Основы программирования на Python

Вводный курс

Версия 2

С. Шапошникова
(plustilino)

Лаборатория юного линуксоида
http://younglinux.info, 2011

Пояснительная записка

Курс по информатике "Основы программирования на Python" представляет собой вводный курс по программированию, дающий представление о базовых понятиях структурного программирования (данных, операциях, переменных, ветвлениях в программе, циклах и функциях).

Выбор Python обусловлен тем, что это язык, обладающий рядом преимуществ перед другими языками для начинающих изучать программирование: ясность кода, быстрота реализации.

Курс рассчитан примерно на 15 часов.

Данный курс не является пособием по языку программирования Python.

Цели и задачи курса

Основной целью курса "Основы программирования на Python" является формирование базовых понятий структурного программирования, развитие логики обучающихся.

Программа курса

Поверхностное представление о языках программирования и их историческом развитии, способах трансляции программного кода. Типы данных (целые числа, числа с плавающей точкой, строки) и структуры данных (строки, списки, словари), переменные, выражения, ветвления (**if, if-else, if-elif-else**) и циклы (**while, for**). Ввод и вывод данных. Понятие о функции, локальных и глобальных переменных.

Авторские права

Материалы, составляющие данное пособие, распространяются на условии лицензии GNU FDL. Учебник не содержит неизменяемых разделов. Автор пособия указан на первой странице обложки. Встречающиеся в книге названия могут являться торговыми марками соответствующих владельцев.

Курс "Основы программирования на Python" первоначально публиковался на сайте http://younglinux.info с 2009 года.

Содержание

Урок 1.
История языков программирования. Компиляция и интерпретация

Программа. Язык программирования

Программу можно представить как набор последовательных команд (алгоритм) для объекта (исполнителя), который должен их выполнить для достижения определенной цели. Так условно можно запрограммировать и человека, если составить ему инструкцию "как приготовить оладьи", а он начнет четко ее исполнять. При этом инструкция (программа) для человека будет написана на так называемом естественном языке (русском, английском или др.).

Обычно принято программировать не людей, а вычислительные машины, используя при этом специальные языки. Использование особых языков вызвано тем, что машины не в состоянии понимать наши (человеческие) языки. Для "инструктирования" машин предназначены разнообразные *языки программирования*, которые характеризуются синтаксической однозначностью (например, в них нельзя менять местами определенные слова) и ограниченностью (строго определенный набор слов и символов).

Основные этапы развития языков программирования

Первые программы писались на *машинном языке*, т.к. для ЭВМ того времени еще не существовало развитого программного обеспечения, а машинный язык — это единственный способ взаимодействия с аппаратным обеспечением компьютера. Каждую команду машинного языка напрямую выполняет то или иное электронное устройство. Данные и команды программисты записывали в цифровом виде (например, в шестнадцатеричной или двоичной системах счисления). Понять программу на таком языке очень сложно; кроме того, даже небольшая программа получалась состоящей 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 и не цифрами привело к появлению *ассемблеров*. Это языки, в которых вместо численного обозначения команд и областей памяти используются словесно-буквенные.

При этом появляется проблема: машина не в состоянии понять наборы букв. 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 — низкоуровневые интерфейсы операционных систем, части драйверов и др.).

После ассемблеров наступил рассвет языков так называемого высокого уровня. Для этих языков потребовалось разрабатывать более сложные трансляторы, т.к. 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. Т.е., написав один раз программу, программист мог выполнить ее на любой машине.

Следующим значимым шагом было появление объектно-ориентированных языков программирования. С помощью таких языков программист как бы оперирует виртуальными объектами. На сегодняшний день, реализация больших и сложных проектов осуществляется в основном с помощью объектно-ориентированного программирования.

Разнообразия языков программирования

На сегодняшний день существует огромное 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 (т.е. языки программирования).

Многие программисты старались в прошлом и стараются сейчас придумать свой язык программирования, обладающий теми или иными преимуществами. Хотя подавляющее большинство программистов в настоящее время тратят огромное количество времени на изучение уже существующего арсенала инструментов.

Все существующее многообразие языков можно условно классифицировать по разным критериям. Например, по типу решаемых задач (языки системного или прикладного назначения, языки для web-разработки и др.).

Трансляция

Ранее было сказано, что для перевода кода с одного языка программирования (например, языка высокого уровня) на другой (например, машинный язык) требуется специальная программа — транслятор.

Механизм этого перевода весьма сложен, при этом выделяют 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.

При интерпретации выполнение кода происходит последовательно (можно сказать, строка за строкой). Операционная система взаимодействует с интерпретатором, а не исходным кодом.

Выполнение откомпилированной программы происходит быстрее, т.к. она представляет собой готовый машинный код. Однако на современных компьютерах снижение скорости выполнения при интерпретации обычно не заметно.

Урок 2.
Знакомство с Python и средами программирования

История

Язык программирования Python был создан примерно в 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 уже целое сообщество программистов присоединилось к его улучшению.

Python активно совершенствуется и в настоящее время. Часто выходят его новые версии. Официальный сайт http://python.org.

Особенности

Python – это интерпретируемый язык программирования: исходный код частями преобразуется в машинный в процессе выполнения специальной программой — интерпретатором.

Python характеризуется ясным синтаксисом. Читать код на этом языке программирования достаточно легко, т.к. в нем мало вспомогательных элементов, а правила языка заставляют программистов делать отступы. Понятно, что хорошо оформленный текст с малым количеством отвлекающих элементов читать и понимать легче.

Python – это полноценный, можно сказать универсальный, язык программирования. Он поддерживает объектно-ориентированное программирование (на самом деле он и разрабатывался как объектно-ориентированный язык).

Также Python распространяется свободно на основании лицензии подобной GNU General Public License.

Дзэн Питона

Если интерпретатору Питона дать команду `import this` (импортировать "сам объект"), то выведется так называемый "Дзен Питона", иллюстрирующий идеологию и особенности данного языка. Глубокое понимание этого дзена приходит тем, кто сможет освоить язык Python в полной мере и приобретет опыт практического программирования.

№	Фраза	Перевод
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

Интерактивный режим

В основном интерпретатор выполняет команды построчно: пишешь строку, нажимаешь Enter, интерпретатор выполняет ее, наблюдаешь результат.

Это очень удобно, когда человек только изучает программирование или тестирует какую-нибудь небольшую часть кода. Ведь если работать на компилируемом языке, то пришлось бы сначала написать код на исходном языке программирования, затем скомпилировать и уж потом запустить исполняемый файл на выполнение.

Работать в интерактивном режиме в ОС Linux можно в консоли. Для этого следует выполнить команду python. Запустится интерпретатор, где сначала выведется информация об интерпретаторе. Далее, последует приглашение к вводу (>>>).

Запустите интерпретатор Питона.

Поскольку никаких команд мы пока не знаем, то будем использовать Питон как калькулятор (возможности языка это позволяют).

```
2 + 5
3 * (5 - 8)
2.4 + 3.0 / 2
```
и т.д.

Наберите подобные примеры в интерактивном режиме (в конце каждого нажимайте Enter).

Ответ выдается сразу после нажатия Enter (завершения ввода команды).

Бывает, что в процессе ввода была допущена ошибка или требуется повторить ранее используемую команду. Чтобы не писать строку сначала, в консоли можно прокручивать список команд, используя для этого стрелки на клавиатуре.

Другой вариант работы в интерактивном режиме — это работа в среде разработки IDLE, у которой есть интерактивный режим работы. В отличие от консольного варианта здесь можно наблюдать подсветку синтаксиса (в зависимости от значения синтаксической единицы она выделяется определенным цветом). Прокручивать список ранее введенных команд можно с помощью комбинаций Alt+N, Alt+P.

Запустите IDLE. Попробуйте решать математические примеры здесь.

Создание скриптов

Несмотря на удобства интерактивного режима работы при написании программ на Питоне, обычно требуется сохранять исходный программный код для последующего использования. В таком случае подготавливаются файлы, которые передаются затем интерпретатору на исполнение. По отношению к интерпретируемым языкам программирования часто исходный код называют скриптом. Файлы с кодом на Python обычно имеют расширение *py*.

Подготовить скрипты можно в той же среде IDLE. Для этого, после запуска программы в меню следует выбрать команду **File → New Window** (Crtl + N), откроется новое окно. Затем желательно сразу сохранить файл (не забываем про расширение *py*). После того как код будет подготовлен, снова сохраните файл (чтобы обновить сохранение). Ну и наконец, можно запустить скрипт, выполнив команду меню **Run → Run Module** (F5). После этого в первом окне появится результат выполнения кода. (Примечание: если набирать код, не сохранив файл в начале, то подсветка синтаксиса будет отсутствовать.)

Подготовьте скрипт (с примерами). Запустите его на выполнение.

На самом деле скрипты можно готовить в любом текстовом редакторе (желательно, чтобы он поддерживал подсветку синтаксиса языка Python). Кроме того, существуют специальные программы для разработки.

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

В консоли передайте интерпретатору Питона на выполнение подготовленный файл.

Кроме того, существует возможность настроить выполнение скриптов с помощью двойного клика по файлу (в Windows данная возможность присутствует изначально).

Урок 3.
Типы данных в программировании. Определение переменной

Данные и их типы

Можно заметить, что все, что мы делаем, мы делаем над чем-то — какими-то предметами или объектами. Мы меняем свойства объектов и их возможности. Программы для компьютеров также манипулируют какими-то объектами (назовем их пока *данными*).

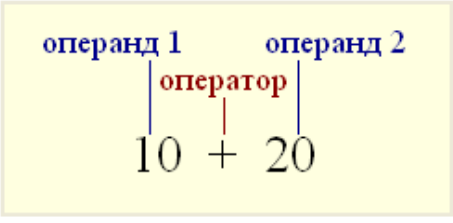
Очевидно, данные бывают разными. Очень часто компьютерной программе приходится работать с числами и строками. Например, на прошлом уроке мы "манипулировали" числами, выполняя над ними арифметические операции. Можно сказать, что операция сложения выполняла изменение первого числа на величину второго, или умножение увеличивало одно число в количество раз, соответствующее второму.

Числа в свою очередь также бывают разными: целыми, дробными, могут иметь огромное значение или очень длинную дробную часть. При знакомстве с языком программирования Python мы точно столкнемся с тремя типами данных:

- *целые числа (integer)* – положительные и отрицательные целые числа, а также 0 (например, 4, 687, -45, 0).
- *числа с плавающей точкой (float point)* – дробные числа (например, 1.45, -3.789654, 0.00453). Примечание: разделителем целой и дробной части служит точка, а не запятая.
- *строки (string)* — набор символов, заключенных в кавычки (например, "ball", "What is your name?", 'dkfjUUv', '6589'). Примечание: кавычки в Python могут быть одинарными или двойными.

Операции

Можно сказать, что *операция* — это выполнение каких-нибудь действий над данными (*операндами*). Для выполнения конкретных действий требуются специальные инструменты — *операторы*.



Например, символ "+" по отношению к числам выполняет операцию сложения, а по

отношению к строкам — конкатенацию (соединение). Парный знак ** возводит первое число в степень второго.

Выражение	**Результат выполнения**
34.907 + 320.65	355.55699999999996
"Hi, " + "world :)"	'Hi, world :)'
"Hi, " * 10	'Hi, Hi, Hi, Hi, Hi, Hi, Hi, Hi, Hi, Hi, '

Изменение типа данных

Что будет, если мы попытаемся выполнить в одном выражении операцию над разными типами данным? Например, сложить целое и дробное число, число и строку. Однозначный ответ дать нельзя: так, при складывании целого числа и числа с плавающей точкой, получается число с плавающей точкой, а если попытаться сложить любое число и строку, то интерпретатор Python выдаст ошибку.

Выражение	**Результат выполнения**
1 + 0.65	1.6499999999999999
"Hi, " + 15	О ш и б к а

Однако, бывают случаи, когда программа получает данные в виде строк, а оперировать должна числами (или наоборот). В таком случае используются специальные функции (особые операторы), позволяющие преобразовать один тип данных в другой. Так функция **int()** преобразует переданную ей строку (или число с плавающей точкой) в целое, функция **str()** преобразует переданный ей аргумент в строку, **float()** - в дробное число.

Выражение	**Результат выполнения**
int ("56")	56
int (4.03)	4
int ("comp 486")	О ш и б к а
str (56)	'56'
str (4.03)	'4.03'
float (56)	56.0
float ("56")	56.0

Переменные

Данные хранятся в ячейках памяти компьютера. Когда мы вводим число, оно помещается в память. Но как узнать, куда именно? Как в последствии обращаться к этим данными? Раньше, при написании программ на машинном языке, обращение к ячейкам памяти осуществляли с помощью указания регистров. Но уже с появлением ассемблеров, при обращении к данным стали использовать так называемые *переменные*. Механизм связи между переменными и данными может различаться в зависимости от языка программирования и типа данных. Пока достаточно запомнить,

что данные связываются с каким-либо именем и в дальнейшем обращение к ним возможно по этому имени.

В программе на языке Python связь между данными и переменными устанавливается с помощью знака =. Такая операция называется присваиванием. Например, выражение `sq = 4` означает, что на объект (данные) в определенной области памяти ссылается имя sq и обращаться к ним теперь следует по этому имени.

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 (_).

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

Пример работы с переменными в интерактивном режиме:

```
>>> apples = 100
>>> eat_day = 5
>>> day = 7
>>> apples = apples - eat_day * day
>>> apples
65
>>>
```

Практическая работа

1. Переменной var_int присвойте значение 10, var_float - значение 8.4, var_str - "No".
2. Измените значение, хранимое в переменной var_int, увеличив его в 3.5 раза, результат свяжите с переменной big_int.
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 big_int 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.

Результат выполнения практической работы

```
>>> var_int = 10
>>> var_float = 8.4
>>> var_str = "No"
>>> big_int = var_int * 3.5
>>> var_float = var_float — 1
>>> var_int / var_float
1.3513513513513513
>>> big_int / var_float
4.72972972972973
>>> var_str = var_str * 2 + "Yes" * 3
>>> var_int
10
>>> var_float
7.4
>>> big_int
35.0
>>> var_str
'NoNoYesYesYes'
>>>
```

Вопросы

1. Какие типы данных вы знаете? Опишите их.
2. Можно ли преобразовать дробное число в целое? целое в дробное? В каких случаях можно строку преобразовать в число?
3. Приведите примеры операций. Для чего предназначена операция присвоения?
4. Какие существуют правила и рекомендации для именования переменных?

Урок 4.
Логические выражения

Логического выражения и логический тип данных

Часто в реальной жизни мы соглашаемся или отрицаем то или иное утверждение, событие, факт. Например, "Сумма чисел 3 и 5 больше 7" является правдивым утверждением, а "Сумма чисел 3 и 5 меньше 7" - ложным. Можно заметить, что с точки зрения логики подобные фразы предполагают только два результата: "Да" (правда) и "Нет" (ложь). Подобное используется в программировании: если результатом вычисления выражения может быть лишь истина или ложь, то такое выражение называется логическим.

На прошлом уроке были описаны три типа данных: целые, дробные числа, а также строки. Также выделяют *логический тип данных*. У этого типа всего два возможных значения: **True** (правда) — 1 и **False** (ложь) — 0. Только эти значения могут быть результатом логических выражений.

Логические операторы

Говоря на естественном языке (например, русском) мы обозначаем сравнение словами "равно", "больше", "меньше". В языках программирования используются специальные знаки, подобные тем, которые используются в математических выражениях: > (больше), < (меньше), >= (больше или равно), <= (меньше или равно).

Новыми для вас могут оказаться обозначение равенства: == (два знака "равно"); а также неравенства !=. Здесь следует обратить внимание на следующее: не путайте операцию присваивания, обозначаемую в языке Python одиночным знаком "равно", и операцию сравнения (два знака "равно"). Присваивание и сравнение — совершенно разные операции.

Примеры работы с логическими выражениями на языке программирования Python (после # написаны комментарии):

```
x = 12 – 5 # это не логическая операция,
# а операция присваивания переменной x результата выражения 12 — 5
x == 4 # x равен 4
x == 7 # x равен 7
x != 7 # x не равен 7
x != 4 # x не равен 4
x  > 5 # x больше 5
x  < 5 # x меньше 5
x >= 6 # x больше или равен 6
x <= 6 # x меньше или равен 6
```

Определите устно результаты выполнения операций, приведенных в примере выше. Проверьте правильность ваших предположений, выполнив данные выражения с помощью интерпретатора языка Python.

Сложные логические выражения

Логические выражения типа `verymuch >= 1023` является простым. Однако, на практике не редко используются более сложные. Может понадобиться получить ответа "Да" или "Нет" в зависимости от результата выполнения двух простых выражений. Например, "на улице идет снег или дождь", "переменная new больше 12 и меньше 20" и т.п.

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

Чтобы получить истину (**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** (ложь), то и все сложное выражение будет ложным.

Чтобы получить истину (**True**) при использовании оператора **or**, необходимо, чтобы результаты хотя бы одного простого выражения, входящего в состав сложного, был истинным. В случае оператора **or** сложное выражение становится ложным лишь тогда, когда ложны все составляющие его простые выражения.

Примеры работы со сложными логическими выражениями на языке программирования Python (после # написаны комментарии):

```
x = 8
y = 13
x == 8 and y < 15 #  x равен 8 и y меньше 15
x > 8 and y < 15 #  x больше 8 и y меньше 15
x != 0 or y >15 #  x не равен 0 или y меньше 15
x < 0 or y >15 #  x меньше 0 или y меньше 15
```

Определите устно результаты выполнения операций, приведенных в примере выше. Проверьте правильность ваших предположений, выполнив данные выражения с помощью интерпретатора языка Python.

Практическая работа

1. Присвойте двум переменным любые числовые значения.
2. Составьте четыре сложных логических выражения с помощью оператора **and**, два из которых должны давать истину, а два других - ложь.
3. Аналогично выполните п. 2, но уже используя оператор **or**.
4. Попробуйте использовать в сложных логических выражениях работу с переменными строкового типа.

Примерный результат выполнения практической работы

```
>>> num1 = 34
>>> num2 = 8.5
>>> num1 > 12 and num2 != 12
```

```
True
>>> num1 == 34 and num2 >= 8
True
>>> num1 != 34 and num2 != 12
False
>>> num1 <= 12 and num1 == 0
False
>>> num1 != 34 or num2 != 12
True
>>> num1 < 1 or num2 > 9.6
False
>>> str1 = "a"
>>> str2 = "b"
>>> str1 < "c" and str2 != "a"
True
>>>
```

Урок 5.
Условный оператор. Инструкция if

Ход выполнения программы может быть линейным, т.е. таким, когда выражения выполняются, начиная с первого и заканчивая последним, по порядку, не пропуская ни одной строки кода. Но чаще бывает совсем не так. При выполнении программного кода некоторые его участки могут быть пропущены. Чтобы лучше понять почему, проведем аналогию с реальной жизнью. Допустим, человек живет по расписанию (можно сказать, расписание — это своеобразный "программный код", который следует выполнить). В его расписании в 18.00 стоит поход в бассейн. Однако человеку поступает информация, что воду слили, и бассейн не работает. Вполне логично отменить свое занятие по плаванию. Т.е. одним из условий посещения бассейна должно быть его функционирование, иначе должны выполняться другие действия.

Похожая нелинейность действий может быть предусмотрена и в компьютерной программе. Например, часть кода должна выполняться лишь при определенном значении конкретной переменной. Обычно в языках программирования используется приблизительно такая конструкция:

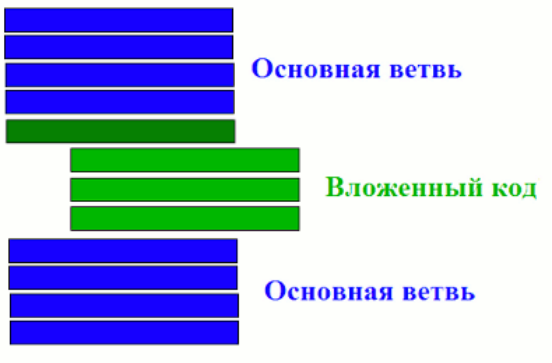
```
if  a логический_оператор b :
        выражения, выполняемые при результате True
        в логическом выражении
```

Пример ее реализации на языке программирования Python:

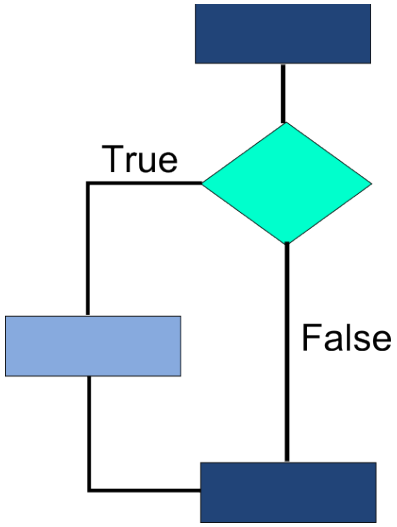
```
if numbig < 100: # если значение переменной numbig меньше 100, то ...
        c = a**b    # возвести значение переменной a в степень b,
                    # результат присвоить c.
```

Первая строка конструкции **if** — это заголовок, в котором проверяется условие выполнения строк кода после двоеточия (тела конструкции). В примере выше тело содержит всего лишь одно выражение, однако чаще их бывает куда больше.

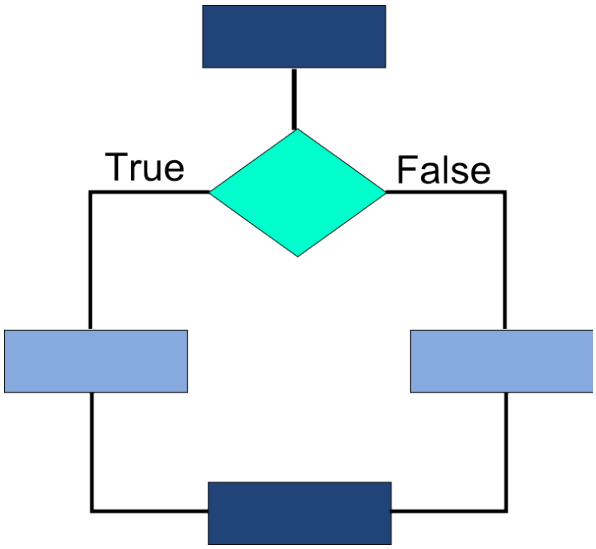
Про Python говорят, что это язык программирования с достаточно ясным и легко читаемым кодом. Это связано с тем, что в нем сведены к минимуму вспомогательные элементы (скобки, точка с запятой), а для разделения синтаксических конструкций используются отступы от начала строки. Учитывая это, в конструкции **if** код, который выполняется при соблюдении условия, должен обязательно иметь отступ вправо. Остальной код (основная программа) должен иметь тот же отступ, что и слово **if**. Обычно отступ делается с помощью клавиши Tab.



Можно изобразить блок-схему программы, содержащей инструкцию **if**, в таком виде:



Встречается и более сложная форма ветвления: **if–else**. Если условие при инструкции **if** оказывается ложным, то выполняется блок кода при инструкции **else**.



Пример кода с веткой **else** на языке программирования Python:

```
print "Привет"
tovar1 = 50
tovar2 = 32
if tovar1+ tovar2 > 99 :
        print "Сумма не достаточна"
else:
```

```
        print "Чек оплачен"
print "Пока"
```

Практическая работа

1. Напишите программный код, в котором в случае, если значение некой переменной больше 0, выводилось бы специальное сообщение (используйте функцию **print**). Один раз выполните программу при значении переменной больше 0, второй раз — меньше 0.
2. Усовершенствуйте предыдущий код с помощью ветки **else** так, чтобы в зависимости от значения переменной, выводилась либо 1, либо -1.
3. Самостоятельно придумайте программу, в которой бы использовалась инструкция **if** (желательно с веткой **else**). Вложенный код должен содержать не менее трех выражений.

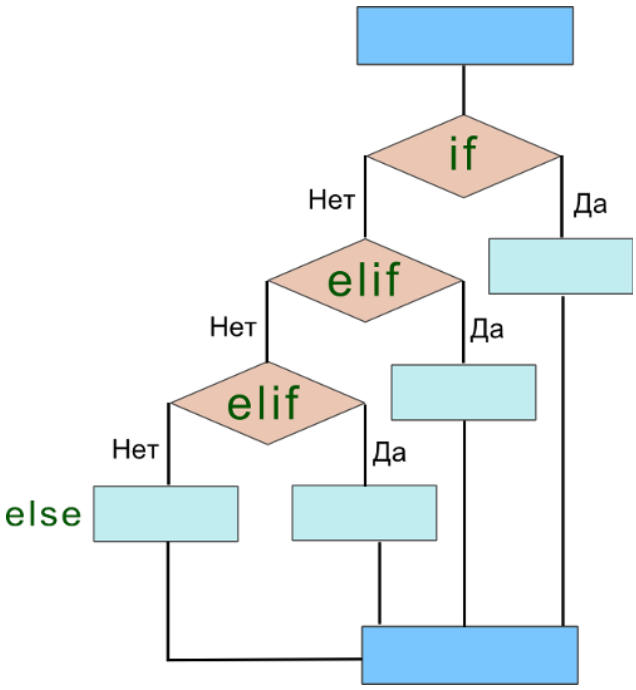
Урок 6.
Множественное ветвление

Логика выполняющейся программы может быть сложнее, чем выбор одной из двух ветвей. Например, в зависимости от значения той или иной переменной, может выполняться одна из трех (или более) ветвей программы.

Как организовать такое множественное ветвление? Наверное, можно использовать несколько инструкций **if**: сначала проверяется условное выражение в первой инструкции **if** (если оно возвращает истину, то будет выполняться вложенный в нее блок кода), затем во второй инструкции **if** и т.д. Однако при таком подходе проверка последующих инструкций будет продолжаться даже тогда, когда первое условие было истинным, и блок кода при данной ветке был выполнен. Проверка последующих условий может оказаться бессмысленной.

Обычно такую проблему можно решить с помощью вложенных конструкций **if-else**. Однако при этом часто появляется проблема правильной трактовки кода: непонятно, к какому **if** относится **else** (хотя в Python такая путаница не возможна из-за обязательных отступов).

С другой стороны, в ряде языков программирования, в том числе и Python, предусмотрено специальное расширение инструкции **if**, позволяющее направить поток выполнения программы по одной из множества ветвей. Данная расширенная инструкция, помимо необязательной части **else**, содержит ряд ветвей **elif** (сокращение от "else if" - "еще если") и выглядит примерно так, как показано на блок-схеме. Частей **elif** может быть сколь угодно много (в пределах разумного, конечно).



В отличии от использования множества одиночных инструкций **if**, инструкция **if-elif-else** прекращает просмотр последующих ветвей, как только логическое выражение в

текущей ветке вернет **true**. Например, если выражение при **if** (первая ветка) будет истинным, то после выполнения вложенного блока выражений, программа вернется в основную ветку.

Примеры скриптов с использованием инструкции **if-elif-else** на языке программирования Python:

```
x = -10

if x > 0:
        print 1
elif x < 0:
        print -1
else:
        print 0


result = "no result"
num1 = 3

if num1 == 0:
        result = 0
elif num1==1:
        result = 1
elif num1==2:
        result = 2
elif num1==3:
        result = 3
elif num1==4:
        result = 4
elif num1==5:
        result = 5
else:
        print "Error"

print result
```

В какой момент прекратиться выполнение инструкции **if-elif-else** в примерах выше. При каком значении переменной могла сработать ветка **else**?

Практическая работа

1. Напишите программу по следующему описанию:
    a. двум переменным присваиваются числовые значения;
    b. если значение первой переменной больше второй, то найти разницу значений переменных (вычесть из первой вторую), результат связать с третьей переменной;
    c. если первая переменная имеет меньшее значение, чем вторая, то третью переменную связать с результатом суммы значений двух первых переменных;
    d. во всех остальных случаях, присвоить третьей переменной значение

первой переменной;

e. вывести значение третьей переменной на экран.

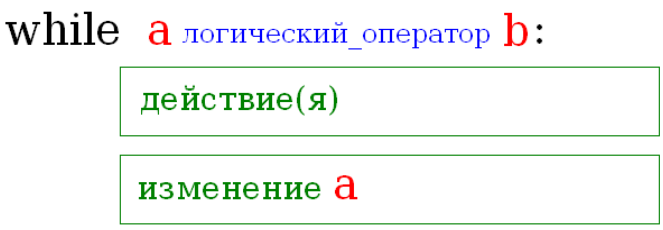
2. Придумайте программу, в которой бы использовалась инструкция **if-elif-else**. Количество ветвей должно быть как минимум четыре.

Урок 7.
Цикл while

Циклы — это инструкции, выполняющие одну и туже последовательность действий, пока действует заданное условие.

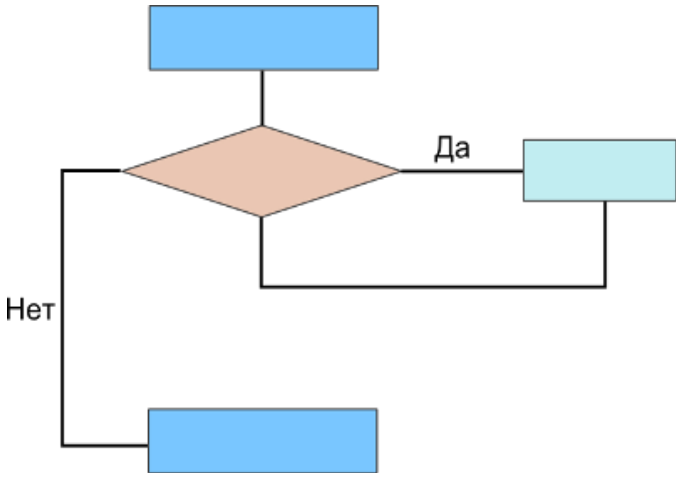
В реальной жизни мы довольно часто сталкиваемся с циклами. Например, ходьба человека — вполне циклическое явление: шаг левой, шаг правой, снова левой-правой и т.д., пока не будет достигнута определенная цель (например, школа или магазин). В компьютерных программах наряду с инструкциями ветвлениями (т.е. выбором пути действия) также существуют инструкции циклов (повторения действия). Если бы инструкций цикла не существовало, пришлось бы много раз вставлять в программу один и тот же код подряд столько раз, сколько нужно выполнить одинаковую последовательность действий.

Универсальным организатором цикла в языке программирования Python (как и во многих других языках) является конструкция **while**. Слово "while" с английского языка переводится как "пока" ("пока логическое выражение возвращает истину, выполнять определенные операции"). Конструкцию **while** на языке Python можно описать следующей схемой:



Эта схема приблизительна, т.к. логическое выражение в заголовке цикла **while** может быть более сложным, а изменяться может переменная (или выражение) b.

Может возникнуть вопрос: "Зачем изменять a или b?". Когда выполнение программного кода доходит до цикла **while**, выполняется логическое выражение в заголовке, и, если было получено **True** (истина), выполняются вложенные выражения. После поток выполнения программы снова возвращается в заголовок цикла **while**, и снова проверяется условие. Если условие никогда не будет ложным, то не будет причин остановки цикла и программа *зациклится*. Чтобы этого не произошло, необходимо предусмотреть возможность выхода из цикла — ложность выражения в заголовке. Таким образом, изменяя значение переменной в теле цикла, можно довести логическое выражение до ложности.



Эту изменяемую переменную, которая используется в заголовке цикла **while**, обычно называют *счетчиком*. Как и всякой переменной ей можно давать произвольные имена, однако очень часто используют буквы i и j. Простейший цикл на языке программирования Python может выглядеть так:

```
str1 = "+"
i = 0
while i < 10:
        print (str1)
        i = i + 1
```

В последней строчке кода происходит увеличение значения переменной i на единицу, поэтому с каждым оборотом цикла ее значение увеличивается. Когда будет достигнуто число 10, логическое выражение i < 10 даст ложный результат, выполнение тела цикла будет прекращено, а поток выполнения программы перейдет на команды следующие за всей конструкцией цикла. Результатом выполнения скрипта приведенного выше является вывод на экран десяти знаков + в столбик. Если увеличивать счетчик в теле цикла не на единицу, а на 2, то будет выведено только пять знаков, т.к цикл сделает лишь пять оборотов.

Более сложный пример с использованием цикла:

```
fib1 = 0
fib2 = 1
print (fib1)
print (fib2)
n = 10
i = 0
while i < n:
        fib_sum = fib1 + fib2
        print (fib_sum)
        fib1 = fib2
        fib2 = fib_sum
        i = i + 1
```

Этот пример выводит *числа Фибоначчи* — ряд чисел, в котором каждое последующее число равно сумме двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13 и т.д. Скрипт выводит двенадцать членов ряда: два (0 и 1) выводятся вне цикла и десять выводятся в результате выполнения цикла.

Как это происходит? Вводятся две переменные (fib1 и fib2), которым присваиваются

начальные значения. Присваиваются значения переменной n и счетчику i, между которыми те или иные математические отношения формируют желаемое число витков цикла. Внутри цикла создается переменная fib_sum, которой присваивается сумма двух предыдущих членов ряда, и ее же значение выводится на экран. Далее изменяются значения fib1 и fib2 (первому присваивается второе, а второму - сумма), а также увеличивается значение счетчика.

Практическая работа

1. Напишите скрипт на языке программирования Python, выводящий ряд чисел Фибоначчи (см. пример выше). Запустите его на выполнение. Затем измените код так, чтобы выводился ряд чисел Фибоначчи, начиная с пятого члена ряда и заканчивая двадцатым.
2. Напишите цикл, выводящий ряд четных чисел от 0 до 20. Затем, каждое третье число в ряде от -1 до -21.
3. Самостоятельно придумайте программу на Python, в которой бы использовался цикл **while**.

Урок 8.
Ввод данных с клавиатуры

Компьютерные программы обрабатывают данные, производя над ними операции, которые задал программист, и которые были обусловлены поставленными задачами. Данные в программу можно "заложить" в процессе ее разработки. Однако такая программа всегда будет обрабатывать одни и те же данные и возвращать один и тот же результат. Чаще требуется совершенно другое — программа должна обрабатывать разные (относительно, в определенном диапазоне) данные, которые поступают в нее из внешних источников. В качестве последних могут выступать файлы или клавиатура. Когда информация вводится с клавиатуры, а результаты выводятся на экран монитора, то можно говорить об интерактивном режиме работы программы. Она обменивается информацией с внешней для нее средой: может выводить и получать данные в процессе выполнения, и не является замкнутой сама на себе. С выводом данных мы уже отчасти знакомы: выводом на экран (и не только) в языке программирования Python занимается функция **print()**.

Ввод данных с клавиатуры в программу (начиная с версии Python 3.0) осуществляется с помощью функции **input()**. Когда данная функция выполняется, то поток выполнения программы останавливается в ожидании данных, которые пользователь должен ввести с помощью клавиатуры. После ввода данных и нажатия Enter, функция **input()** завершает свое выполнение и возвращает результат, который представляет собой строку символов, введенных пользователем.

```
>>> input()
1234
'1234'
>>> input()
Hello World!
'Hello World!'
>>>
```

Когда выполняющаяся программа предлагает пользователю что-либо ввести, то пользователь может не понять, что от него хотят. Надо как-то сообщить, ввод каких данных ожидает программа. С этой целью функция **input()** может принимать необязательный аргумент-приглашение строкового типа; при выполнении функции сообщение будет появляться на экране и информировать человека о запрашиваемых данных.

```
>>> input("Введите номер карты: ")
Введите номер карты: 98765
'98765'
>>> input('Input your name: ')
Input your name: Sasha
'Sasha'
>>>
```

Из примеров видно, что данные возвращаются в виде строки, даже если было введено число. В более ранних версиях Python были две встроенные функции, позволяющие получать данные с клавиатуры: **raw_input()**, возвращающая в программу строку и

input(), возвращающая число. Начиная с версии Python 3.0, если требуется получить число, то результат выполнения функции **input()** изменяют с помощью функций **int()** или **float()**.

```
>>> input('Введите число: ')
Введите число: 10
'10'
>>> int(input('Введите число: '))
Введите число: 10
10
>>> float(input('Введите число: '))
Введите число: 10
10.0
>>>
```

Результат, возвращаемый функцией **input()**, обычно присваивают переменной для дальнейшего использования в программе.

```
>>> userName = input('What is your name? ')
What is your name? Masha
>>> exp = input('3*34 = ')
3*34 = 102
>>> exp = int(exp) + 21
>>> userName
'Masha'
>>> exp
123
>>>
```

Практическая работа

1. Создайте скрипт (файл data.py), который бы запрашивал у пользователя
   - его имя: "What is your name?"
   - возраст: "How old are you?"
   - место жительства: "Where are you live?"

   , а затем выводил три строки
   - "This is *имя*"
   - "It is *возраст*"
   - "He live in *место_жительства*"

   , где вместо *имя, возраст, место_жительства* должны быть соответствующие данные, введенные пользователем.

2. Напишите программу (файл example.py), которая предлагала бы пользователю решить пример 4*100-54. Если пользователь напишет правильный ответ, то получит поздравление от программы, иначе – программа сообщит ему об ошибке. (При решении задачи используйте конструкцию **if-else**.)

3. Перепишите предыдущую программу так, чтобы пользователю предлагалось решать пример до тех пор, пока он не напишет правильный ответ. (При решении задачи используйте цикл **while**.)

Урок 9.
Строки как последовательности символов

Строки уже упоминались в уроке о типах данных; рассмотрим их более подробно.

Строка — это сложный тип данных, представляющий собой последовательность символов.

Строки в языке программирования Python могут заключаться как в одиночные, так и двойные кавычки. Однако, начало и конец строки должны обрамляться одинаковым типом кавычек.

Существует специальная функция **len()**, позволяющая измерить длину строки. Результатом выполнения данной функции является число, показывающее количество символов в строке.

Также для строк существуют операции *конкатенации* (+) и *дублирования* (*).

```
>>> len('It is a long string')
19
>>> '!!!' + ' Hello World ' + '!!!'
'!!! Hello World !!!'
>>> '-' * 20
'--------------------'
>>>
```

В последовательностях важен порядок символов, у каждого символа в строке есть уникальный порядковый номер — *индекс*. Можно обращаться к конкретному символу в строке и извлекать его с помощью *оператора индексирования*, который представляет собой квадратные скобки с номером символа в них.

```
>>> 'morning, afternoon, night'[1]
'o'
>>> tday = 'morning, afternoon, night'
>>> tday[4]
'i'
>>>
```

В примере, выражение 'morning, afternoon, night'[1] привело к извлечению второго символа. Дело в том, что индексация начинается не с единицы, а с нуля. Поэтому, когда требуется извлечь первый символ, то оператор индексирования должен выглядеть так: [0]. Также позволительно извлекать символы, начиная отсчет с конца. В этом случае отсчет начинается с -1 (последний символ).

```
>>> tday_ru = 'утро, день, ночь'
>>> tday_ru[0]
'у'
>>> tday_ru[-1]
'ь'
>>> tday_ru[-3]
'о'
>>>
```

Очевидно, что удобнее работать не с самими строками, а с переменными, которые на них ссылаются. Результат выполнения выражения индексирования можно присвоить другой переменной.

```
>>> a = "very big string"
>>> a[6]
'i'
>>> b = a[0]
>>> b
'v'
>>>
```

Можно извлекать из строки не один символ, а несколько, т.е. получать срез (*подстроку*). Оператор извлечения среза из строки выглядит так: [X:Y). X – это индекс начала среза, а Y – его окончания; причем символ с номером Y в срез уже не входит. Если отсутствует первый индекс, то срез берется от начала до второго индекса; при отсутствии второго индекса, срез берется от первого индекса до конца строки.

```
>>> tday = 'morning, afternoon, night'
>>> tday[0:7]
'morning'
>>> tday[9:-7]
'afternoon'
>>> tday[-5:]
'night'
>>>
```

Кроме того, можно извлекать символы не подряд, а через определенное количество символов. В таком случае оператор индексирования выглядит так: [X:Y:Z); Z – это шаг, через который осуществляется выбор элементов.

```
>>> str4 = "Full Ball Fill Pack Ring"
>>> str4[::5]
'FBFPR'
>>> str4[0:15:2]
'Fl alFl '
>>>
```

Практическая работа

1. Свяжите переменную с любой строкой, состоящей не менее чем из 8 символов. Извлеките из строки первый символ, затем последний, третий с начала и третий с конца. Измерьте длину вашей строки.
2. Присвойте произвольную строку длиной 10-15 символов переменной и извлеките из нее следующие срезы:
    - первые восемь символов;
    - четыре символа из центра строки;
    - символы с индексами кратными трем.

Урок 10.
Списки — изменяемые последовательности

Списки в языке программирования Python, как и строки, являются упорядоченными последовательностями. Однако, в отличии от строк, списки состоят не из символов, а из различных объектов (значений, данных), и заключаются не в кавычки, а в квадратные скобки [ ]. Объекты отделяются друг от друга с помощью запятой.

Списки могут состоять из различных объектов: чисел, строк и даже других списков. В последнем случае, списки называют вложенными.

```
[23, 656, -20, 67, -45]   # список целых чисел
[4.15, 5.93, 6.45, 9.3, 10.0, 11.6]   # список из дробных чисел
["Katy", "Sergei", "Oleg", "Dasha"]   # список из строк
["Москва", "Титова", 12, 148]   # смешанный список
[[0, 0, 0], [0, 0, 1], [0, 1, 0]]   # список, состоящий из списков
```

Как и над строками над списками можно выполнять операции соединения и повторения:

```
>>> [45, -12, 'april'] + [21, 48.5, 33]
[45, -12, 'april', 21, 48.5, 33]
>>> [[0,0],[0,1],[1,1]] * 2
[[0, 0], [0, 1], [1, 1], [0, 0], [0, 1], [1, 1]]
>>>
```

По аналогии с символами строк, можно получать доступ к объектам списка по их индексам, извлекать срезы, измерять длину списка:

```
>>> li = ['a','b','c','d','e','f']
>>> len(li)
6
>>> li[0]
'a'
>>> li[4]
'e'
>>> li[0:3]
['a', 'b', 'c']
>>> li[3:]
['d', 'e', 'f']
>>>
```

В отличии от строк, *списки — это изменяемые последовательности*. Если представить строку как объект в памяти, то когда над ней выполняются операции конкатенации и повторения, то это строка не меняется, а в результате операции создается другая строка в другом месте памяти. В строку нельзя добавить новый символ или удалить существующий, не создав при этом новой строки. Со списком дело обстоит иначе. При выполнении операций другие списки могут не создаваться, а изменяться непосредственно оригинал. Из списков можно удалять элементы, добавлять новые. При этом следует помнить, многое зависит от того, как вы распоряжаетесь переменными. Бывают ситуации, когда списки все-таки копируются. Например, результат операции присваивается другой переменной.

Символ в строке изменить нельзя, элемент списка — можно:

```
>>> mystr = 'abrakadabra'
>>> mylist = ['ab','ra','ka','da','bra']
>>> mystr[3] = '0'
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    mystr[3] = '0'
TypeError: 'str' object does not support item assignment
>>> mylist[1] = 'ro'
>>> mylist
['ab', 'ro', 'ka', 'da', 'bra']
>>>
```

В списке можно заменить целый срез:

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

Более сложная ситуация:

```
>>> alist = mylist[0:2] + [100,'it is ',200] + mylist[2:] # новый список
>>> a2list = mylist # создается вторая ссылка-переменная на первый список
>>> alist
[10, 20, 100, 'it is ', 200, 'ka', 'da', 'bra']
>>> a2list
[10, 20, 'ka', 'da', 'bra']
>>> a2list[0] = '!!!' # изменяем список
>>> a2list
['!!!', 20, 'ka', 'da', 'bra']
>>> mylist # обе переменные связаны с одним списком
['!!!', 20, 'ka', 'da', 'bra']
>>>
```

Практическая работа

1. Создайте два любых списка и свяжите их с переменными.
2. Извлеките из первого списка второй элемент.
3. Измените во втором списке последний объект. Выведите список на экран.
4. Соедините оба списка в один, присвоив результат новой переменной. Выведите получившийся список на экран.
5. "Снимите" срез из соединенного списка так, чтобы туда попали некоторые части обоих первых списков. Срез свяжите с очередной новой переменной. Выведите значение этой переменной.
6. Добавьте в список-срез два новых элемента и снова выведите его.

Пример выполнения практической работы

1
```
>>> spisok1 = [45, 2, 8, 97, 34]
>>> spisok2 = [65, 23, 10]
```

2
```
>>> spisok1 [1]
2
```

3
```
>>> spisok2 [-1] = 12
>>> spisok2
[65, 23, 12]
```

4
```
>>> big_spisok = spisok1 + spisok2
>>> big_spisok
[45, 2, 8, 97, 34, 65, 23, 12]
```

5
```
>>> small_spisok = big_spisok [3:7]
>>> small_spisok
[97, 34, 65, 23]
```

6
```
>>> small_spisok [4:6] = [56, 84]
>>> small_spisok
[97, 34, 65, 23, 56, 84]
```

Урок 11.
Введение в словари

Одним из сложных типов данных (наряду со строками и списками) в языке программирования Python являются словари. *Словарь - это **изменяемый** (как список) **неупорядоченный** (в отличие от строк и списков) набор пар "ключ:значение".*

Чтобы представление о словаре стало более понятным, можно провести 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 описывать с помощью Python, то английские слова будут ключами, а русские — их значениями:

```
{'cat':'кошка', 'dog':'собака', 'bird':'птица', 'mouse':'мышь'}
```

Обратите внимание на фигурные скобки, именно с их помощью определяется словарь. Синтаксис словаря на Питоне можно описать такой схемой:

Если создать словарь в интерпретаторе Python, то после нажатия Enter можно наблюдать, что последовательность вывода пар "ключ:значение" не совпадает с тем, как было введено:

```
>>> {'cat':'кошка', 'dog':'собака', 'bird':'птица', 'mouse':'мышь'}
{'bird': 'птица', 'mouse': 'мышь', 'dog': 'собака', 'cat': 'кошка'}
>>>
```

Дело в том, что в словаре абсолютно не важен порядок пар, и интерпретатор выводит их в случайном порядке. Тогда как же получить доступ к определенному элементу, если индексация не возможна в принципе? Ответ: в словаре доступ к значениям осуществляется по ключам, которые заключаются в квадратные скобки (по аналогии с индексами строк и списков).

```
>>> dic ={'cat':'кошка','dog':'собака','bird':'птица','mouse':'мышь'}
>>> dic['cat']
'кошка'
>>> dic['bird']
'птица'
>>>
```

Словари, как и списки, являются изменяемым типом данных: можно изменять, добавлять и удалять элементы (пары "ключ:значение"). Изначально словарь можно создать пустым (например, d = {}) и лишь потом заполнить его элементами. 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). Удаление элемента словаря осуществляется с помощью функции **del()**.

```
>>> dic ={'cat':'кошка','dog':'собака','bird':'птица','mouse':'мышь'}
>>> dic['elephant'] = 'бегемот'
>>> dic['fox'] = 'лиса'
>>> dic
```

```
{'fox': 'лиса', 'dog': 'собака', 'cat': 'кошка', 'elephant': 'бегемот', 'mouse': 'мышь',
'bird': 'птица'}
>>> dic['elephant'] = 'слон'
>>> del(dic['bird'])
>>> dic
{'fox': 'лиса', 'dog': 'собака', 'cat': 'кошка', 'elephant': 'слон', 'mouse': 'мышь'}
>>>
```

Тип данных ключей и значений словарей не обязательно должны быть строками. Значения словарей могут быть более сложными (содержать структуры данных, например, другие словари или списки).

```
>>> d = {1:'one',2:'two',3:'three'}
>>> d
{1: 'one', 2: 'two', 3: 'three'}
>>> d = {10:[3,2,8], 100:[1,10,5], 1000:[23,1,5]}
>>> d
{1000: [23, 1, 5], 10: [3, 2, 8], 100: [1, 10, 5]}
>>> d = {1.1:2, 1.2:0, 1.3:8}
>>> d
{1.3: 8, 1.2: 0, 1.1: 2}
>>> d = {1.1:2, 10:'apple', 'box':100}
>>> d
{'box': 100, 10: 'apple', 1.1: 2}
>>>
```

Словари — это широко используемый тип данных языка Python. Для работы с ними существует ряд встроенных функций.

Практическая работа

1. Создайте словарь, связав его с переменной school, и наполните его данными, которые бы отражали количество учащихся в десяти разных классах (например, 1а, 1б, 2б, 6а, 7в и т.д.).
2. Узнайте сколько человек в каком-нибудь классе.
3. Представьте, что в школе произошли изменения, внесите их в словарь:
    - в трех классах изменилось количество учащихся;
    - в школе появилось два новых класса;
    - в школе расформировали один из классов.
4. Выведите содержимое словаря на экран.

Урок 12.
Цикл for в языке программирования Python

В седьмом уроке был рассмотрен цикл **while**. Однако это не единственный способ организации в языке Python повторения группы выражений. В программах, написанных на Питоне, широко применяется цикл **for**, который представляет собой цикл обхода заданного множества элементов (символов строки, объектов списка или словаря) и выполнения в своем теле различных операций над ними. Например, если имеется список чисел, и необходимо увеличить значение каждого элемента на две единицы, то можно перебрать список с помощью цикла **for**, выполнив над каждым его элементом соответствующее действие.

```
>>> spisok = [0,10,20,30,40,50,60,70,80,90]
>>> i = 0
>>> for element in spisok:
        spisok[i] = element + 2
        i = i + 1

>>> spisok
[2, 12, 22, 32, 42, 52, 62, 72, 82, 92]
>>>
```

В примере переменная i нужна для того, чтобы записать изменившееся значение элемента в список. В ней хранится значение индекса очередного элемента списка. В то время, как переменная element связывается со значением очередного элемента данных. В заголовке цикла **for** происходит обращение очередному элементу списка. В теле цикла элементу с индексом i присваивается сумма значения текущего (обрабатываемого) элемента и двойки. Далее индекс увеличивается на единицу, а поток выполнения программы переходит снова в заголовок цикла **for**, где происходит обращение к следующему элементу списка. Когда все элементы обработаны цикл **for** заканчивает свою работу. Отсутствие очередного элемента является условием завершения работы цикла **for** (для сравнения: в цикле **while** условием завершения служит результат **false** логического выражения в заголовке). Еще один момент: если счетчик не увеличивать на единицу (выражение i = i + 1), то не смотря на то, что все элементы списка будут обработаны, результат все время будет присваиваться первому элементу списка (с индексом 0).

С таким же успехом перебирать можно и строки, если не пытаться их при этом изменять:

```
>>> stroka = "привет"
>>> for bukva in stroka:
        print(bukva, end=' * ')


п * р * и * в * е * т *
>>>
```

Цикл **for** используется и для работы со словарями:

```
>>> d = {1:'one',2:'two',3:'three',4:'four'}
>>> for key in d:
        d[key] = d[key] + '!'

>>> d
{1: 'one!', 2: 'two!', 3: 'three!', 4: 'four!'}
>>>
```

Цикл **for** широко используется в языке программирования Python, т.к. является важным инструментом при обработки структур данных. Также следует запомнить, что цикл **for** в Питоне особенный. Он не является аналогом циклов **for** во многих других языках программирования, где представляет собой, так называемый, *цикл со счетчиком*.

Практическая работа

1. Создайте список, состоящий из четырех строк. Затем, с помощью цикла **for**, выведите строки поочередно на экран.
2. Измените предыдущую программу так, чтобы в конце каждой буквы строки добавлялось тире. (Подсказка: цикл **for** может быть вложен в другой цикл.)
3. Создайте список, содержащий элементы целочисленного типа, затем с помощью цикла перебора измените тип данных элементов на числа с плавающей точкой. (Подсказка: используйте встроенную функцию **float()**.)

Пример выполнения практической работы

```
>>> # задание 1
>>> list1 = ['hi','hello','good morning','how do you do']
>>> for i in list1:
        print(i)

hi
hello
good morning
how do you do
>>> # задание 2
>>> for i in list1:
        for j in i:
                print(j,end='.')
        print()


h.i.
h.e.l.l.o.
g.o.o.d. .m.o.r.n.i.n.g.
h.o.w. .d.o. .y.o.u. .d.o.
>>> # задание 3
>>> list2 = [56,78,45,23]
>>> i = 0
>>> for a in list2:
        list2[i] = float(a)
```

```
        i = i + 1

>>> list2
[56.0, 78.0, 45.0, 23.0]
>>>
```

Урок 13.
Функции в программировании

Функции в программировании можно представить как изолированный блок кода, обращение к которому в процессе выполнения программы может быть многократным. Зачем нужны такие блоки инструкций? В первую очередь, чтобы сократить объем исходного кода: рационально вынести часто повторяющиеся выражения в отдельный блок и, затем, по мере надобности, обращаться к нему.

Представим себе следующую ситуацию. Требуется написать скрипт, который при выполнении должен три раза запрашивать у пользователя разные данные, но выполнять с ними одни и те же действия.

```
a = int(input('Введите первое число: '))
b = int(input('Введите второе число: '))
if a > b:
    print(a-b)
else:
    print(b-a)

c = int(input('Введите первое число: '))
d = int(input('Введите второе число: '))
if c > d:
    print(c-d)
else:
    print(d-c)

e = int(input('Введите первое число: '))
f = int(input('Введите второе число: '))
if e > f:
    print(e-f)
else:
    print(f-e)
```

Данная программа находит модуль разницы двух чисел. Очевидно, что такая запись исходного кода не рациональна: получаются три почти одинаковых блока кода. Почему бы не использовать цикл **while** для организации повторения?

```
i = 0
while i < 3:
    a = int(input('Введите первое число: '))
    b = int(input('Введите второе число: '))
    if a > b:
        print(a-b)
    else:
        print(b-a)
    i = i + 1
```

Однако, в этом случае есть один нюанс. Вводимые пользователем данные всегда связываются с переменными a и b. При каждом витке цикла прежние данные утрачиваются. Что же делать, если все шесть чисел, введенных пользователем надо сохранить для дальнейшего использования в программе? Рассмотрим решение этой задачи с использованием функции.

```
def diff():
    m = int(input('Введите первое число: '))
    n = int(input('Введите второе число: '))
    if m > n:
        print(m-n)
    else:
        print(n-m)
    return m,n

a,b = diff()
c,d = diff()
e,f = diff()
```

def – это инструкция (команда) языка программирования Python, позволяющая создавать функцию. diff – это имя функции, которое (так же как и имена переменных) может быть почти любым, но желательно осмысленным. После в скобках перечисляются параметры функции. Если их нет, то скобки остаются пустыми. Далее идет двоеточие, обозначающее окончание заголовка функции (аналогично с условиями и циклами). После заголовка с новой строки и с отступом следуют выражения тела функции. В конце тела функции присутствует инструкция **return** (может и не быть), которая возвращает значение(я) в основную ветку программы. В данном случае, если бы в функции не было инструкции **return**, то в основную программу ничего бы не возвращалось, и переменным a и b (c и d, а также e и f) числовые значения не присваивались бы.

После функции идет, так называемая, основная ветка программы, в которой переменным попарно присваивается результат выполнения вызываемой функции. В иных ситуациях, когда функция не возвращает значений, ее вызов не связывается с переменной.

Выражения тела функции выполняются лишь тогда, когда она вызывается в основной ветке программы. Так, например, если функция присутствует в исходном коде, но нигде не вызывается в нем, то содержащиеся в ней инструкции не будут выполнены ни разу.

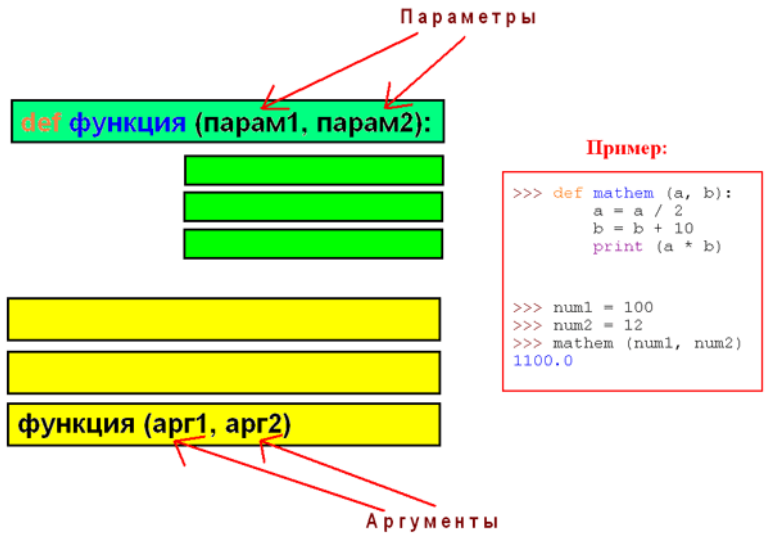
Практическая работа

1. Напишите функцию, которая вычисляет сумму трех чисел и возвращает результат в основную ветку программы.
2. Придумайте программу, в которой из одной функции вызывается вторая. При этом ни одна из них ничего не возвращает в основную ветку программы, обе должны выводить результаты своей работы с помощью функции **print()**.

Урок 14.
Параметры и аргументы функций. Локальные и глобальные переменные

Параметры и аргументы функций

Часто функция используется для обработки данных, полученных из внешней для нее среды (из основной ветки программы). Данные передаются функции при ее вызове в скобках и называются *аргументами*. Однако, чтобы функция могла "взять" передаваемые ей данные, необходимо при ее создании описать *параметры* (в скобках после имени функции), представляющие собой переменные.



Когда функция вызывается, конкретные аргументы подставляются вместо параметров-переменных. Почти всегда количество аргументов и параметров должно совпадать (хотя можно запрограммировать переменное количество принимаемых аргументов). В качестве аргументов могут выступать как непосредственно значения, так и переменные, ссылающиеся на них.

Локальные и глобальные переменные

Если записать в IDLE приведенную ниже функцию, и затем попробовать вывести значения переменных, то обнаружится, что некоторые из них почему-то не существуют:

```
>>> def mathem(a,b):
        a = a/2
        b = b+10
        print(a+b)

>>> num1 = 100
>>> num2 = 12
>>> mathem(num1,num2)
72.0
>>> num1
```

```
100
>>> num2
12
>>> a
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    a
NameError: name 'a' is not defined
>>> b
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    b
NameError: name 'b' is not defined
>>>
```

Переменные num1 и num2 не изменили своих первоначальных значений. Дело в том, что в функцию передаются копии значений. Прежние значения из основной ветки программы остались по прежнему связанны с их переменными.

А вот переменных a и b оказывается нет и в помине (ошибка "name 'b' is not defined" переводится как "переменная b не определена"). Эти переменные существуют лишь в момент выполнения функции и называются *локальными*. В противовес им, переменные num1 и num2 видны не только во внешней ветке, но и внутри функции:

```
>>> def mathem2():
        print(num1+num2)

>>> mathem2()
112
>>>
```

Переменные, определенные в основной ветке программы, являются *глобальными*.

Практическая работа

1. Создайте функцию:
   ```
   def func1(num):
           n = num * 5
           print (n)
   ```
   Вызовите ее, передав в качестве аргумента значение глобальной переменной, затем любое число и, наконец, любую строку.

2. Выполните с помощью интерпретатора Python скрипт, предварительно исправив код функции так, чтобы она возвращала значение переменной n:
   ```
   >>> def func(n):
           if n < 3:
                   n = n*10

   >>> a = 2
   >>> b = func(a)
   >>> a
   2
   >>> b # Почему с переменной не связано никакого значения?
   >>>
   ```

Урок 15.
Проверочная работа по основам программирования на Python

Задание 1

Напишите код по следующему словесному алгоритму:

1. Попросить пользователя ввести число от 1 до 9. Полученные данные связать с переменной x.
2. Если пользователь ввел число от 1 до 3 включительно, то ...
    2.1 попросить ввести строку. Полученные данные связать с переменной s;
    2.2 попросить пользователя ввести число повторов строки. Полученные данные связать с переменной n, предварительно преобразовав их в целочисленный тип;
    2.3 выполнить цикл повторения строки n раз;
    2.4 вывести результат работы цикла.
3. Если пользователь ввел число от 4 до 6 включительно, то ...
    3.1 попросить пользователя ввести степень, в которую следует возвести число. Полученные данные связать с переменной m;
    3.2 реализовать возведение числа x в степень m;
    3.3 вывести полученный результат.
4. Если пользователь ввел число от 7 до 9 включительно, то выполнять увеличения числа x на единицу в цикле 10 раз, при этом на экран выводить все 10 чисел.
5. Во всех остальных случаях выводить надпись "Ошибка ввода".

Задание 2

Напишите программу, которая бы выполняла следующие задачи:

1. выводила название программы "Общество в начале XXI века";
2. запрашивала у пользователя его возраст;
3. если пользователь вводит числа от 0 до 7, то программа выводила надпись "Вам в детский сад";
4. от 7 до 18 - "Вам в школу";
5. от 18 до 25 - "Вам в профессиональное учебное заведение";

6. от 25 до 60 - "Вам на работу";
7. от 60 до 120 – "Вам предоставляется выбор";
8. меньше 0 или больше 120 – пятикратный вывод надписи "Ошибка! Это программа для людей!"

В программе желательно использовать все "атрибуты" структурного программирования: функцию, ветвление и цикл.