть не столько условным оператором, сколько переключателем. В Python 3.10 появился оператор **match-case**, который для простых случаев можно считать аналогом **switch**.

В Python есть два типа циклов – `while` и `for`. В цикле `for` выполняется перебор элементов структур данных или последовательное извлечение элементов из объектов-итераторов. В Python нет вариации `for` как цикла со счетчиком. Хотя в языке программирования Java есть обе разновидности `for`. В ряде других языков цикл `for` – это исключительно цикл со счетчиком.

В программах могут возникать ошибки и исключительные ситуации, нарушающие нормальный ход выполнения. Программист должен уметь их предусмотреть и внедрить "код-перехватчик". В Python исключения обрабатываются с помощью оператора **try-except**, который также следует отнести к управляющим конструкциям. Следует отметить, что в крупных проектах тестированием программ занимаются отдельные люди, которых называют тестерами.

Если вы изучали данный курс с целью подготовки к экзамену по информатике, то следующий этап – это научиться решать задачи по программированию, изучить основные алгоритмы.

Если ваша цель – познакомиться с программированием глубже, возможно стать на путь профессионального разработчика, то следующий шаг – это объектно-ориентированное программирование (ООП). Данная парадигма прочно обосновалась и нашла широкое распространение.

В основе ООП лежат все те же типы данных, функции и управляющие конструкции, которые изучаются в рамках структурного программирования. Однако ООП идет дальше, объединяет данные и функции в классы, между объектами которых организуется взаимодействие. Здесь больше абстракции и подобия реальному миру.

Урок 25. Особенности работы операторов and и or

В уроке про логические операторы было сказано, что `and` и `or` вычисляют стоящее по правую сторону от них выражение, только если по левому еще не ясен результат всего выражения.

В случае `and`, если первый операнд возвращает ложь, то все выражение будет ложно, независимо от значения второго операнда, поэтому вычислять второй смысла нет.

В случае `or`, если первое выражение возвращает истину, то второе не имеет смысла вычислять, так как все сложное логическое выражение все равно будет истинным.

Такое сокращенное вычисление используется не только в Питоне, но и ряде других языков программирования. Однако в Python `and` и `or` можно использовать не только по отношению к логическим операндам, а вообще к любым типам. При этом тип результата может быть не булевый. И отсюда вытекает, что `and` и `or` не совсем логические операторы.

Классика жанра:

```
>>> a = 10
>>> b = 0
>>> a > b and b > 0
False
>>> a > b and b >= 0
True
```

По-сути здесь в обоих случаях нам возвращается результат вычисления второго операнда. Если бы первый (`a > b`) был ложным, мы бы до второго вообще не дошли. Python вернул бы нам результат вычисления первого выражения, который был бы всегда `False`. Это правило используется и в случае данных других типов.

```
>>> a = 10
>>> b = 5
>>> a and b
5
>>> b and a
10
>>> 0 and b
0
```

Если бы здесь `and` работал как нормальный логический оператор, мы бы в двух случаях получили `True` и в последнем `False`. Потому что и 5 и 10 – это истины, 0 – ложь.

Однако `and` возвращает нам сами данные и делает это по следующему правилу: если первый операнд истина, то вернуть значение второго, не важно какое оно. Если первый операнд ложь, то его и вернуть.

То есть `and` и `or` оценивают данные с точки зрения булевого типа, но возвращает сами данные. Булев тип лишь тогда является результатом операции, когда сами операнды дают булев тип. Причем можно комбинировать разные типы данных:

```
>>> [] and 'hi'
[]
>>> 'hi' and [1,2,3]
[1, 2, 3]
>>> 'hi' and False
False
>>> True and 'hi'
'hi'
```

В случае с `or` результат этих же выражений будет выглядеть так:

```
>>> [] or 'hi'
'hi'
>>> 'hi' or [1,2,3]
'hi'
>>> 'hi' or False
'hi'
>>> True or 'hi'
True
```

До второго выражения мы доходим только в случае, если первое ложно. И независимо от того, что там во втором (возможно тоже ложь), это второе возвращается:

```
>>> [] or ''
''
```

Практическая работа

1. Напишите одну строчку кода, в котором у пользователя запрашивается строка. Если он ничего не вводит и нажимает Enter, то соответствующей переменной присваивается вами заданная строка по-умолчанию.
2. Напишите программу, в которой проверяется, содержится ли введенное значение в списке. Если оно не содержится, происходит выход из программы. Проверку и выход запишите в одну строку.

Урок 26. Множества

Множества – это еще одна встроенная в Python структура данных, относящаяся к коллекциям наряду со списками, словарями и кортежами. Множество неупорядоченно как словарь, изменчиво как словарь и список, содержит просто элементы как список и кортеж, а не пары `ключ:значение` как словарь.

Особенностью множества является уникальность каждого ее элемента. Другими словами, в списке может быть два элемента со значением, скажем, 100, а во множестве такого быть не может.

Понятие множеств пришло из математики. Также представление о множествах используют в компьютерной графике. Существуют три базовые операции, которые выполняют над множествами. Это объединение, пересечение и разность.

Допустим, у вас есть две фигуры, каждая точка которых описывается координатами. Каждая фигура определяется своим множеством точек. При объединении этих фигур во множество попадут все точки обеих фигур, но если фигуры перекрывались, то точки области перекрытия в результирующем множестве будут представлены в единственном числе.

В случае пересечения в результирующее множество попадут только точки, которые есть и у первой фигуры и у второй. Другими словами, их область перекрытия.

При нахождении разности множеств имеет значение, что из чего вычитается. В итоговое множество попадают все точки "уменьшаемого", исключая те, которые являются общими с "вычитаемым".

В языке программирования Python изменяемое множество можно создать двумя способами. Во-первых, в фигурных скобках перечислить через запятую элементы. Во вторых, вызвать встроенную функцию `set()`.

```
>>> a = {1, 2, 3, 4, 1}
>>> a
{1, 2, 3, 4}
>>> b = [1, 2, 3]
>>> b = set(b)
>>> b
{1, 2, 3}
>>> c = set(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: set expected at most 1 arguments, got 3
>>> set('abc')
{'b', 'c', 'a'}
>>> set([1,2,4])
{1, 2, 4}
```

Обратите внимание, во множестве *a* уже нет повтора единицы. В `set()` можно передавать только итерируемый объект, а не перечислять сами элементы.

Для операций над множествами есть специальные методы, однако также можно использовать знаки, предназначенные в Python для побитовых операций над числами, и знак минус. В случае множеств эти операции обозначают следующее:

- `|` – объединение множеств,
- `&` – пересечение,
- `-` – разность,
- `^` – симметрическая разность.

```
>>> a = {'red', 'green', 'blue', 'white'}
>>> b = {'black', 'white', 'blue', 'yellow', 'brown'}
>>> a | b
{'green', 'black', 'blue', 'red', 'yellow', 'white', 'brown'}
>>> a & b
{'white', 'blue'}
>>> a - b
{'green', 'red'}
>>> b - a
{'black', 'yellow', 'brown'}
>>> a ^ b
{'green', 'yellow', 'brown', 'black', 'red'}
```

В случае симметрической разности исключаются совпадающие элементы.

Это же с помощью вызова самих методов:

```
>>> a.union(b)
{'green', 'black', 'blue', 'red', 'yellow', 'white', 'brown'}
>>> a.intersection(b)
{'white', 'blue'}
>>> a.difference(b)
{'green', 'red'}
>>> b.difference(a)
{'black', 'yellow', 'brown'}
>>> a.symmetric_difference(b)
{'green', 'yellow', 'brown', 'black', 'red'}
```

Все методы объекта типа `set` можно посмотреть командой `dir(set)`.

Множества, также как другие коллекции, поддерживают операцию `in` – проверку на вхождение элемента в структуру.

```
>>> 'black' in a
False
>>> 'black' in b
True
>>> 'black' not in b
False
```

Практическая работа

Множество как структура данных подходит для хранения списка тегов. Если пользователь вводит новый тег, он добавляется во множество. Если вводит тег, который уже есть во множестве, дублирования не произойдет.

Напишите программу, в которой пользователь вводит слова. Если слово до этого не вводилось, оно добавляется ко множеству. Если слово уже было введено ранее, то пользователь получает сообщение "уже есть".

Урок 27. Матрицы

В программировании матрицами обычно называют двумерные массивы. Двумерные – значит, каждый элемент массива сам представляет собой массив. Эти вложенные массивы обычно одномерные, иначе придется говорить о трехмерных или многомерных матрицах.

При этом все вложенные массивы должны иметь одинаковую друг с другом длину, потому что матрицу обычно представляют в виде правильного прямоугольника, то есть привычной двумерной таблицы, в которой все строки равны между собой по длине.

Размерность таблицы, измеряется количеством строк и столбцов. В переводе на язык матриц количество строк – это количество вложенных массивов, количество столбцов – это количество элементов в одном вложенном массиве.

Поскольку в Python нет встроенного в язык типа данных "массив", а есть во многом похожий "список", то и матрицы будем строить из списка. Пусть нам нужна матрица 3x4 из случайных чисел. Программа ее генерации и вывода на экран может выглядеть так:

```
from random import randint
N = 3
M = 4
a = []

for i in range(N):
    b = []
```

```
    for j in range(M):
        b.append(randint(1,99))
    a.append(b)

print(a, end='\n\n')

for i in range(N):
    for j in range(M):
        print("%3d" % a[i][j], end='')
    print()
```

Результат:

```
[[38, 21, 71, 71], [7, 68, 74, 72], [23, 98, 60, 44]]

 38 21 71 71
  7 68 74 72
 23 98 60 44
```

Сначала мы выводим матрицу в одну строку только для того, чтобы показать, как данные хранятся в списке.

Обращение к элементам матрицы происходит через их индексы. Сначала указывается номер строки, затем номер столбца. Так, чтобы в примере выше извлечь число 7, надо написать выражение `a[1][0]`, потому что число 7 находится во второй строке, чей индекс 1, и в первом столбце, чей индекс 0.

В Python из-за возможностей цикла `for` не обязательно пользоваться индексами при выполнении перебора. Цикл вывода на экран может выглядеть и так:

```
for i in a:
    for j in i:
        print("%3d" % j, end='')
    print()
```

Здесь мы извлекаем сначала очередной список из *a*. После перебираем этот вложенный список во вложенном цикле.

При изучении программирования, когда дело доходит до матриц, то рассматривается множество задач: на нахождение определенных элементов, вычисление сумм, сортировку и тому подобное. В основном эти задачи связаны с перебором элементов то так, то этак. В Python многое упрощается в связи с наличием встроенных функций и методов. Хотя никто не мешает использовать и изучать классические алгоритмы.

Пусть надо отсортировать элементы строк матрицы. Можно воспользоваться одним из алгоритмов сортировки, например, методом пузырька:

```
from random import randint
N = 3
M = 4

# функция вывода матрицы на экран
def output_matrix(matrix):
    for row in matrix:
        for item in row:
            print("%3d" % item, end='')
        print()

# заполнение матрицы
a = []
for i in range(N):
    b = []
    for j in range(M):
        b.append(randint(1,99))
    a.append(b)

# вывод матрицы до сортировки
output_matrix(a)
print()

# сортировка в строках матрицы
for i in range(N):
    for j in range(M-1):
        for k in range(M-j-1):
            if a[i][k] > a[i][k+1]:
                a[i][k], a[i][k+1] = a[i][k+1], a[i][k]

# вывод матрицы после сортировки
output_matrix(a)
```

Пример выполнения программы:

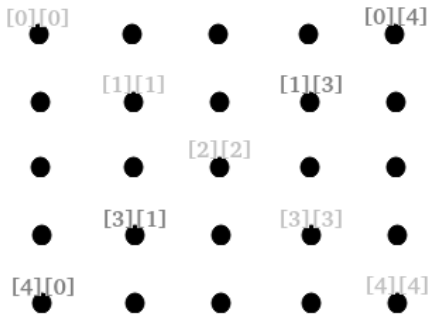
```
 67 48 51 72
 14 82 64 54
 98 48 18 33

 48 51 67 72
 14 54 64 82
 18 33 48 98
```

С другой стороны, у списка есть метод `sort()`, что упрощает решение подобной задачи. В результате часть, отвечающая за сортировку, может выглядеть проще:

```
# сортировка в строках матрицы
for i in range(N):
    a[i].sort()
```

В квадратных матрицах (у таких количество строк равно количеству столбцов) выделяют диагонали. Главная диагональ идет от верхнего левого угла к нижнему правому. Побочная диагональ – от верхнего правого к нижнему левому. Первый и второй индекс любого элемента главной диагонали одинаковые. Так первый элемент [0][0], второй [1][1].

Индексы каждого элемента побочной диагонали в сумме должны давать размерность матрицы за минус 1 (если индексация идет с нуля).



Вычисление суммы элементов каждой диагонали выполняется без вложенных циклов:

```
from random import randint
N = 5

# заполняем и сразу выводим
a = []
for i in range(N):
    b = []
    for j in range(N):
        n = randint(1, 9)
        b.append(n)
        print("%3d" % n, end='')
    a.append(b)
    print()

# находим сумму элементов главной и побочной диагоналей
diagonal1 = 0
diagonal2 = 0
for i in range(N):
    diagonal1 += a[i][i]
    diagonal2 += a[i][N-1-i]

print(diagonal1)
```

```
print(diagonal2)
```

Пример выполнения:

```
  6  8  9  5  5
  7  1  5  3  8
  1  5  4  4  8
  6  9  5  3  4
  2  2  7  7  1
15
23
```

Практическая работа

1. Заполните матрицу случайными положительными и отрицательными числами. Выведите ее на экран так, чтобы вместо отрицательных чисел стояли прочерки, например:

```
  2  -  5  6
  -  0  -  -
  5  4  -  -
```

2. Найдите сумму элементов каждого столбца матрицы.

Урок 28. Генераторы списков, или списочные выражения

В программировании списки или массивы – часто используемые структуры данных. Нередко надо выполнить их предварительное заполнение элементами. Делается это с помощью цикла. Например:

```
>>> a = []
>>> for i in range(10):
...     a.append(i+1)
...
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Пример заполнения случайными числами:

```
>>> import random
>>> a = []
```

```
>>> for i in range(10):
...     a.append(random.randint(1,50))
...
>>> a
[31, 16, 36, 29, 8, 28, 40, 36, 4, 19]
```

Однако в Python подобные вещи можно делать проще – с помощью генератора списка (списочного выражения). Генераторы списков относят к так называемому "синтаксическому сахару" языков программирования. Это значит, что без них можно обойтись, что доказывают примеры выше, но с ними кода становится меньше.

Создание последовательности натуральных чисел до десяти с помощью генератора списка будет выглядеть так:

```
>>> a = [i+1 for i in range(10)]
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Первое, на что следует обратить внимание в выражении `[i+1 for i in range(10)]`, – это квадратные скобки, которые обрамляют всю конструкцию. Именно квадратные скобки говорят нам, что создается список, а не что-то еще. Если использовать круглые, вы получите объект-генератор – совсем другую вещь. Если использовать фигурные, то получится либо словарь, либо множество.

```
>>> b = (i+1 for i in range(10))
>>> b
<generator object <genexpr> at 0x7f03a32b4990>
>>> c = {i+1 for i in range(10)}
>>> c
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
>>> d = {str(i+1):i+1 for i in range(10)}
>>> d
{'1': 1, '2': 2, '3': 3, '4': 4, '5': 5, '6': 6, '7': 7, '8': 8, '9': 9, '10':
10}
>>> type(a), type(b), type(c), type(d)
(<class 'list'>, <class 'generator'>, <class 'set'>, <class 'dict'>)
```

Теперь разберем саму конструкцию, находящуюся внутри скобок, то есть `i+1 for i in range(10)`. В ней можно выделить три части, которые задает программист, и, следовательно, их содержание может быть произвольным. Это

1. то, что находится до слова **`for`**;
2. то, что находится между **`for`** и **`in`**;
3. то, что находится после слова **`in`**.

Ключевые слова `for` и `in`, как и квадратные скобки, – неизменные элементы конструкции.

До `for` указывается элемент, который будет помещаться в список на каждой итерации цикла `for`. С элементом можно производить какие-либо действия. По-сути это вынесенное вперед тело цикла. Если тело нельзя записать в одно выражение, то от использования генератора списка придется оказаться. С другой стороны, вы можете увидеть подобное:

```
>>> c = [(i+1, (i+1)/2) for i in range(4)]
>>> c
[(1, 0.5), (2, 1.0), (3, 1.5), (4, 2.0)]
```

Однако здесь на каждой итерации лишь создается кортеж, состоящий из двух элементов.

Конструкция **for … in …** аналогична заголовку обычного цикла `for`. На каждой итерации мы извлекаем из объекта, указанного после `in`, очередной элемент и присваиваем его переменной, указанной до `in`. Причем этих переменных может быть несколько:

```
>>> c
[(1, 0.5), (2, 1.0), (3, 1.5), (4, 2.0)]
>>> d = [i+j for i, j in c]
>>> d
[1.5, 3.0, 4.5, 6.0]
```

В примере из списка *c* на каждой итерации цикла извлекается очередной кортеж и автоматически распаковывается так, что его первый элемент присваивается переменной *i*, второй – *j*.

После `in` может стоять любой объект, который способен превращаться в итератор (итерируемый объект, например, список или строка) или сам по себе являющийся итератором (например, файловый объект).

Существуют расширенные варианты генераторов списков. Во-первых, в нем может быть второй (и даже третий) цикл `for`, выполняющий роль вложенного в первый. Во вторых, после `for` может стоять условный оператор `if`, который проверяет на что-то элемент. Элемент добавляется в список, только если проходит проверку.

Пример с двойным `for`:

```
>>> s1 = 'abcd'
>>> s2 = '01'
>>> [i+j for i in s1 for j in s2]
['a0', 'a1', 'b0', 'b1', 'c0', 'c1', 'd0', 'd1']
```

Пример с `if`:

```
>>> a
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [i for i in a if i%2==0]
[2, 4, 6, 8, 10]
```

Пример с двойным `for` и `if`:

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> [(i,j) for i in a for j in b if i*j <= 10]
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5)]
```

В английском языке генераторы списков называют list comprehension, а генераторами называют одну из разновидностей итераторов. Поэтому не следует путать генераторы списков и генераторы. Генератор списка – это списочное выражение, синтаксическая конструкция, которая создает список. А настоящий генератор – это объект, который создается генераторным выражением лишь подобным генератору списка, или с помощью функции, содержащей инструкцию `yield`.

Генераторы – это разновидность объектов-итераторов, философию которых проще понять после изучения объектно-ориентированного программирования. Поэтому в данном курсе генераторы не рассматриваются.

Практическая работа

1. Перепишите три последних примера из урока с помощью обычного цикла `for` (то есть без использования генераторов списков).
2. Напишите генератор списка, который заполняет список данными, которые вводит пользователь. Другой генератор списка должен преобразовывать данные в числа. Если какой-либо элемент первого списка нельзя преобразовать в число, то во второй список этот элемент попадать не должен.

Урок 29. Lambda-выражения

Lambda-выражение – это особая конструкция языка программирования Python, в результате выполнения которой создается анонимная объект-функция.

```
>>> a = lambda i,j: i+j
>>> def aa(i, j):
...     return i + j
...
>>> type(a), type(aa)
(<class 'function'>, <class 'function'>)
>>> a(1,2)
3
>>> aa(1,2)
3
```

В примере создаются две функции, делающие одно и то же. Первая создается с помощью lambda-выражения, вторая – обычным способом, с помощью инструкции `def`. Первый способ – это не только более компактный вариант записи простых функций. Иногда lambda-выражения удобнее и позволяют создать функции там, где `def` использовать нельзя.

Кстати, обычная функция, если содержит всего одно выражение в теле, может быть записана в одну строку:

```
def summa(x, y): return x + y
```

Почему lambda-функция анонимная? Когда функция создается с помощью `def`, то она не может существовать без имени. Такая функция обязательно связывается с идентификатором, указанным в ее заголовке. Так сказать, имеет место неявное присваивание. В примере выше это *aa*.

Lambda-функция имени не имеет, в `lambda параметры: выражение` никакого имени нет. То, что мы потом явно связываем эту функцию с переменной, исключительно дело наше. Могли бы так не делать.

```
>>> a = (lambda i,j: i+j)(1,2)
>>> a
3
>>> (lambda i,j: i+j)(4,2)
6
```

Здесь в первом случае переменной *a* присваивается значение, которое возвращает функция, а не сама функция. То есть лямбда-выражение возвращает неименованную функцию, которая сразу вызывается. То, что она вернула, присваивается переменной. Сама функция при этом теряется.

Пример коллекции функций:

```
>>> a = [(lambda i,j:i+j), (lambda i,j:i-j), (lambda i: i//2)]
>>> a[0](1,4)
5
>>> a[1](1,4)
-3
>>> a[2](10)
5
```

Подобное бывает удобно для последующей, своего рода, пакетной обработки данных. Если индексы ни о чем не говорят, можно использовать словарь:

```
>>> b = {'+':(lambda i,j:i+j), '-':(lambda i,j:i-j)}
>>> b['+'](1098, 325)
1423
```

Рассмотрим еще один пример, где используют лямбда-функцию. Допустим, у нас есть матрица, которую мы хотим отсортировать:

```
>>> a = [[1, 10], [0, 15], [-1, 12], [3, 9]]
>>> a.sort()
>>> a
[[-1, 12], [0, 15], [1, 10], [3, 9]]
```

По-умолчанию сортировка выполняется по первому элементу каждого вложенного списка. Чтобы изменить принцип сортировки, надо в метод `sort()` передать значение для параметра **`key`**. Конечно, мы можем определить обычную функцию:

```
>>> def num_item(i): return i[1]
>>> a.sort(key=num_item)
>>> a
[[3, 9], [1, 10], [-1, 12], [0, 15]]
```

Однако будет проще с lambda-выражением:

```
>>> a.sort()
>>> a
[[-1, 12], [0, 15], [1, 10], [3, 9]]
>>> a.sort(key=lambda i: i[1])
>>> a
[[3, 9], [1, 10], [-1, 12], [0, 15]]
```

Практическая работа

1. Создайте список лямбда-выражений. Одно складывает аргументы, другое – вычитает, третье – умножает, четвертое – делит. Запросите у пользователя два числа и в цикле "прогоните" их через все lambda-функции списка.
2. Дан список строк. Отсортируйте его сначала по последним буквам слов, потом по длине слов.

Урок 30. Сортировка списков

В Python для сортировки списков предусмотрен метод `sort`, также можно использовать встроенную в Python функцию `sorted`. Основная разница между ними заключается в том, что метод списка сортирует сам список, к которому применяется (говорят, сортировка происходит на месте), а функция `sorted` возвращает новый список, оставляя оригинал неизменным.

```
>>> a = [1, -2, 0]
>>> b = [10, -20, 3]
>>> a.sort()
>>> c = sorted(b)
>>> a
[-2, 0, 1]
>>> b
[10, -20, 3]
>>> c
[-20, 3, 10]
```

Как `sort`, так и `sorted` принимают аргумент `True` или `False` по ключу `reverse`. Значение `False` задано по-умолчанию, поэтому сортировка происходит от меньшего к большему, то есть по возрастанию. Для обратной сортировки, то есть по убыванию, параметру `reverse` следует присвоить значение `True`.

```
>>> a = [4, 2, 5, 3]
>>> a.sort(reverse=True)
>>> a
[5, 4, 3, 2]
```

Кроме этого `sort` и `sorted` имеют параметр `key`, которому присваивается функция. Об этом упоминалось в прошлом уроке, где сортировка выполнялась по вторым элементам вложенных списков. При этом `key` не обязательно используется только при наличии вложенных списков.

```
>>> a = [1, 2, 3, 4, 5, 6]
>>> sorted(a, key=lambda i: i%2)
```

```
[2, 4, 6, 1, 3, 5]
```

Здесь мы отсортировали так, что сначала идут четные элементы, потом нечетные.

В результирующий список помещаются элементы исходного списка. Однако сортировка происходит не в зависимости от их непосредственного значения, а в зависимости от того, что возвращает функция, переданная `key`, примененная по отношению к каждому элементу.

В примере выше, если применить lambda-функцию к каждому элементу списка-оригинала, то получится такой список-маска: [1, 0, 1, 0, 1, 0]. Он и будет отсортирован. Однако в список-результат вместо значений из маски будут подставлены связанные с ними значения из списка-оригинала.

Тот факт, что у нас внутри себя отсортированы как четные (2, 4, 6), так и нечетные (1, 3, 5) числа, никак не связан с выполненной сортировкой. Это связано только с тем, как элементы были заданы в исходном списке. Сравните:

```
>>> a = [6, 4, 1, 5, 3, 2]
>>> sorted(a, key=lambda i: i%2)
[6, 4, 2, 1, 5, 3]
```

Параметру `key` не обязательно присваивать лямбда-функцию. Это может быть обычная функция или даже встроенная.

```
>>> s = ['bb', 'd', 'cccc', 'aaa']
>>> sorted(s)
['aaa', 'bb', 'cccc', 'd']
>>> sorted(s, key=len)
['d', 'bb', 'aaa', 'cccc']
>>> sorted(s, key=len, reverse=True)
['cccc', 'aaa', 'bb', 'd']
```

Здесь мы сортируем строки, которые по-умолчанию сортируются лексикографически. Однако используя встроенную функцию `len`, которая в данном случае возвращает длину строки, мы можем сортировать в зависимости от длины строк.

Рассмотрим пример с обычной функцией. В программе ниже список представляет собой маленькую базу данных. Каждый элемент содержит сведения о юном спортсмене: имя, возраст, рост и вес. Пользователь может заказать сортировку по любому полю. Мы сортируем с помощью функции `sorted`, чтобы исходный список оставался неизменным.

```
def sort_col(item):
    return item[int(n)]


a = [['Петя', 10, 130, 35], ['Вася', 11, 135, 39],
```

```
     ['Женя', 9, 120, 33], ['Дима', 10, 128, 30]]

n = input('Сортировать по имени(0), '
          'возрасту(1), росту(2), весу(3): ')

b = sorted(a, key=sort_col)

for i in b:
    print("%7s %3d %4d %3d" % (i[0], i[1], i[2], i[3]))
```

Если выбрать сортировку по весу, вывод будет таким:

```
Сортировать по имени(0), возрасту(1), росту(2), весу(3): 3
   Дима  10  128  30
   Женя   9  120  33
   Петя  10  130  35
   Вася  11  135  39
```

Отдельно посмотрим на определение функции *sort_col*:

```
def sort_col(item):
    return item[int(n)]
```

Она принимает некий объект *item*, извлекает из него элемент под номером *n* и возвращает этот элемент.

При использовании *sort_col* в качестве значения параметра `key` функции `sorted` происходит вызов *sort_col* по отношению к каждому элементу списка. В данном случае элементом списка является вложенный список условного формата [имя, возраст, рост, вес], он передается в функцию. Функция *sort_col* извлекает из него элемент под заданным номером *n* и передает этот элемент функции `sorted`. В свою очередь `sorted` сортирует элементы исходного списка, исходя из соответствующих значений, которые вернул *sort_col*.

Здесь вместо обычной можно использовать lambda-функцию:

```
b = sorted(a, key=lambda item: item[int(n)])
```

Однако если требуются более сложная обработка данных без полноценной функции не обойтись.

Практическая работа

Доработайте приведенную в уроке программу про юных спортсменов таким образом, чтобы пользователь выбирал не только столбец сортировки, также мог указать тип сортировки – по возрастанию или убыванию.

Урок 31. Фильтрация списков

Помимо сортировки частым действием по отношению к спискам (или другим составным объектам) является их фильтрация. Например, надо убрать из списка отрицательные значения, или пустые строки, или оставить в нем только объекты с определенными свойствами.

В Python для решения подобных задач достаточно написать небольшой код. Пусть из смешанного списка требуется отфильтровать только числа. Код без "синтаксического сахара" будет выглядеть примерно так:

```
a = ['one', 5, 'two', 4.22, 8, 'j', 0.35]
b = []
for item in a:
    if type(item) in (int, float):
        b.append(item)
print(b)
```

Результат:

```
[5, 4.22, 8, 0.35]
```

Здесь фильтрация выполняется в цикле `for`. Поскольку цикл достаточно прост, мы можем заменить его списочным выражением:

```
a = ['one', 5, 'two', 4.22, 8, 'j', 0.35]
b = [item for item in a if type(item) in (int, float)]
print(b)
```

Однако это не всё. В Python есть функция `filter`, которая непосредственно предназначена для решения задач фильтрации. Он принимает два аргумента: первый – функцию, второй – итерируемый объект (последовательность, итератор, или подобное). В данном случае в качестве второго параметра мы рассматриваем только списки.

Функция, которая передается первым аргументом в `filter`, вызывается на каждый элемент списка. Если эта функция возвращает что угодно, но не объекты `None`, `False` и их эквиваленты (`0`, `[]`, `''` и т. п.), то переданный ей текущий элемент списка проходит фильтрацию, он остается.

В `filter` можно передавать как обычную, так и лямбда-функцию. Рассмотрим сначала пример с обычной функцией:

```
def numbers(item):
    if type(item) in (int, float):
        return True
    else:
        return False
```

```
a = ['one', 5, 'two', 4.22, 8, 'j', 0.35]
b = filter(numbers, a)
print(list(b))
```

Обратим внимание, **функция `filter` возвращает** не список, а **итератор**. Для простоты вывода на экран мы преобразовываем его в список.

В функции *numbers* мы прописываем ветку **`else`** для ясности кода. Однако ее можно опустить.

```
def numbers(item):
    if type(item) in (int, float):
        return True
```

Фильтрация и в этом случае будет работать также. Почему? Когда в *numbers* передается не число, выражение `return True` не выполняется. Получается, что функция ничего не возвращает. На самом деле в таких случаях функция возвращает объект **`None`**. В свою очередь для функции **`filter`** что **`None`**, что **`False`** – это сигнал отбросить элемент.

Раз тело *numbers* упрощается, заменим ее на lambda-функцию:

```
a = ['one', 5, 'two', 4.22, 8, 'j', 0.35]
b = filter(lambda item: type(item) in (int, float), a)
print(list(b))
```

Выражение `type(item) in (int, float)` логическое, оно возвращает либо **`True`**, либо **`False`**.

Функция **`filter`** в качестве первого аргумента может принимать не только другую функцию, но и объект **`None`**. В этом случае элементы списка будут оцениваться сами по себе как **`True`** или **`False`** (или их эквиваленты). В примере ниже из первого списка будут изъяты нули, из второго – пустые строки.

```
a = [8, -1, 0, 3, 0]
b = ['a', 'b', '', 'dd']
f_a = filter(None, a)
f_b = filter(None, b)
print(list(f_a))
print(list(f_b))
```

Результат:

```
[8, -1, 3]
['a', 'b', 'dd']
```

Практическая работа

Пользователь вводит два натуральных числа, первое больше второго. Используя функцию `filter`, вывести на экран все натуральные числа, кратные второму числу и не превышающие первое. Например, были введены 10 и 3. Вывод: 3, 6, 9.

Урок 32. Функция zip

В Python есть встроенная функция `zip`, которая позволяет как бы "застегивать" вместе несколько передаваемых ей итерируемых объектов (например, списков). Делает она это путем объединения элементов разных объектов, стоящих в одинаковых позициях.

```
>>> a = [1, 2, 3, 4]
>>> b = ['a', 'b', 'c']
>>> c = ['I', 'II', 'III', 'IV']
>>> for i in zip(a, b, c):
...     print(i)
...
(1, 'a', 'I')
(2, 'b', 'II')
(3, 'c', 'III')
```

Как видно из примера объект, который возвращает функция `zip`, состоит из кортежей. Первый кортеж объединяет первые элементы переданных в `zip` списков. Второй – вторые и так далее. Количество кортежей равно длине наименьшего списка.

На самом деле объект типа `zip` не состоит из кортежей, он их генерирует, так как является объектом-итератором. Это значит, что получить значения из экземпляра `zip` можно только один раз.

```
>>> z = zip(a, b)
>>> z
<zip object at 0x7fc7697a4800>
>>> for i in z:
...     print(i)
...
(1, 'a')
(2, 'b')
(3, 'c')
>>> for i in z:
...     print(i)
...
>>>
```

При этом из того, что возвращает **zip**, можно сразу получить структуру данных, воспользовавшись соответствующей встроенной функцией:

```
>>> a = [5, 6, 3]
>>> b = [1.1, -0.05, 0.3]
>>> ab = list(zip(a, b))
>>> ab
[(5, 1.1), (6, -0.05), (3, 0.3)]
>>> ab = tuple(zip(a, b))
>>> ab
((5, 1.1), (6, -0.05), (3, 0.3))
```

С помощью **zip** можно получить так называемую транспонированную матрицу, когда строки становятся столбцами, а столбцы строками.

```
>>> m = [('a', 'b', 'c'), ('A', 'B', 'C')]
>>> m2 = list(zip(*m))
>>> m2
[('a', 'A'), ('b', 'B'), ('c', 'C')]
>>>
>>> for row in m:
...     print(row)
...
('a', 'b', 'c')
('A', 'B', 'C')
>>>
>>> for row in m2:
...     print(row)
...
('a', 'A')
('b', 'B')
('c', 'C')
```

В примере выражение `*m`, которое передается в функцию **zip**, обозначает распаковку списка. В результате в функцию передается не сам список, а два кортежа, в каждом из которых по три элемента. Первые элементы формируют первый кортеж нового объекта, вторые – второй, третьи – третий кортеж.

Если zip-экземпляр присваивать нескольким переменным, то он сразу проитерируется, и каждая переменная получит свой кортеж.

```
>>> a = [1, 2, 3]
>>> b = [-1, -2, -3]
>>> x, y, z = zip(a, b)
>>> x
```

```
(1, -1)
>>> y
(2, -2)
>>> z
(3, -3)
```

Понятно, что при этом количество переменных слева от знака присваивания должно совпадать с количеством кортежей в zip-объекте.

Часто кортежи сразу распаковывают в заголовке цикла `for`. Так делают, если над элементами разных итерируемых объектов надо выполнять однотипные действия или производить операции между ними самими.

```
>>> qty = [2, 5, 1, 4]
>>> price = [100, 91, 34, 15]
>>> for q, p in zip(qty, price):
...     print(q * p)
...
200
455
34
60
```

Когда требуется объединить итерируемые объекты разной длины так, чтобы элементы самого длинного не были утеряны, функция `zip` не подходит. В этом случае можно воспользоваться функцией `zip_longest` из модуля `itertools`.

```
>>> from itertools import zip_longest
>>>
>>> a = [1, 2, 3, 4]
>>> b = ['a', 'b', 'c']
>>>
>>> list(zip_longest(a, b))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, None)]
>>>
>>> list(zip_longest(a, b, fillvalue='?'))
[(1, 'a'), (2, 'b'), (3, 'c'), (4, '?')]
```

Практическая работа

Вычислите суммы столбцов матрицы. Используйте в программе функцию `zip`.

Решения и пояснения к практическим работам

Урок 4. Типы данных. Переменные

1. Переменной *var_int* присвойте значение 10, *var_float* - значение 8.4, *var_str* - "No".
2. Значение, хранимое в переменной *var_int*, увеличьте в 3.5 раза. Полученный результат свяжите с переменной *var_big*.
3. Измените значение, хранимое в переменной *var_float*, уменьшив его на единицу, результат свяжите с той же переменной.
4. Разделите *var_int* на *var_float*, а затем *var_big* на *var_float*. Результат данных выражений не привязывайте ни к каким переменным.
5. Измените значение переменной *var_str* на "NoNoYesYesYes". При формировании нового значения используйте операции конкатенации (`+`) и повторения строки (`*`).
6. Выведите значения всех переменных.

Решение

```
>>> var_int = 10
>>> var_float = 8.4
>>> var_str = 'No'
>>> var_big = var_int * 3.5
>>> var_float = var_float - 1
>>> var_int / var_float
1.3513513513513513
>>> var_big / var_float
4.72972972972973
>>> var_str = var_str * 2 + 'Yes' * 3
>>> var_int
10
>>> var_float
7.4
>>> var_str
'NoNoYesYesYes'
>>> var_big
35.0
```

Урок 5. Ввод и вывод данных

1. Напишите программу (файл *user.py*), которая запрашивала бы у пользователя:
- его имя (например, "What is your name?")
- возраст ("How old are you?")
- место жительства ("Where are you live?")

После этого выводила бы три строки:
"This is *имя*"
"It is *возраст*"
"(S)he live in *место_жительства*"

Решение

```
u_name = input("What is your name? ")
old = input("How old are you? ")
live = input("Where are you live? ")
print("This is", u_name)
print("It is", old)
print("(S)he lives in ", live)
```

2. Напишите программу (файл *arithmetic.py*), которая предлагала бы пользователю решить пример 4 * 100 - 54. Потом выводила бы на экран правильный ответ и ответ пользователя. Подумайте, нужно ли здесь преобразовывать строку в число.

Решение

```
ex = input("4 * 100 - 54 = ")
print("Right answer is", 346)
print("Your answer is", ex)
```

Преобразовывать строку, введенную пользователем, к числу не обязательно, так как никакие математические операции с ним, в том числе сравнение, не выполняются.

3. Запросите у пользователя четыре числа. Отдельно сложите первые два и отдельно вторые два. Разделите первую сумму на вторую. Выведите результат на экран так, чтобы ответ содержал две цифры после запятой.

Решение

```
n1 = float(input())
n2 = float(input())
n3 = float(input())
n4 = float(input())
n12 = n1 + n2
n34 = n3 + n4
print("%.2f" % (n12/n34))
```

Урок 6. Логические выражения и операторы

1. Присвойте двум переменным любые числовые значения.
2. Используя переменные из п. 1, с помощью оператора `and` составьте два сложных логических выражения, одно из которых дает истину, другое – ложь.
3. Аналогично выполните п. 2, но уже с оператором `or`.

Решение

```
>>> a = 3
>>> b = 4
>>> a > 0 and b < 10
True
>>> a > b and a + b < 10
False
>>> a > b or a + b < 10
True
>>> a == 0 or b == 0
False
```

4. Попробуйте использовать в логических выражениях переменные строкового типа. Объясните результат.

Решение

```
>>> s1 = "hello"
>>> s2 = "world"
>>> s1 > s2
False
```

Строки сравниваются по символам. Числовой код буквы 'h' меньше кода 'w', поэтому "hello" меньше "world", а не больше. Если первые символы совпадают, то начинают сравниваться вторые и так далее:

```
>>> "he" > "hal"
True
```

При использовании логических операторов `and` и `or` к строкам результат имеет не булевый, а строковый тип:

```
>>> s1 and s2
'world'
>>> s1 or s2
'hello'
```

Здесь не происходит посимвольного сравнения. Оператор `and` возвращает вторую строку, если первая не пуста. Иначе первую. Оператор `or` возвращает первую, если она не пуста. Иначе вторую.

5. Напишите программу, которая запрашивала бы у пользователя два числа и выводила бы `True` или `False` в зависимости от того, больше первое число второго или нет.

Решение

```
n1 = int(input())
n2 = int(input())
print(n1 > n2)
```

Урок 7. Ветвление. Условный оператор

1. Напишите программу, которая просит пользователя что-нибудь ввести с клавиатуры. Если он вводит какие-нибудь данные, то на экране должно выводиться сообщение "OK". Если он не вводит данные, а просто нажимает Enter, то программа ничего не выводит на экран.

Решение

```
if input() != '':
    print("OK")
```

Если ничего не введено, то функция `input()` возвращает пустую строку. Следовательно, если результат не равен пустой строке, то значит, что-то было введено.

Поскольку пустая строка (не содержащая ни одного символа) сама по себе соответствует `False`, то сравнение введенной строки с пустой строкой здесь лишнее. Задачу можно решить еще проще:

```
if input():
    print('OK')
```

2. Напишите программу, которая запрашивает у пользователя число. Если оно больше нуля, то в ответ на экран выводится число 1. Если введенное число не является положительным, то на экран должно выводиться -1.

Решение

```
a = int(input())
if a > 0:
    print(1)
else:
    print(-1)
```

Урок 8. Исключения и их обработка в Python

Напишите программу, которая запрашивает ввод двух значений. Если хотя бы одно из них не является числом, то должна выполняться конкатенация, то есть соединение, строк. В остальных случаях введенные числа суммируются.

Решение

Решить задачу можно несколькими способами. Наиболее элегантным вариантом может быть использование дополнительных переменных для сохранения численных значений. Если преобразование прошло успешно, то складывать значения этих переменных. Если же возникает исключение, то складывать значения "старых" переменных, в которых по-прежнему хранятся строки:

```
a_str = input("Первое значение: ")
b_str = input("Второе значение: ")

try:
    a_num = float(a_str)
    b_num = float(b_str)
    print("Результат:", a_num + b_num)
except ValueError:
    print("Результат:", a_str + b_str)
```

Причина необходимости ввода дополнительных переменных в том, что мы не знаем, какое из двух исходных значений окажется не числом. Если первая переменная была успешно преобразована к числовому типу, а вторая – нет, то попытка их сложения приведет к исключению `TypeError` (нельзя складывать число и строку). Вот как выглядела бы программа, содержащая подобный неявный баг:

```
a = input("Первое значение: ")
b = input("Второе значение: ")

try:
    a = float(a)
    b = float(b)
    print("Результат:", a + b)
except ValueError:
    print("Результат:", a + b)
```

Она прекрасно работает в трех случаях:

- когда вводятся два числа;
- когда вводятся две строки;
- когда первое значение строка, а второе – число.

Однако этот код генерирует ошибку, когда первое значение число, а второе – строка. В этом случае переход к ветке `except` происходит уже после того, как значение переменной *a* преобразовано в тип `float`. В результате в теле `except` мы пытаемся сложить число и строку, что невозможно.

Для правильного решения не обязательно вводить дополнительные переменные. В теле `try` можно просто попробовать преобразовать к числу. Если попытка для обоих значений будет удачной, то программа перейдет на ветку `else`, где следует повторно преобразовать к числу. В случае неудачи поток выполнения программы окажется в теле `except`:

```
...
try:
    float(a)
    float(b)
except ValueError:
    print("Результат:", a + b)
else:
    print("Результат:", float(a) + float(b))
```

Есть и другие способы решения. Можно обратно преобразовать число в строку. Однако тут есть одна особенность. Если мы используем функцию `float()`, а не `int()`, то у числа появляется дробная часть. Так при преобразовании строки "4" в тип `float` получаем 4.0, а преобразование 4.0 в строку дает "4.0". Это уже не совсем то, что ввел пользователь. Вот так это выглядит с функцией `int()`:

```
...
try:
    a = int(a)
    b = int(b)
    print("Результат:", a + b)
except ValueError:
    print("Результат:", str(a) + b)
```

Сложение и вывод на экран, чтобы не повторять их дважды, можно вынести в блок `finally`:

```
...
try:
    a = int(a)
    b = int(b)
except ValueError:
    a = str(a)
finally:
    print("Результат:", a + b)
```

В ветке `except` переменную *b* конвертировать необязательно, так как если исключение возникло на "уровне" переменной *a*, то до преобразования значения *b* к числу дело не дошло. Оно так и осталось строкой. Если же ошибка возникает на "уровне" *b*, то ее значение опять-таки остается строкой.

Можно извратиться сильнее и написать вложенную конструкцию **try-except-else**:

```
...
try:
    a = int(a)
except ValueError:
    print("Результат:", a + b)
else:
    try:
        b = int(b)
    except ValueError:
        a = str(a)
    finally:
        print("Результат:", a + b)
```

Ход выполнения здесь таков. Если *a* нельзя преобразовать к числу, то переходим к `except` внешнего блока и складываем строки. При этом внешнее `else` игнорируется. Если *a* преобразуется в число, то переходим к `else` и преобразуем *b*. В случае неудачи конвертируем *a* обратно в строку. В `finally` складываем либо строки, либо числа.

Еще один вариант:

```
...
try:
    a = int(a)
    b = int(b)
except ValueError:
    pass # ничего не делать

if type(a) == int and type(b) == str:
    a = str(a)

print("Результат:", a + b)
```

Здесь с помощью функции `type()` проверяется тип переменной. Команда `pass` служит заполнителем тела сложной инструкции, так как пустым (без единой строчки кода) оно быть не может.

Урок 9. Множественное ветвление: if-elif-else

1. Спишите вариант кода программы "про возраст" с `if` и тремя ветками `elif` из урока. Дополните его веткой `else`, обрабатывающие случаи, когда пользователь вводит числа не входящие в заданные четыре диапазона. Подумайте, почему в первой версии программы (когда использовались не связанные друг с другом условные операторы) нельзя было использовать `else`, а для обработки не входящих в диапазоны случаев пришлось бы писать еще один `if`?

Решение

Вот так выглядит программа с веткой `else`, обрабатывающей значения, не входящие в заданные диапазоны:

```
old = int(input('Ваш возраст: '))

print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')
elif 6 <= old < 12:
    print('"Марсианин"')
elif 12 <= old < 16:
    print('"Загадочный остров"')
elif 16 <= old:
    print('"Поток сознания"')
else:
    print('-')
```

Причина, по которой нельзя использовать ветку `else` в программе с отдельными `if`, заключается в том, что тело `else` сработает во всех случаях, когда логическое выражение при его `if` возвращает ложь. Однако до этого или после этого может сработать какое-то другое `if`, и в данном `else` необходимости уже нет. Допустим, программа была написана так:

```
old = int(input('Ваш возраст: '))

print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')

if 6 <= old < 12:
    print('"Марсианин"')

if 12 <= old < 16:
    print('"Загадочный остров"')
```

```
if 16 <= old:
    print('"Поток сознания"')
else:
    print('фильмов нет')
```

Здесь `else` относится только к последнему `if`. Пусть пользователь ввел число 10 и сработало второе `if`. Третье и четвертое `if` возвращают ложь, и вроде бы ничего страшного не происходит. Но у четвертого `if` есть ветка `else`. И ее тело будет выполнено! Ведь логическое выражение при `if`, которому оно принадлежит, вернуло ложь. В результате вывод программы будет таким:

```
Ваш возраст: 10
Рекомендовано: "Марсианин"
фильмов нет
```

Это пример логической ошибки, когда программа делает не то, что планировал программист. Чтобы такая программа с несвязанными `if` работала корректно, надо вместо `else` использовать еще один `if`:

```
...
if 16 <= old:
    print('"Поток сознания"')

if old < 3:
    print('-')
```

2. Усовершенствуйте предыдущую программу, обработав исключение `ValueError`, возникающее, когда вводится не целое число.

Решение

Можно впихнуть всю конструкцию множественного ветвления в тело `try`:

```
try:
    old = int(input('Ваш возраст: '))
    print('Рекомендовано:', end=' ')

    if 3 <= old < 6:
        print('"Заяц в лабиринте"')
    elif 6 <= old < 12:
        print('"Марсианин"')
    elif 12 <= old < 16:
        print('"Загадочный остров"')
    elif 16 <= old:
```

```
        print('"Поток сознания"')
    else:
        print('-')
except ValueError:
    print('Ошибка ввода')
```

Однако более нагляден такой код:

```
try:
    old = int(input('Ваш возраст: '))
except ValueError:
    print('Ошибка ввода')
    quit()
print('Рекомендовано:', end=' ')

if 3 <= old < 6:
    print('"Заяц в лабиринте"')
elif 6 <= old < 12:
    print('"Марсианин"')
elif 12 <= old < 16:
    print('"Загадочный остров"')
elif 16 <= old:
    print('"Поток сознания"')
else:
    print('-')
```

Здесь в случае возникновения исключения происходит досрочный выход из программы с помощью вызова функции `quit()`. При этом условный оператор остается в основной ветке программы.

3. Напишите программу, которая запрашивает на ввод число. Если оно положительное, то на экран выводится цифра 1. Если число отрицательное, выводится -1. Если введенное число – это 0, то на экран выводится 0. Используйте в коде условный оператор множественного ветвления.

Решение

При решении задачи можно использовать две ветки `elif`:

```
n = int(input())
if n > 0:
    print(1)
elif n < 0:
    print(-1)
elif n == 0:
    print(0)
```

Также последний `elif` можно заменить на `else`, поскольку, если число не положительное и не отрицательное, то это ноль.

```
n = int(input())
if n > 0:
    print(1)
elif n < 0:
    print(-1)
else:
    print(0)
```

Однако первый вариант кода с точки зрения наглядности лучше. Глядя на него, сразу ясно, в каких случаях выполняется та или иная ветка условного оператора. С другой стороны, с точки зрения эффективности кода лучше второй вариант, так как в нем процессору не приходится обрабатывать третье логическое выражение (`n == 0`).

Урок 10. Циклы в программировании. Цикл while

1. Измените последний код из урока так, чтобы переменная *total* не могла уйти в минус. Например, после предыдущих вычитаний ее значение стало равным 25. Пользователь вводит число 30. Однако программа не выполняет вычитание, а выводит сообщение о недопустимости операции, после чего осуществляет выход из цикла.

Решение

Для решения задачи надо проверять два условия. Первое, и оно проверяется в заголовке цикла, – переменная *total* должна быть больше 0. Лишь после этого имеет смысл запрашивать у пользователя очередное число.

После того как пользователь ввел число, надо проверить, не больше ли оно значения *total*. Это можно сделать, например, так: `n <= total`. Если подобное выражение возвращает истину, то следует выполнить вычитание *n* из *total*. Иначе, следует выйти из цикла. То есть мы будем использовать условный оператор, вложенный в тело цикла. Выход из цикла осуществляется с помощью оператора `break`.

```
total = 100

while total > 0:
    n = int(input())
    if n <= total:
        total = total - n
    else:
        print("Нельзя вычесть", n, "из", total)
        break
```

```
print("Осталось", total)
```

Пример выполнения программы:

```
6
70
45
Нельзя вычесть 45 из 24
Осталось 24
```

Код можно упростить, избавившись от ветки `else`, если `break` поместить в `if`, а его условие изменить на обратное:

```
total = 100

while total > 0:
    n = int(input())
    if n > total:
        print("Нельзя вычесть", n, "из", total)
        break
    total = total - n

print("Осталось", total)
```

Если происходит заход в тело `if`, то выполняется оператор `break`, и поток выполнения так и не доходит до выражения `total = total - n`.

2. Используя цикл `while`, выведите на экран для числа 2 его степени от 0 до 20. Возведение в степень в Python обозначается как `**`.

Решение:

Ответом к данной задаче может быть такой код:

```
i = 0
while i <= 20:
    print("%7d" % 2**i)
    i += 1
```

Благодаря форматированному выводу (`"%7d"`) числа выравниваются по правому краю поля из семи знакомест. Это позволяет выводить единицы под единицами, десятки под десятками и т. д.

Обратите внимание, что переменная *i* здесь используется и как счетчик для цикла, и как показатель степени.

Урок 11. Функции в программировании

В программировании можно из одной функции вызывать другую. Для иллюстрации этой возможности напишите программу по следующему описанию.

Основная ветка программы, не считая заголовков функций, состоит из одной строки кода. Это вызов функции `test()`. В ней запрашивается на ввод целое число. Если оно положительное, то вызывается функция `positive()`, тело которой содержит команду вывода на экран слова "Положительное". Если число отрицательное, то вызывается функция `negative()`, ее тело содержит выражение вывода на экран слова "Отрицательное".

Понятно, что вызов `test()` должен следовать после определения функций. Однако имеет ли значение порядок определения самих функций? То есть должны ли определения `positive()` и `negative()` предшествовать `test()` или могут следовать после него? Проверьте вашу гипотезу, поменяв объявления функций местами. Попробуйте объяснить результат.

Решение

Программа должна выглядеть примерно так:

```
def positive():
    print("Положительное")

def negative():
    print("Отрицательное")

def test():
    a = int(input())
    if a > 0:
        positive()
    elif a < 0:
        negative()

test()
```

Определения функций можно поменять местами:

```
def test():
    a = int(input())
    if a > 0:
        positive()
    elif a < 0:
        negative()

def positive():
    print("Положительное")

def negative():
```

```
    print("Отрицательное")

test()
```

Хотя по логике вещей, если `positive` и `negative` определяются позже по ходу программы, то они еще неизвестны в теле функции `test`. Поэтому второй вариант программы вроде бы должен вызывать ошибку.

На самом деле, когда интерпретатор читает код, то, встретив определение функции, он запоминает ее имя, но в тело не заходит. Потому что его незачем исполнять до вызова функции. До вызова функции интерпретатор в неведении о содержимом ее тела.

Таким образом программа исполняется в следующей последовательности:

1. Интерпретатор читает заголовок `def test()` и теперь знает о существовании функции `test`.
2. Читает заголовок `def positive()` и теперь знает о существовании функции `positive`.
3. Читает заголовок `def negative()` и теперь знает о существовании функции `negative`.
4. Переходит к вызову `test()`.
5. Интерпретатор идет туда, где запомнил это имя, и исполняет тело `test`.
6. В нем он встречает вызов либо функции `positive()`, либо `negative()`.
7. Он уже знает о них и идет к месту их определения.
8. Исполняет тело одной из них.
9. Возвращается в функцию `test`.
10. Завершает исполнение тела `test`.
11. Возвращается в основную ветку.
12. Завершает программу.

Урок 12. Локальные и глобальные переменные

В языке Python можно внутри одной функции определять другую. Напишите программу по следующему описанию.

В основной ветке программы вызывается функция `cylinder()`, которая вычисляет площадь цилиндра. В теле `cylinder` определена функция `circle`, вычисляющая площадь круга по формуле πr^2. В теле `cylinder` у пользователя спрашивается, хочет ли он получить только площадь боковой поверхности цилиндра, которая вычисляется по формуле $2\pi rh$, или полную площадь цилиндра. В последнем случае к площади боковой поверхности цилиндра должен добавляться удвоенный результат вычислений функции `circle()`.

Как вы думаете, можно ли из основной ветки программы вызвать функцию, вложенную в другую функцию? Почему?

Решение

Учитывая, что на данном этапе передача аргументов в функцию и возврат результата еще не изучались, для хранения площади круга придется использовать глобальную переменную:

```
def cylinder():

    def circle():
        global s_circle
        s_circle = 3.14 * r ** 2

    r = float(input("Радиус: "))
    h = float(input("Высота: "))

    s_cylinder = 2 * 3.14 * r * h

    if input("Боковую - 1, полную - 2: ") == '2':
        circle()
        s_cylinder += 2 * s_circle

    print(s_cylinder)


s_circle = 0

cylinder()
```

Имейте в виду, что подобный код с точки зрения нормального программирования – извращение. Мы прибегаем к нему лишь с целью прийти к мысли о необходимости возврата значений из функции.

Нельзя из основной ветки программы вызвать функцию, находящуюся в локальной области видимости другой функции. В глобальной области интерпретатор о ней ничего не знает, эта вложенная функция для него не существует.

Другое дело – классы, являющиеся базовым элементом объектно-ориентированного программирования. Функции внутри них называются методами. Такие функции зачастую доступны из глобальной области.

Урок 13. Возврат значений из функции. Оператор return

1. Напишите программу, в которой вызывается функция, запрашивающая с ввода две строки и возвращающая в программу результат их конкатенации. Выведите результат на экран.

Решение

Вариант решения:

```
def concat():
    a = input()
    b = input()
    c = a + b
    return c


print(concat())
```

По условию задачи функция должна только возвращать строку, она не должна выводить ее на экран. Поэтому решение, подобное приведенному ниже, не верно, так как не соответствует поставленной задаче:

```
def concat():
    a = input()
    b = input()
    c = a + b
    print(c)


concat()
```

Функцию можно упростить, отказавшись от переменной *c*:

```
def concat():
    a = input()
    b = input()
    return a+b
```

Или вообще от всех переменных:

```
def concat():
    return input() + input()
```

В последнем случае сначала выполняется первая функция `input()`, потом – вторая, потом операция конкатенации. После этого результат передается оператору `return`.

2. Напишите функцию, которая считывает с клавиатуры числа и перемножает их до тех пор, пока не будет введен 0. Функция должна возвращать полученное произведение. Вызовите функцию и выведите на экран результат ее работы.

Решение

Вариант решения:

```
def mult():
    mult = 1
    n = int(input())
    if n == 0:
        mult = 0
    while n != 0:
        mult *= n
        n = int(input())
    return mult


print(mult())
```

В теле функции переменная *mult* хранит произведение. Ее исходное значение не может быть нулем, так как впоследствии при умножении на нуль всегда будет получаться нуль.

Однако, если первое введенное пользователем число – это ноль, то данную переменную можно обнулить. Хотя это не обязательно.

Для подсчета произведения следует использовать цикл `while`. Условие продолжения цикла – значение *n* не равно 0. В теле цикла значение этой переменной должно меняться.

Урок 14. Параметры и аргументы функций

Напишите программу, в которой определена функция *int_test*, имеющая один параметр. Функция проверяет, можно ли переданное ей значение преобразовать к целому числу. Если можно, возвращает логическое `True`. Если нельзя – `False`.

В основной ветке программы присвойте переменной *s* то, что пользователь вводит с клавиатуры. Вызовите функцию `int_test()`, передав ей значение *s*. Если функция возвращает истину, преобразуйте строку *s* в число *n* и выведите на экран значение `n + 10`.

Решение

```
def int_test(a):
    try:
        int(a)
        return True
    except ValueError:
```

```
        return False


s = input()

if int_test(s) == True:
    n = int(s)
    print(n + 10)
```

В заголовке условного оператора сравнение результата `int_test(s)` с `True` не обязательно, так как допустимы записи `if True` (тело будет выполняться) или `if False` (тело не будет выполняться). Выражение `if int_test(s)` превращается в одну из них после вызова функции.

Урок 15. Встроенные функции

1. Напишите программу, которая в цикле запрашивает у пользователя номера символов по таблице Unicode и выводит соответствующие им символы. Завершает работу при вводе нуля.

Решение

```
c = int(input())
while c != 0:
    print(chr(c))
    c = int(input())
```

Условие выполнения тела цикла – значение *c* не равно нулю. Первый ввод символа должен находиться за пределами цикла, чтобы переменная *c* уже содержала число перед первой его проверкой.

Не обязательно вкладывать функцию `chr()` в `print()`. Можно присвоить результат преобразования другой переменной (или перезаписать эту же), после чего передать значение переменной в `print()`:

```
...
c = chr(c)
print(c)
...
```

2. Напишите программу, которая измеряет длину введенной строки. Если строка длиннее десяти символов, то выносится предупреждение. Если короче, то к строке добавляется столько символов `*`, чтобы ее длина составляла десять символов, после чего новая строка должна выводиться на экран.

Решение

```
s = input()
if len(s) > 10:
    print('Warning!')
else:
    diff = 10 - len(s)
    s = s + '*' * diff
    print(s)
```

В переменную *diff* записывается, сколько символов не хватает до десяти. Если длина *s* уже составляет 10 символов, то *diff* будет присвоено значение 0.

В выражении `s + '*' * diff` происходят операции конкатенации и мультипликации строки. Сначала символ '*' мультиплицируется *diff* раз. Например, если *diff* имеет значение 4, то получится строка '****'. Если же *diff* равно нулю, то строка станет пустой.

Далее получившаяся в результате мультипликации строка присоединяется к *s*. Результат присваивается переменной *s*, перезаписывая ее старое значение.

3. Напишите программу, которая запрашивает у пользователя шесть вещественных чисел. На экран выводит минимальное и максимальное из них, округленные до двух знаков после запятой. Выполните задание без использования встроенных функций `min()` и `max()`.

Решение

```
n = float(input())
minimum = n
maximum = n
i = 1
while i < 6:
    n = float(input())
    if n < minimum:
        minimum = n
    elif n > maximum:
        maximum = n
    i += 1
print("Minimum:", round(minimum, 2))
print("Maximum:", round(maximum, 2))
```

Инициируем переменные *minimum* и *maximum* первым введенным числом. Тем самым мы предполагаем, что оно является минимум, и оно является максимумом.

Поскольку одно число уже было "забрано" с ввода, то осталось "взять" только пять. Поэтому количество итераций цикла должно быть 5, а не 6. И поэтому счетчик *i* можно инициировать единицей, а не нулем. Хотя можно было бы и нулем, а заголовке цикла использовать выражение `i < 5`.

В теле цикла запрашивается очередное число. Если оно меньше хранимого минимума, то его следует сохранить. Старый минимум при этом затрется.

Иначе проверяется, не больше ли число хранимого максимума. Если так, то он перезаписывается.

В конце цикла `while` следует не забывать увеличивать счетчик.

Урок 16. Модули

Пример кода основного скрипта (имя файла, например, *main.py*):

```
import square

figure = input("Rectangle - 1, triangle - 2, circle - 3: ")
if figure == '1':
    a = float(input("Width: "))
    b = float(input("Height: "))
    print(square.rectangle(a, b))
elif figure == '2':
    a = float(input("Base: "))
    h = float(input("Height: "))
    print(square.triangle(a, h))
elif figure == '3':
    r = float(input("Radius: "))
    print(square.circle(r))
else:
    print("Input error")
```

Урок 17. Генератор псевдослучайных чисел

1. Используя функцию `randrange()` получите псевдослучайное четное число в пределах от 6 до 12. Также получите число кратное пяти в пределах от 5 до 100.

Решение:

```
random.randrange(6, 12, 2)
random.randrange(5, 100, 5)
```

2. Напишите программу, которая запрашивает у пользователя границы диапазона и какое (целое или вещественное) число он хочет получить. Выводит на экран подходящее случайное число.

Решение:

```
import random

int_or_float = input("Integer - 1, float - 2: ")
```

```
if int_or_float == '1':
    low = int(input("Low: "))
    high = int(input("High: "))
    n = random.randint(low, high)
    print(n)
elif int_or_float == '2':
    low = float(input("Low: "))
    high = float(input("High: "))
    n = random.random() * (high - low) + low
    print(round(n, 2))
```

Урок 18. Списки

1. Напишите программу, которая запрашивает с ввода восемь чисел, добавляет их в список. На экран выводит их сумму, максимальное и минимальное из них. Для нахождения суммы, максимума и минимума воспользуйтесь встроенными в Python функциями `sum()`, `max()` и `min()`.

Решение:

```
a = []
i = 0
while i < 8:
    a.append(int(input()))
    i += 1
print(sum(a), max(a), min(a))
```

2. Напишите программу, которая генерирует сто случайных вещественных чисел и заполняет ими список. Выводит получившийся список на экран по десять элементов в ряд. Далее сортирует список с помощью метода `sort()` и снова выводит его на экран по десять элементов в строке. Для вывода списка напишите отдельную функцию, в качестве аргумента она должна принимать список.

Решение:

```
import random


def output_list(lst):
    i = 1
    while i <= 100:
        print("%.2f" % lst[i-1], end=' ')
        if i % 10 == 0:
            print()
        i += 1
```

```
a = []
i = 0
while i < 100:
    a.append(random.random())
    i += 1
output_list(a)
a.sort()
print()
output_list(a)
```

Чтобы выводить по десять элементов в ряд, надо после каждого десятого (такой будет кратен десяти) переходить на новую строку. Однако, поскольку индексация списка начинается с нуля, прибегаем к хитрости: счетчик инициируется единицей, а при извлечении элемента из списка значение счетчика уменьшается на 1.

Урок 19. Цикл for

1. Заполните список случайными числами. Используйте в коде цикл `for`, функции `range()` и `randint()`.

Решение:

```
import random

a = []
for i in range(10):
    a.append(random.randint(20, 40))
```

2. Если объект `range` (диапазон) передать встроенной в Python функции `list()`, то она преобразует его к списку. Создайте таким образом список с элементами от 0 до 100 и шагом 17.

Решение:

```
>>> list(range(0, 100, 17))
[0, 17, 34, 51, 68, 85]
```

3. В заданном списке, состоящем из положительных и отрицательных чисел, посчитайте количество отрицательных элементов. Выведите результат на экран.

Решение:

```
a = [-1, 3, 9, -6, 12, 3, 0, -18, 4]
neg = 0
for i in a:
```

```
    if i < 0:
        neg += 1
print(neg)
```

Здесь нет необходимости в функции `range()`. Достаточно перебрать элементы списка и оценить каждый на отрицательность. Если условие выполняется, то счетчик отрицательных элементов увеличивается на 1.

4. Напишите программу, которая заполняет список пятью словами, введенными с клавиатуры, измеряет длину каждого слова и добавляет полученное значение в другой список. Например, список слов – ['yes', 'no', 'maybe', 'ok', 'what'], список длин – [3, 2, 5, 2, 4]. Оба списка должны выводиться на экран.

Решение:

```
N = 5
words = [''] * N
length = [0] * N
for i in range(N):
    words[i] = input()
    length[i] = len(words[i])
print(words)
print(length)
```

Задача не обязательно должна быть решена так. Можно было создать пустые списки и добавлять в них методом `append()`.

Тут же демонстрируется прием формирования одного списка путем мультипликации другого списка из одного элемента. Этот элемент повторяется в данном случае 5 раз. Получаются списки ['', '', '', '', ''] и [0, 0, 0, 0, 0]. В итоге в цикле `for` элементы не добавляются, а заменяются уже существующие.

В языке Python нет констант, но обозначение переменной заглавной буквой может служить соглашением, что она выполняет такую роль. Такая переменная используется в качестве постоянного значения во многих местах программы. Если потом потребуется поменять размерность списков, то придется менять значение только в одном месте кода, не надо просматривать все.

Урок 20. Функция enumerate

Дан список чисел. Используя функцию `enumerate()` в заголовке цикла `for`, создайте второй список, в котором каждый элемент должен быть строкой, включающей через пробел индекс и значение соответствующего элемента первого списка.

```
a = [4, -3, 8, 12]
b = []
```

```
for index, value in enumerate(a):
    s = str(index) + ' ' + str(value)
    b.append(s)

print(b)
```

Результат выполнения:

```
['0 4', '1 -3', '2 8', '3 12']
```

Урок 21. Строки

1. Вводится строка, включающая строчные и прописные буквы. Требуется вывести ту же строку в одном регистре, который зависит от того, каких букв больше. При равном количестве преобразовать в нижний регистр. Например, вводится строка "HeLLo World", она должна быть преобразована в "hello world", потому что в исходной строке малых букв больше. В коде используйте цикл `for`, строковые методы `upper()` (преобразование к верхнему регистру) и `lower()` (преобразование к нижнему регистру), а также методы `isupper()` и `islower()`, проверяющие регистр строки или символа.

Решение:

```
s = input()

n_small = 0
n_big = 0

for i in s:
    if i.islower():
        n_small += 1
    elif i.isupper():
        n_big += 1

if n_small >= n_big:
    print(s.lower())
else:
    print(s.upper())
```

Одиночный символ – та же строка. К нему применимы методы строк. В программе из исходной строки поочередно извлекаются символы, проверяется их регистр, и увеличивается счетчик больших или малых букв.

Если символ не нижнего регистра, еще не значит, что он верхнего, так как могут быть знаки препинания и т. п. Поэтому в цикле `for` нельзя использовать ветку `else`, а следует делать проверку на верхний регистр в `elif`.

2. Строковый метод `isdigit()` проверяет, состоит ли строка только из цифр. Напишите программу, которая запрашивает с ввода два целых числа и выводит их сумму. В случае некорректного ввода программа не должна завершаться с ошибкой, а должна продолжать запрашивать числа. Обработчик исключений try-except использовать нельзя.

Решение:

```
a = ''
while a.isdigit() == False:
    a = input('1st number: ')

b = ''
while b.isdigit() == False:
    b = input('2nd number: ')

a = int(a)
b = int(b)
print(a + b)
```

Урок 22. Кортежи

1. Чтобы избежать изменения исходного списка, не обязательно использовать кортеж. Можно создать его копию с помощью метода списка `copy()` или взять срез от начала до конца `[:]`. Скопируйте список первым и вторым способом и убедитесь, что изменение копий никак не отражается на оригинале.

Решение:

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = a[:]
>>> a, b, c
([1, 2, 3], [1, 2, 3], [1, 2, 3])
>>> b.append(4)
>>> c.insert(1, 'word')
>>> a, b, c
([1, 2, 3], [1, 2, 3, 4], [1, 'word', 2, 3])
```

2. Заполните один кортеж десятью случайными целыми числами от 0 до 5 включительно. Также заполните второй кортеж числами от -5 до 0. Для заполнения кортежей числами напишите одну функцию. Объедините

два кортежа с помощью оператора `+`, создав тем самым третий кортеж. С помощью метода кортежа `count()` определите в нем количество нулей. Выведите на экран третий кортеж и количество нулей в нем.

Вариант решения:

```
from random import randint

def fill_tuple(a, b):
    l = []
    for i in range(10):
        l.append(randint(a, b))
    return tuple(l)

first = fill_tuple(0, 5)
second = fill_tuple(-5, 0)
third = first + second
count_zero = third.count(0)
print(third)
print(count_zero)
```

Урок 23. Словари

1. Создайте словарь, связав его с переменной *school*, и наполните данными, которые бы отражали количество учащихся в разных классах (1а, 1б, 2б, 6а, 7в и т. п.). Внесите изменения в словарь согласно следующему: а) в одном из классов изменилось количество учащихся, б) в школе появился новый класс, с) в школе был расформирован (удален) другой класс. Вычислите общее количество учащихся в школе.

Решение:

```
school = {'1a': 20, '1b': 21, '6': 25, '7a': 15, '7b': 16}
print(school)

school['1a'] += 2
school['8'] = 10
del school['6']

s = 0
for qty in school.values():
    s += qty

print(school)
print(s)
```

Выражение `school['1a'] += 2` – то же самое, что `school['1a'] = school['1a'] + 2`, т. е. извлекается значение, к нему добавляется два, новое значение записывается по ключу.

Выполнение:

```
{'1a': 20, '1b': 21, '7a': 15, '7b': 16, '6': 25}
{'7b': 16, '8': 10, '1a': 22, '1b': 21, '7a': 15}
84
```

2. В Python ключи словаря можно получить, воспользовавшись встроенной функцией `list()`. Присвойте одной переменной произвольный словарь, второй – список его ключей, полученный из `list()`, третьей – результат выполнения словарного метода `keys()`. После этого внесите изменения в словарь, например, добавив в него еще одну запись. Выведите значения переменных на экран. Сделайте вывод об особенностях объектов типа `dict_keys`.

Решение:

```
>>> a = {1: 10, 2: 20}
>>> b = list(a)
>>> c = a.keys()
>>> b
[1, 2]
>>> c
dict_keys([1, 2])
>>> a[3] = 30
>>> a
{1: 10, 2: 20, 3: 30}
>>> b
[1, 2]
>>> c
dict_keys([1, 2, 3])
```

Экземпляры `dict_keys`, а также `dict_values` и `dict_items`, остаются связанными со своим словарем. Поэтому когда словарь меняется, изменяется и их содержимое. Другими словами, они дают нам динамическое представление содержимого словаря.

Урок 24. Функция open. Чтение и запись текстовых файлов в Python

1. Создайте файл *data.txt* по образцу урока. Напишите программу, которая открывает этот файл на чтение, построчно считывает из него данные и записывает строки в другой файл (*dataRu.txt*), заменяя английские числительные русскими, которые содержатся в списке (`["один", "два", "три", "четыре", "пять"]`), определенном до открытия файлов.

Решение:

Файл *data.txt*:

```
one - 1 - I
two - 2 - II
three - 3 - III
four - 4 - IV
five - 5 - V
```

Программа:

```
ru = ["один", "два", "три", "четыре", "пять"]
i = 0  # индексы для списка

f1 = open("data.txt")
f2 = open("dataRu.txt", 'w')

# Цикл "перебирает" строки первого файла.
for s in f1:
    line = s.split(' - ')  # текущая строка преобразуется в список
    line[0] = ru[i]  # замена первого элемента в строке на слово из ru
    i += 1  # переход к следующему слову в ru
    line = ' - '.join(line)  # список в строку
    f2.write(line)  # запись строки во второй файл

f1.close()
f2.close()
```

Результат (файл *dataRu.txt*):

```
один - 1 - I
два - 2 - II
три - 3 - III
четыре - 4 - IV
пять - 5 - V
```

Другой способ решения:

```
ru = ["один", "два", "три", "четыре", "пять"]

f1 = open("data.txt")
f2 = open("dataRu.txt", 'w')

# Цикл перебирает слова из списка.
for word in ru:
    current_line = f1.readline().split(' - ')
    current_line[0] = word
    f2.write(' - '.join(current_line))
```

```
f1.close()
f2.close()
```

2. Создайте файл *nums.txt*, содержащий несколько чисел, записанных через пробел. Напишите программу, которая подсчитывает и выводит на экран общую сумму чисел, хранящихся в этом файле.

Решение:

```
f = open("nums.txt")
n = f.read() # считываются все данные в одну строку
n = n.split() # строка в список, элементы списка имеют строковый тип
for i in range(len(n)):
    n[i] = int(n[i]) # элементы списка преобразуются в числа
print(sum(n))
```

Урок 25. Особенности работы операторов and и or

1. Напишите одну строчку кода, в котором у пользователя запрашивается строка. Если он ничего не вводит и нажимает Enter, то соответствующей переменной присваивается вами заданная строка по-умолчанию.

Решение:

```
>>> a = input() or 'unknown'
lip
>>> a
'lip'
>>> a = input() or 'unknown'

>>> a
'unknown'
```

2. Напишите программу, в которой проверяется, содержится ли введенное значение в списке. Если оно не содержится, происходит выход из программы. Проверку и выход запишите в одну строку.

Решение:

```
>>> a = [1, 2, 3]
>>> b = int(input())
4
>>> b in a or quit()
pl@comp:~$
```

```
>>> a = [1, 2, 3]
>>> b = int(input())
3
```

```
>>> b in a or quit()
True
```

Урок 26. Множества

Множество как структура данных подходит для хранения списка тегов. Если пользователь вводит новый тег, он добавляется во множество. Если вводит тег, который уже есть во множестве, дублирования не произойдет.

Напишите программу, в которой пользователь вводит слова. Если слово до этого не вводилось, оно добавляется ко множеству. Если слово уже было введено ранее, то пользователь получает сообщение "уже есть".

Вариант решения:

```
print('To stop input "stop"')

tags = set()

phrase = ''
while phrase != 'stop':
    phrase = input()
    if phrase in tags:
        print("Tags have this phrase")
        continue
    if phrase != 'stop':
        tags.add(phrase)

print(tags)
```

Пустое множество нельзя создать, используя пустые фигурные скобки, так как в этом случае создается пустой словарь.

Урок 27. Матрицы

1. Заполните матрицу случайными положительными и отрицательными числами. Выведите ее на экран так, чтобы вместо отрицательных чисел стояли прочерки, то есть вывод должен выглядеть примерно так:

```
  2  -  5  6
  -  0  -  -
  5  4  -  -
```

Решение:

```
from random import randint
```

```
N = 3
M = 4

a = []
for i in range(N):
    b = []
    for j in range(M):
        b.append(randint(-9, 9))
    a.append(b)

print(a, end='\n\n')

for i in range(N):
    for j in range(M):
        if a[i][j] >= 0:
            print("%3d" % a[i][j], end='')
        else:
            print("  -", end='')
    print()
```

2. Найдите сумму элементов каждого столбца матрицы.

Решение:

```
from random import randint
N = 3
M = 4

a = []
for i in range(N):
    b = []
    for j in range(M):
        b.append(randint(1, 9))
    a.append(b)

for i in range(N):
    for j in range(M):
        print("%3d" % a[i][j], end='')
    print()
print()

for j in range(M):
    s = 0  # сумма элементов очередного столбца
    for i in range(N):
        s += a[i][j]
    print("%3d" % s, end='')
```

```
print()
```

В данном варианте решения при нахождении суммы столбцов внешний цикл перебирает столбцы, а внутренний – элементы внутри столбца. То есть во внутреннем цикле у нас первый индекс меняется, а второй, который обозначает текущий столбец, остается постоянным. При построчном переборе это наоборот.

Однако можно было бы выполнить и привычный построчный перебор, предварительно создав массив для хранения сумм столбцов:

```
s = [0] * M
for i in range(N):
    for j in range(M):
        s[j] += a[i][j]

print(s)
```

Урок 28. Генераторы списков

1. Перепишите три последних примера из урока с помощью обычного цикла `for` (то есть без использования генераторов списков).

Решения:

```
>>> s1 = 'abcd'
>>> s2 = '01'
>>> s3 = []
>>> for i in s1:
...     for j in s2:
...             s3.append(i+j)
...
>>> s3
['a0', 'a1', 'b0', 'b1', 'c0', 'c1', 'd0', 'd1']
```

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> b = []
>>> for i in a:
...     if i % 2 == 0:
...             b.append(i)
...
>>> b
[2, 4, 6, 8, 10]
```

```
>>> a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = []
>>> for i in a:
...     for j in b:
...             if i*j <= 10:
...                     c.append((i, j))
...
>>> c
[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5)]
```

2. Напишите генератор списка, который заполняет список данными, которые вводит пользователь. Другой генератор списка должен преобразовывать данные в числа. Если какой-либо элемент первого списка нельзя преобразовать в число, то во второй список этот элемент попадать не должен.

Решение:

```
>>> a = [input() for i in range(5)]
22
d
5
34
vv
>>> a
['22', 'd', '5', '34', 'vv']
>>> b = [int(i) for i in a if i.isdigit()]
>>> b
[22, 5, 34]
```

Урок 29. Lambda-выражения

1. Создайте список лямбда-выражений. Одно складывает аргументы, другое – вычитает, третье – умножает, четвертое – делит. Запросите у пользователя два числа и в цикле "прогоните" их через все lambda-функции списка.

```
m = [
    (lambda x, y: x+y),
    (lambda x, y: x-y),
    (lambda x, y: x*y),
    (lambda x, y: round(x/y, 1))
]
a = int(input())
b = int(input())
for i in m:
```

```
    print(i(a, b))
```

2. Дан список строк. Отсортируйте его сначала по последним буквам слов, потом по длине слов.

```
>>> a = ['apple', 'banana', 'orange', 'tomato', 'soup']
>>> a.sort(key=lambda i: i[-1])
>>> a
['banana', 'apple', 'orange', 'tomato', 'soup']
>>> a.sort(key=len)
a
>>> ['soup', 'apple', 'banana', 'orange', 'tomato']
>>> a.sort(key=lambda i: len(i))
```

В примере показано, что излишне писать lambda-функцию для измерения длины строки (хотя ошибки не будет), так как есть встроенная функция `len()`.

Урок 30. Сортировка списков

Доработайте приведенную в уроке программу про юных спортсменов таким образом, чтобы пользователь выбирал не только столбец сортировки, также мог указать тип сортировки – по возрастанию или убыванию.

```
a = [['Петя', 10, 130, 35], ['Вася', 11, 135, 39],
     ['Женя', 9, 120, 33], ['Дима', 10, 128, 30]]

n = input('Сортировать по имени(0), '
          'возрасту(1), росту(2), весу(3): ')
t = int(input('По возрастанию(0), по убыванию(1): '))

b = sorted(a, key=lambda item: item[int(n)], reverse=bool(t))

for i in b:
    print("%7s %3d %4d %3d" % (i[0], i[1], i[2], i[3]))
```

Урок 31. Фильтрация списков

Пользователь вводит два натуральных числа, первое больше второго. Используя функцию **`filter`**, вывести на экран все натуральные числа, кратные второму числу и не превышающие первое. Например, были введены 10 и 3. Вывод: 3, 6, 9.

```
n = int(input('Предел '))
a = int(input('Кратны числу '))
```

```
r = filter(lambda i: i % a == 0, range(1, n))
print(list(r))
```

Пример выполнения:

```
Предел 100
Кратны числу 14
[14, 28, 42, 56, 70, 84, 98]
```

Вместо функции `range`, создающей итерируемый объект с числами от 1 до *n*, можно было бы предварительно заполнить список:

```
d = [i for i in range(1, n)]
r = filter(lambda i: i % a == 0, d)
```

Урок 32. Функция zip

Вычислите суммы столбцов матрицы. Используйте в программе функцию `zip`.

```
a = [(1, 2, 3), (4, 5, 6),
     (7, 8, 9), (3, 2, 1)]

for i in a:
    for j in i:
        print("%3d" % j, end=' ')
    print()

for i in zip(*a):
    print("%3d" % sum(i), end=' ')
print()
```

Объект, который возвращает выражение `zip(*a)`, генерирует кортежи, состоящие из элементов столбцов исходной матрицы. Если бы данный объект был преобразован в список, мы бы получили транспонированную матрицу (столбцы стали бы строками). Поскольку бывшие столбцы теперь строки, перебираем их в цикле `for`, применив к каждому функцию `sum`.

Результат выполнения:

```
  1   2   3
  4   5   6
  7   8   9
  3   2   1
 15  17  19
```