GeekBrains

Основы Python

Урок 1. Знакомство с Python

Урок содержит базовую информацию, необходимую для успешного старта в сфере разработки на Python, в том числе описание установки интерпретатора в различные ОС и среды разработки. В рамках курса приводится описание понятия динамической типизации, особенностей использования арифметических и логических операций. Отдельные разделы урока посвящены способам форматирования строк, следованиям, ветвлениям и циклам. В конце приводится список основных ошибок разработчика и пути их решения.

Оглавление

На этом уроке студент:

1. Установит интерпретатор Python и среду разработки PyCharm.
2. Научится создавать небольшие программы, выполнять их отладку и запуск.
3. Научится запрашивать данные для программы и выводить результаты её работы.
4. Познакомится с арифметическими и логическими операциями в Python.
5. Узнает, как реализовать в программе следования, ветвления и циклы.
6. Научится форматировать строки.
7. Узнает об основных ошибках начинающих разработчиков.

Python и его преимущества

Python (читается как Питон или Пайтон) — интерпретируемый, объектно-ориентированный высокоуровневый язык программирования с динамической типизацией.

Интерпретируемый — исходный код программы не преобразуется в машинный для непосредственного выполнения центральным процессором, исполняется с помощью специальной программы-интерпретатора.

Высокоуровневый — наличие в языке смысловых конструкций, кратко описывающих структуры данных и операции над ними. Их описания на машинном коде очень длинны и сложны для понимания.

Преимущества:

1. Минимальный порог вхождения. Благодаря языку программирования Python попробовать свои силы в написании кода может даже человек, никогда не работавший в сфере разработки ПО.
2. «Дружелюбный» синтаксис. Позволяет легко разбираться в собственном коде и читать чужой.
3. Поддержка дополнительных библиотек. Библиотека представляет собой набор компонентов кода, расширяющих стандартные возможности языка.
4. Переносимость программ. Большая часть программ на языке Python выполняется без изменений на всех основных платформах.
5. Прикладная применимость. Python позволяет создавать приложения в различных областях.

На каких проектах применяют Python

Python — язык программирования широкого профиля. С его помощью решаются задачи в таких областях, как:

1. **Веб-приложения**. Python выступает языком реализации логики работы таких приложений (бэкендов).
2. **Алгоритмы машинного обучения**, реализуемые в рекомендательных системах, а также в системах распознавания лиц, голоса и т. д.
3. **Проекты в области искусственного интеллекта (ИИ)**. В Python предусмотрены возможности для создания приложений ИИ.
4. **Игровые приложения**. Для разработки доступны различные игровые движки, например, PyGame.
5. **Приложения с графическим интерфейсом**. Для разработки GUI могут применяться встроенные инструменты (Tkinter), а также сторонние фреймворки (PyQt).
6. **Системы анализа и визуализации данных**. Например, библиотека Matplotlib предоставляет разработчику широкий комплекс средств построения графиков, диаграмм и т. д.
7. **Системные утилиты**. Python — отличный инструмент для приложений управления службами ОС.
8. **Приложения для работы с БД**. В Python предусмотрены программные интерфейсы для работы с большинством СУБД.
9. **Сложные вычисления**. Например, библиотека NumPy позволяет эффективно выполнять математические расчёты.

Ряд проектов, в которых используется Python

1. Торрент-клиент BitTorrent.
2. Центр приложений Ubuntu.
3. Графическая система Blender.
4. Графический редактор Gimp.
5. Игровые проекты: Civilization IV, Battlefield 2, World of Tanks.
6. Сервис DropBox.
7. Видеохостинг YouTube.

8. Роботизированные устройства от iRobot.

Python используют в своих разработках гиганты IT-рынка: IBM, Instagram, Yahoo, Facebook, Google, Mail.ru и т. д.

Python применяют в своих разработках гиганты финансовой сферы: UBS, JPMorgan, Citadel.

Установка интерпретатора в Windows, Linux, MacOS. Особенности запуска Python-скриптов в каждой из ОС

Различия Python 2.x и Python 3.x

Существуют и параллельно развиваются две версии Python — 2 и 3.

Воспользуемся версией Python 3 и не будем говорить о Python 2, потому как поддержка этой версии интерпретатора действует пока только до 2020 года.

Установка

Как уже отмечалось выше, Python — интерпретируемый язык. То есть, чтобы программы выполнялись, на вашем ПК должна быть установлена программа-интерпретатор.

Статья об установке Python.

Установка под Windows

Скачиваем установщик с официального сайта. Возьмём наиболее свежую версию. Нам подойдёт версия 3.5 и старше, желательно установить свежую версию — 3.9. Следуем указанию мастера установки. Процесс установки описан в инструкции.

Установка под Linux

Здесь всё совсем просто: в любой Linux-системе Python предустановлен изначально, поскольку он — стандартный компонент. Но будьте внимательны, сразу установлены две версии Python 2 и Python 3. Загруженная версия третьего Python может быть недостаточно актуальной, поэтому потребуется обновить интерпретатор до свежей версии. Инструкция со скриншотами приведена в отдельном файле в материалах урока. Процесс установки описан в инструкции.

Установка под MacOS

Процесс установки описан в инструкции.

Запуск и выполнение

Программы на Python — это обычные текстовые файлы, которые вы можете набирать в чистом текстовом редакторе. Чистым называется любой текстовый редактор, который не добавляет никаких символов, кроме, набранных вами (MS Word точно не подойдёт).

Например:

- для WIndows: Sublime, Notepad++;
- для Linux: Sublime, Notepadqq;
- для MacOS: Sublime, Coda2.

Под Windows

При установке интерпретатора автоматически установится простая графическая IDLE (среда разработки).

Для запуска: Пуск → Программы → Python 3.x → IDLE (Python GUI).

Чтобы запустить интерактивную оболочку интерпретатора, в командной строке наберём:

```
python
```

Важно! *Если у вас интерпретатор не прописан в переменных среды, то вместо команды* ***python*** *укажите полный путь к интерпретатору Python, например:*

```
C:/Python37/python.exe
```

Под Linux

Для запуска интерактивной оболочки интерпретатора выполним в консоли:

```
python3
```

Оболочка Python — это место, где можно исследовать синтаксис Python, получить интерактивную справку по командам и отлаживать небольшие программы. Сама по себе оболочка Python — замечательная интерактивная площадка для игр с языком.

Как правило, программы состоят более, чем из одной строки. Для ввода полноценной программы нужно воспользоваться любым текстовым редактором, например, Notepad++. Все скрипты (программы) Python должны иметь расширение **.py**.

Для запуска Python-скрипта:

```
python <путь к скрипту>/<имя_скрипта>.py
```

Пример (для Windows):

```
python C:/scripts/my_script.py
```

Обратите внимание

Python — мультиплатформенный язык программирования. Это значит, что программа будет одинаково работать на любой операционной системе. Например, если вы работаете под MacOS, а преподаватель — под Windows, вы также успешно сможете пройти курс. Всё, что от вас требуется — корректно выполнить установку интерпретатора и среды разработки для своей операционной системы. Сам код, который пишете вы, преподаватель, одногруппники и все программисты Python на планете, одинаково работает и на MacOS, и на Linux, и на Windows.

Что такое IDE. Особенности установки и запуска PyCharm в различных ОС

Набирать программы в текстовом редакторе, а потом смотреть результат в консоли не очень удобно и занимает много времени. Поэтому рекомендуем пользоваться IDE. Можете использовать любую привычную вам IDE. Хорошая IDE — PyCharm.

IDE (интегри́рованная среда́ разрабо́тки. англ. Integrated development environment) — комплекс программных средств, используемый программистами для разработки ПО.

PyCharm можно скачать с официального сайта для различных ОС. Community-версия бесплатна, её опций на 100% хватит для изучения Python.

Особенности установки IDE PyCharm для каждой из представленных ОС приведены в файлах-инструкциях материалов урока.

Установить PyCharm.

Итак, интерпретатор установлен, текстовый редактор готов к приёму ваших первых программ. И как говорится, лучший способ познакомиться с языком программирования — это начать на нём писать.

Введение в стандарты программирования на Python

Одно из важнейших требований к коду Python-разработчика — следование стандарту PEP-8. Это описание рекомендованного стиля кода. Причём PEP-8 действует для основного текста программы, а для строк документации разработчику рекомендуется придерживаться положений PEP-257. Документ содержит объёмное описание стандарта. На этом курсе мы познакомимся только с частью его положений, необходимых для отработки учебных примеров и выполнения практических заданий.

1. **Избегайте дополнительных пробелов в скобках (круглых, квадратных, фигурных).**

Некорректно:

```
x = [ '2', 4 ]
y = ( x [ 1 ] , x [0] )
z = { 'key' : y [ 0 ] }
```

Корректно:

```
x = ['2', 4]
y = (x[1], x[0])
z = {'key': y[0]}
```

2. **Используйте пробелы вокруг арифметических операций.**

Некорректно:

```
(a+b)+c=a+(b+c)
```

Корректно:

```
(a + b) + c = a + (b + c)
```

3. **Имена переменных и функций, атрибутов и методов класса задавайте в нижнем регистре. Разделяйте подчёркиванием входящие в имена слова.**

Некорректно:

```
MyVar, myVar, Var, VAR, MyFunc, myFunc, Func, FUNC
```

Корректно:

```
var, my_var, func, my_func
```

4. При оформлении блоков кода в Python позаботьтесь об отступах.

Рекомендуемый отступ составляет четыре пробельных символа. Знаки табуляции применять не рекомендуется. В популярных IDE не требуется ставить пробелы вручную. При переходе на очередную строку программного кода число пробельных символов определяется автоматически.

Некорректно:

```
Главная инструкция:
    Вложенная инструкция
```

Корректно:

```
Главная инструкция:
    Вложенная инструкция
```

Визуально фрагменты кода идентичны, но в первом отступ выполнен с помощью табуляции, а во втором — с помощью четырёх пробельных символов.

5. Будьте внимательнее при комбинировании апострофов и кавычек.

При определении строк кавычки и апострофы равнозначны. Но во время их комбинирования возможны ошибки:

Некорректно:

```
print("This is my string - "text"")
print('This is my string - 'text'')
```

Корректно:

```
print("This is my string - 'text'")
```

```
print('This is my string - "text"')
```

Из чего состоит программа

Суть любой программы — получение, обработка и вывод данных. Данные в Python представлены объектами.

Программы на языке Python можно разложить на такие составляющие, как модули, инструкции, выражения и объекты.

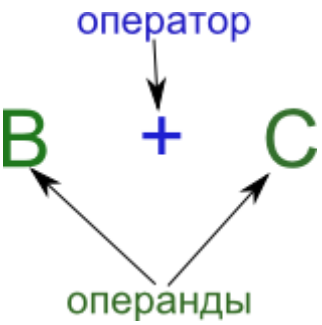
При этом:

1. Программы делятся на модули (файлы с расширением .py).
2. Модули содержат инструкции.
3. Инструкции состоят из выражений.
4. Выражения создают и обрабатывают объекты.

Понятия (выражения, операции, инструкции) довольно условны, но для эффективного понимания языка определимся с терминами, которыми мы будем оперировать.

Операция (англ. statement) — наименьшая автономная часть языка программирования; команда.

Пример операции:



Если в оболочке Python мы введём:

```
>>> 2 + 4
```

Получим результат — 6.

- 2 + 4 — операция;
- 2 и 4 — операнды;
- + — оператор;
- 6 — результат операции.

Операции, которые возвращают результат, будем называть выражениями. Действия, которые не возвращают результат, а указывают интерпретатору, что делать, — это инструкции.

Выражение — это операция, которая возвращает значение.

Инструкция — операция, которая не возвращает значение.

Чтобы сохранить некоторые значения (данные) и воспользоваться ими далее в программе, используются переменные.

Рассмотрим такой пример:

```
a = 10
b = a + 5
print("10+5 =", b)
```

Разберём каждую строчку нашей программы:

1. a = 10 — создаём переменную **a** и присваиваем ей значение 10, то есть теперь в переменной **a** у нас хранится значение 10.
2. b = a + 5 — создаём новую переменную **b**, затем присваиваем ей выражение (a + 5). Так как в переменной **a** у нас хранится значение 10, вместо **a** подставляется её значение — 10. Получаем: b = (10 + 5). Или после сложения: b = 15.
3. print(“10 + 5 = ”, b) — команда (функция) print() выводит на экран значение аргументов или, проще говоря, те данные, которые указали в скобках. print() может выводить сразу несколько значений, для этого аргументы указываются через запятую. Наша функция имеет два аргумента: “10 + 5 = ” и b. Первый аргумент — это строка текста. Строку текста легко можно отличить по верхним кавычкам “”. Второй аргумент — переменная **b**, но на экран выводится не имя переменной, а её значение.

Об аргументах будет сказано подробнее в теме функций. Пока просто запомните, что это данные, которые указываем в скобках через запятую. После каждой запятой идёт новый аргумент.

Важно! *Если в качестве операнда в выражении используется имя переменной, то вместо имени подставляется её значение. Прежде чем использовать переменную, её нужно объявить.*

Переменная — поименованная область памяти, имя или адрес, который можно использовать для осуществления доступа к данным, находящимся в переменной (по этому адресу).

Присваивание переменной — передача в переменную нового значения.

Значение переменной — информация, хранящаяся в переменной. В переменной может храниться текст, целое число, число с десятичной точкой и т. д.

Знак **=** — операция присваивания, а также инструкция. То есть такая операция не возвращает результата.

Динамическая типизация как один из важнейших аспектов программирования на Python

Python поддерживает динамическую типизацию, то есть тип переменной определяется автоматически во время исполнения. Поэтому вместо «присваивания значения переменной» лучше говорить о «связывании значения с некоторым именем».

```
>>> a = 8
```

Рассмотрим, как Python обработает это выражение:
В памяти будет создан объект целого типа (int), переменная **a** получит ссылку на этот объект.
Чтобы лучше понять суть динамической типизации, рассмотрим следующий пример:

```
>>> a = 4
>>> a = a + 1
>>> a = "text"
```

В памяти создаётся объект типа int (целое), переменная **a** получает на него ссылку.

В правой части оператора **=** стоит выражение, и сначала будет вычислен результат выражения. После вычисления результата создаётся новый объект типа int (со значением 5). Переменная **a** получит ссылку на новый объект в памяти. На старый объект int (со значением 4) она больше не будет ссылаться.

Затем создаётся новый объект типа str (строка), переменная **a** снова изменит ссылку.

В отличие от языков со статической типизацией, таких как C++ или Pascal, переменная в Python не имеет типа! Правильно говорить: «Переменная указывает на объект такого-то типа». То есть именно объект в памяти имеет тип, а переменная — просто указатель.

Поэтому когда мы связываем с переменной некоторое значение, просто переносим указатель на другой объект. Python предоставляет мощную коллекцию объектных типов, встроенных напрямую в язык.

Встроенные типы данных (часть):

Название типа	Описание	Примечание

int	Это функция, возвращающая целое число в десятичной системе счисления. Пример: 2, 4, 8, -10, -2	См. урок 2
float	Это функция, возвращающая число с плавающей запятой. Пример: 2.6, -5.2	См. урок 2
str	Это функция, возвращающая строку (неизменяемую последовательность символов)	См. урок 2
bool	Это функция, возвращающая булево значение (True или False) для объекта	См. урок 2
list	Функция, возвращающая изменяемую упорядоченную коллекцию объектов произвольных типов. Пример: [2, 2.4, “Hello”]	См. урок 2
tuple	Функция, возвращающая неизменяемую упорядоченную коллекцию объектов произвольных типов. Кортеж. Пример:(2, 2.4, “Hello”)	См. урок 2
dict	Функция, возвращающая неупорядоченную коллекцию произвольных объектов с доступом по ключу. Пример: {“name”: “Вася”, “age”: 10}	См. урок 2

Механизмы реализации ввода/вывода данных

Основное назначение компьютерных программ — обработка данных. Программа может получать их разными способами, например, запрашивать у пользователя. Результат обработки данных может быть возвращён пользователю посредством вывода на экран в текстовой форме.

Чтобы запросить данные у пользователя с клавиатуры, воспользуемся функцией **input()**.

Функция **input()** может получать необязательный аргумент — строку, которая будет выведена в качестве приглашения/уточнения. В качестве результата она вернёт введённые пользователем данные.

```
>>> name = input("Введите ваше имя: ")
```

Введите ваше имя: <здесь программа остановится и будет ждать ввода с клавиатуры>.

Переменной **name** будет присвоена строка введённых символов.

Обратите внимание: input() всегда возвращает строку. Если вы хотите работать с цифрами, используйте функции преобразования типов **int()**, **float()**.

```
>>> a = int(input("Введите целое число: "))
```

Применять функцию **str()** к вводимым строковым данным не требуется.

Неправильно:

```
>>> a = str(input("Введите текст: "))
```

Правильно:

```
>>> a = input("Введите текст: ")
```

Для вывода в консоль пользуемся функцией **print()**.

Функция **print()** принимает неограниченное количество аргументов, которые будут выведены на экран.

```
>>> name = "Вася"
>>> print("Меня зовут", name)
Меня зовут Вася
```

Арифметические и логические операции в Python

Список доступных арифметических операций в Python приводится в таблице:

Арифметические операторы в Python

Оператор	Описание	Примеры
+	Сложение	print(398 + 20) -> 418
-	Вычитание	print(200 - 50) -> 150
*	Умножение	print(34 * 7) -> 238
/	Деление	print(36 / 6) -> 6.0 print(36 / 5) -> 7.2 print(round(36 / 7, 2)) -> 5.14 print(round(-36 / -7, 3)) -> 5.143
//	Целочисленное деление	print(36 // 6) -> 6 print(36 // 5) -> 7 print(-9 // 4) -> -3 print(5 // -2) -> -3
%	Остаток от деления	print(36 % 6) -> 0 print(36 % 5) -> 1
**	Возведение в степень	print(2 ** 16) -> 65536

Обратите внимание на операцию целочисленного деления с участием отрицательных чисел в качестве делимого или делителя. Сравним два примера:

```
print(9 / 4)
print(-9 / 4)
```

Результат:

```
2.25
-2.25
```

Теперь:

```
print(9 // 4)
print(-9 // 4)
```

Результат:

```
2
-3
```

Такой результат обусловлен тем, что целочисленное деление в Python 3 округляет итоговое значение в меньшую сторону. То есть для числа 2.25 это 2, а для числа -2.25 — -3.

С логическими операциями мы отлично знакомы с уроков математики.

Логические операторы в Python

Оператор	Описание	Примеры
>	Больше	print(40 > 40) -> False
<	Меньше	print(3 < 9) -> True
==	Равно	print(10 == 10) -> True
!=	Не равно	print(2 != 2) -> False
>=	Больше или равно	print(40 >= 1) -> True
<=	Меньше или равно	print(3 <= 1) -> False
and	Логическое «И». Возвращает значение «Истина», если оба операнда имеют значение «Истина»	print(True and True) -> True print(True and False) -> False print(False and True) -> False print(False and False) -> False

or	Логическое «ИЛИ». Возвращает значение «Истина», если хотя бы один из операндов имеет значение «Истина»	print(True or True) -> True print(True or False) -> True print(False or True) -> True print(False or False) -> False
not	Логическое «НЕ». Изменяет логическое значение операнда на противоположное	print(not True) -> False print(not False) -> True
in	Оператор проверки принадлежности. Возвращает значение «Истина», если элемент присутствует в последовательности (см. Урок 2)	print(10 in [10, 20, 30]) -> True
is	Оператор проверки тождественности. Возвращает значение «Истина», если операнды ссылаются на один объект (см. Урок 2)	x = 3 y = 3 print(x is y) -> True

Операторные скобки

В любом языке программирования нужно выделять блоки кода. Для этого используются специальные синтаксические конструкции, показывающие начало и конец блока. В Pascal это ключевые слова begin… end; в C++ — фигурные скобки {...}. В Python — операторные скобки, одинаковые отступы слева перед всеми инструкциями блока.

Подобный синтаксис языка хорош тем, что заставляет программиста правильно табулировать свой код, улучшая читабельность.

Символ конца строки в Pascal — точка с запятой. Это значит, что любой код на этом языке можно писать в одну строку. Это сильно ухудшает читабельность.

Следования, ветвления и циклы в Python, вложенные инструкции

Следования, ветвления и циклы

В теории программирования доказано, что программу для решения любой задачи можно составить из трёх структур, называемых следованием, ветвлением и циклом.

Следованием называется конструкция, представляющая собой последовательное выполнение двух или более операторов (простых или составных).

Ветвление задаёт выполнение либо одного, либо другого оператора в зависимости от выполнения какого-либо условия.

Со следованием всё просто: все команды (инструкции) выполняются последовательно, пока программа не завершится.

Познакомимся с ветвлениями.

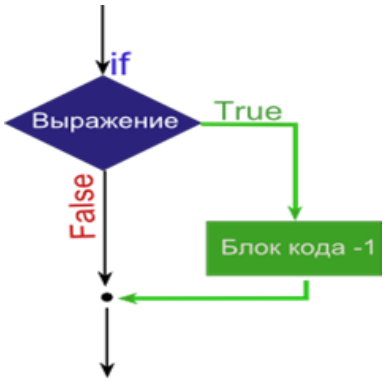


Рис.1 Схема ветвления **if**.

Описание схемы

Оператор **if** называют инструкцией. Помните, что такое инструкция? В качестве выражения может выступать любое выражение, которое будет автоматически преобразовано в логическое.

Цель **if** — выполнить некоторый блок кода при определённом условии.

Если выражение истинно (**True**), то выполняется «Блок кода-1». При ложном выражении (**False**) «Блок кода-1» пропускается, программа выполняется дальше.

Пример:

```
original_password = 'x777' # правильный пароль, хранится в программе
password = input('Введите пароль: ')  # просим пользователя ввести пароль
access = False  # переменная, хранит разрешение на доступ
if password == original_password: # если введен правильный пароль
    print('Пароль принят, добро пожаловать в систему')
    access = True
if password != original_password: # если введен неправильный пароль
    print('Пароль неверен, вход запрещен')
```

Рассмотрим этот пример подробнее.

Цель программы — запросить у пользователя пароль, в случае его корректного ввода дать доступ. Разрешение доступа контролируется переменной access (доступ).

В 4 строке сравниваем введённый пользователем пароль с паролем, хранящимся в программе. Если они равны, то сообщаем пользователю, что его пароль принят, и меняем значение переменной **access** на **True** (**True** — доступ разрешён, **False** — запрещён).

В 7 строке проверяем, если пароль введён неверно, сообщаем об этом пользователю. Так как пароль неверный, значение переменной access оставляем в значении **False**.

В этом примере мы также увидели текстовое описание, которому предшествует символ **#.** В Python так обозначаются однострочные комментарии в коде. Многострочный комментарий в этом случае потребует символа **#** перед каждой строкой, что при большом блоке-комментарии ухудшает презентабельность кода. В этом случае лучше использовать многострочный комментарий:

```
'''
Строка 1
Строка 2
Строка 3
'''

"""
Строка 1
Строка 2
Строка 3
"""
```

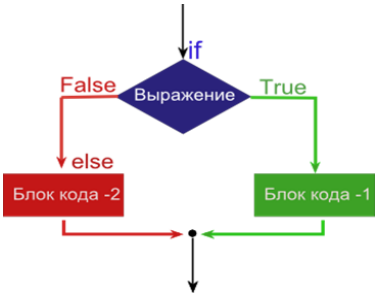


Рис.2 Схема ветвления **if else**

Описание схемы

Если выражение истинно (**True**), то выполняется «Блок кода-1», при ложном выражении (**False**) — «Блок кода-2». То есть выполняется либо первый блок, либо второй.

Используя эти знания, предыдущий пример можно переписать.

Пример:

```
original_password = 'x777' # правильный пароль, хранится в программе
password = input('Введите пароль: ') # просим пользователя ввести пароль
access = False # переменная, хранит разрешение на доступ
# Если введен правильный пароль
if password == original_password:
    print('Пароль принят, добро пожаловать в систему')
    access = True
# Иначе, т.е. если неправильный пароль
else:
    print('Пароль неверен, вход запрещен')
```

Вложенные инструкции

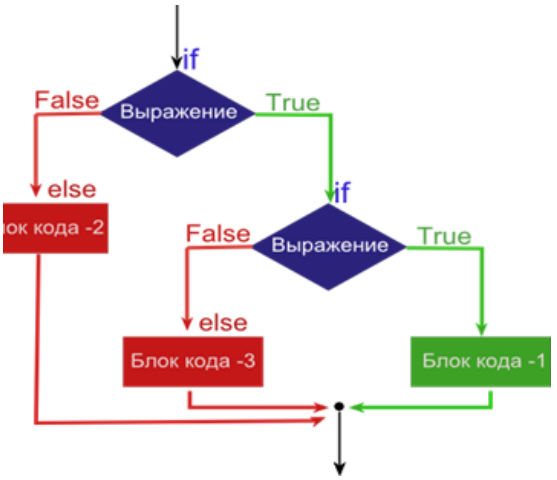


Рис. 3 Схема вложенных инструкций ветвления

Внутри блока условной инструкции могут находиться любые другие инструкции, в том числе и условная. Это вложенные инструкции. Синтаксис вложенной условной инструкции:

```
if условие1:
...
    if условие2:
    ...
    else:
    ...
    ...
else:
...
```

Вместо многоточий можно писать произвольные инструкции. Обратите внимание на размеры отступов перед инструкциями. Блок вложенной условной инструкции отделяется четырьмя пробельными символами.

Уровень вложенности условных инструкций может быть произвольным. То есть внутри одной условной инструкции может быть вторая, а внутри неё — ещё одна и т. д. Условие 2 проверяется, только если верно условие 1.

Вложенные инструкции на одном уровне вложенности

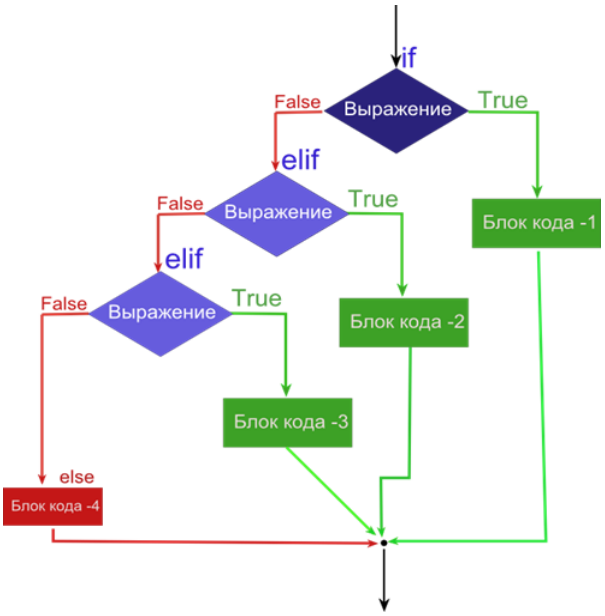


Рис.4 Полная схема инструкции ветвления

Описание схемы

Оператор **elif** переводится как «иначе если». Логическое выражение, стоящее после оператора elif, проверяется, только если все вышестоящие условия ложные. То есть в этой схеме может выполниться только один блок кода: первый, второй, третий или четвёртый. Если одно из выражений истинно, то нижестоящие условия проверяться не будут.

Пример:

```
color = 'red'
if color == 'blue':
    print('синий')
# elif сокращение от else if(иначе если)
elif color == 'red':
    print('красный')
elif color == 'green':
    print('зеленый')
# else выполняется, только если все предыдущие проверки вернули False
else:
```

```
    print('неизвестный цвет')
```

Чтобы проверялись все условия, независимо от результата предыдущего, следует использовать несколько независимых операторов **if**.

Рассмотрим ещё один пример:

```
numb_1 = int(input("Введите первое целое число: "))
numb_2 = int(input("Введите второе целое число: "))

if numb_1 != numb_2:
    print("Числа не равны")
    if numb_1 > numb_2:
        print("Первое число больше второго")
    elif numb_1 < numb_2:
        print("Первое число меньше второго")
elif numb_1 == numb_2:
    print("Числа равны")
```

Результат:

```
Введите первое число: 40
Введите второе число: 20
Числа не равны
Первое число больше второго
```

Пример:

```
numb_1 = float(input("Введите первое вещественное число: "))
numb_2 = float(input("Введите второе вещественное число: "))

if numb_1 >= numb_2:
    print("Первая ветвь")
    if numb_1 > numb_2:
        print("Первое число больше второго")
    else:
        print("Числа равны")
elif numb_1 <= numb_2:
    print("Вторая ветвь")
    if numb_1 < numb_2:
        print("Первое число меньше второго")
    else:
        print("Числа равны")
```

Результат:

```
Введите первое вещественное число: 4.6
Введите второе вещественное число: 1.2
Первая ветвь
```

```
Первое число больше второго
```

Знакомство с циклами

Цикл задаёт многократное выполнение оператора.

Все программы, которые мы писали до сих пор, запускались, выполняли необходимые действия, выводили результат и завершали свою работу. Чтобы выполнить любую из наших программ с другим набором данных, нужно запустить её заново. Но как много реальных программ вы знаете, которые немедленно завершают свою работу после выполнения некоторых действий?

Практически все программы работают непрерывно: выполнив одни действия, ожидают новых инструкций. И так до тех пор, пока пользователь не завершит работу программы. Работу большинства программ можно представить в таком виде: получение данных/инструкций --> обработка данных --> вывод результата --> получение данных/инструкций --> обработка данных --> вывод результата … Так будет происходить, пока пользователь не завершит работу с программой. Это и есть работа программы в цикле.

Циклы — это инструкции, выполняющие одну и ту же последовательность действий, пока актуально заданное условие.

В Python существуют два типа циклов: **while** и **for in**. В этой лекции мы познакомимся только с первым циклом.

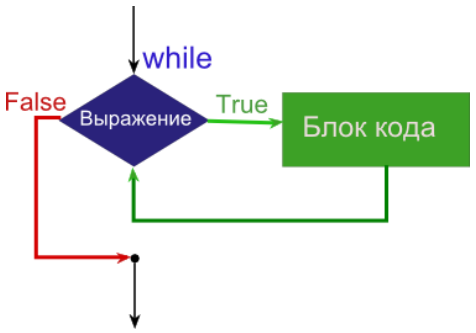


Рис. 5 Схема цикла **while**

Описание схемы

Если выражение истинно (**True**), то выполняется «Блок кода», программа снова возвращается к проверке логического выражения. При ложном выражении (**False**) программа продолжает свою работу, не выполняя «Блок кода». То есть «Блок кода» выполняется до тех пор, пока логическое выражение, стоящее после оператора while, истинно.

Блок кода внутри цикла называется **телом цикла.**

Один шаг цикла (однократное выполнение тела цикла) называется **итерацией.**

Пример цикла:

```
number = int(input('Введите целое число от 0 до 9: '))
while number < 10:
    print(number)
    number = number + 1
print('программа завершена успешно')
```

Рассмотрим пример подробнее.

Программа выводит на экран все числа — от введённого числа до 9, с шагом 1. Например, если мы введём число 7, программа выведет 7, 8 и 9.

Вторая строка — это оператор цикла while и number < 10 — логическое выражение.

Третья и четвёртая строки — это тело цикла, которое будет выполняться до тех пор, пока логическое выражение number < 10 будет истинно. Пятая строчка не относится к телу цикла, так как перед ней нет отступа.

Сколько раз выполнится тело цикла, заранее неизвестно — это зависит от заданного значения переменной number.

Обратите внимание на строчку 4. При каждом выполнении этой строки в цикле её значение будет увеличиваться на единицу до тех пор, пока значение переменной number не станет больше либо равно 10. При этом значении логическое выражение number < 10 станет ложным, цикл завершится.

Зацикливание

Рассмотрим такой пример:

```
a = 5
while a > 0:
    print("!")
    a = a + 1
```

Запустив этот пример, вы увидите кучу восклицательных знаков, и так до бесконечности. Цикл при текущих условиях не завершится никогда, потому что **a** всегда больше нуля, условие **a** > 0 всегда будет верным. В программах нужно избегать бесконечных циклов. Операционная система считает зациклившуюся программу повисшей (нерабочей) и предлагает снять с неё задачу.

Инструкции break, continue

В теле цикла можно использовать вспомогательные инструкции **break** и **continue**. Иногда применение этих инструкций позволяет упростить ваш код и сделать его более читабельным.

Оператор **continue** начинает следующий проход цикла, минуя оставшееся тело цикла.

Оператор **break** досрочно прерывает цикл.

Пример:

```
i = 0
while True:
    i += 1
    if i >= 10:
    # инструкция break при выполнении немедленно заканчивает выполнение цикла
        break
    if i % 2 == 0:
    # переходим к проверке условия цикла,
    # пропуская все операторы за инструкцией
        continue
    print(i)
    #i += 1
```

Подробнее о цикле **while**, операторах **break** и continue читайте по ссылке.

Способы форматирования строк

В процессе работы над программой у разработчика часто возникают ситуации, когда необходимо сформировать строку. Для этого нужно подставить в неё некоторые данные, полученные в процессе выполнения программы. Это данные на основе пользовательского ввода, значения переменных, вывод из файлов и т. д. Для подстановки можно воспользоваться одним из методов форматирования строк: оператором **%**, функцией **format()**, **f**-строками. Последний вариант (**f**-строки) работает быстрее других способов. Поэтому если вы работаете под Python 3.6 и старше, используйте именно его.

Форматирование через оператор %

Если для подстановки требуется только один аргумент, то значением будет сам аргумент.

```
name = input("Enter your name: ")
print("Hello, %s!" % name)
```

Выравнивание по левой стороне.

```
print("%-10s %-10s %-10s" % ('papam1', 'papam2', 'papam3'))
```

Указание количества цифр после запятой.

```
print("%.2f" % (20.0/8))
```

Значения в подстановке могут различаться по типу. В примерах выше мы подставляли значения строкового типа (**%s**), но можно выполнять и другие подстановки.

Подстановка	Тип данных
“%s”	Строка
“%d”	Десятичное число
“%f”	Число с плавающей точкой
“%o”	Число в восьмеричной системе
“%x”	Число в шестнадцатеричной системе

Пример:

```

```

Форматирование через метод format()

Используется специальный символ {} для указания точки подстановки значения, передаваемого методу format. Каждая пара скобок определяет одно место для подстановки.

```
print('{}'.format(['el_1', 'el_2', 'el_3', 'el_4']))
```

Вывод данных столбцами одинаковой ширины по 20 символов с выравниванием по правой стороне.

```
print("{:>20} {:>20} {:>20}".format('my_param_1', 'my_param_2', 'my_param_3'))
```

Указание количества цифр после запятой.

```
print("{:.3f}".format(5.0/3))
```

Передать параметры можно и через указание их индексов в фигурных скобках:

```
print('Третий элемент: {2}; Второй элемент: {1}; Первый элемент:
{0}'.format('el_1', 'el_2', 'el_3'))
```

Результат:

```
Третий элемент: el_3; Второй элемент: el_2; Первый элемент: el_1
```

Форматирование через f-строки

f-строки — механизм форматирования строки с префиксом **f**. Внутри **f**-строки в паре фигурных скобок указываются имена переменных, которые необходимо подставить. Информация об **f**-строках доступна по ссылке.

```
ip = '192.168.1.4'
mask = 10

print(f"ip-params: {ip}, mask: {mask}")
```

Помимо подстановки значений переменных, в фигурных скобках допустимо применить выражение:

```
octets = ['10', '1', '1', '1']
mask = 10

print(f"ip-params: {'.'.join(octets)}, mask: {mask}")
```

Через **f**-строки также возможен вывод столбцами с одинаковым расстоянием между ними:

```
oct1, oct2, oct3, oct4 = [10, 1, 1, 1]

print(f'''IP address: {oct1:<8} {oct2:<8} {oct3:<8} {oct4:<8}''')
```

Частые ошибки начинающих разработчиков. Как их исправить

Проблема 1. TypeError: Can't convert 'int' object to str implicitly

Пример:

```
my_var = input("Введите число: ") + 5
print(my_var)
# Ошибка:
# TypeError: can only concatenate str (not "int") to str
```

Причина: недопустимо применять оператор сложения к строке и числу.

Решение: нужно выполнить преобразование строки к числу, применив функцию **int()**. Обратите внимание, что функция **input()** всегда возвращает строку.

Пример:

```
my_var = int(input("Введите число: ")) + 5
print(my_var)
```

Проблема 2. SyntaxError: invalid syntax

Пример:

```
msg = True
if msg == True
    print("Приветственное сообщение")
# Ошибка:
# SyntaxError: invalid syntax
```

Причина: забыто двоеточие.

Решение:

```
msg = True
if msg == True:
    print("Приветственное сообщение")
```

Проблема 3. SyntaxError: invalid syntax

Пример:

```
msg = True
if msg = True:
    print("Приветственное сообщение")
# Ошибка:
# SyntaxError: invalid syntax
```

Причина: забыт знак равенства.

Решение:

```
msg = True
if msg == True:
    print("Приветственное сообщение")
```

Проблема 4. NameError: name 'my_var' is not defined

Пример:

```
print(my_var)
# Ошибка:
# NameError: name 'my_var' is not defined
```

Причина: переменной **my_var** нет. Возможно, переменная есть, но неправильно указано её имя, или программист забыл инициализировать переменную.

Решение:

```
my_var = "какое-то значение переменной"
print(my_var)
```

Проблема 5. IndentationError: expected an indented block

Пример:

```
my_var = True
if my_var == True:
print("Все верно")
# Ошибка:
# IndentationError: expected an indented block
```

Причина: требуется отступ.

Решение:

```
my_var = True
if my_var == True:
    print("Все верно")
```

Проблема 6. Inconsistent use of tabs and spaces in indentation

Пример:

```
my_var = True
if my_var == True:
    print("Все верно")
    print("Работа программы завершена")
# Ошибка:
# Inconsistent use of tabs and spaces in indentation
```

Причина: использование пробелов и табуляций в отступах по одной программе (файлу-модулю).

Решение: Привести все отступы к единообразию (везде использовать пробелы).

Проблема 7. UnboundLocalError: local variable 'my_var' referenced before assignment

Пример:

```
def my_func():
    my_var += 1
    print(my_var)
```

```
my_var = 10
my_func()
# Ошибка:
# UnboundLocalError: local variable 'my_var' referenced before assignment
```

Причина: попытка обратиться к локальной переменной, которая ещё не создана.

Решение:

```
def my_func(my_var):
    my_var += 1
    print(my_var)

my_var = 10
my_func(my_var)
```

Сводная таблица «Зарезервированные слова»

Модуль keyword позволяет получить список слов (kwlist), зарезервированных для интерпретатора:

Пример:

```
from keyword import kwlist
print(kwlist)

'''
['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break',
'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for',
'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
'''
```

Описание ключевых слов:

Название	Описание	Примечание
`False`	Значение «Ложь»	См. урок 2
`None`	«Не определён», пустой объект	См. урок 2
`True`	Значение «Истина»	См. урок 2
`and`	Логическое «И»	См. урок 1
`as`	Определение псевдонима для объекта	См. урок 4
`assert`	Генерация исключения, если условие ложно	—

`async`	Обозначение функций как сопрограмм для использования цикла событий	—
`await`		
`break`	Выход из цикла	См. урок 1
`class`	Пользовательский тип (класс), содержащий атрибуты и методы	См. урок 6
`continue`	Переход на очередную итерацию цикла	См. урок 1
`def`	Определение функции	См. урок 3
`del`	Удаление объекта	См. урок 7
`elif`	Ещё, иначе, если	См. урок 1
`else`	Иначе, если	См. урок 1
`except`	Перехват исключения	См. урок 2, 8
`finally`	Выполнение инструкций, независимо были ли исключение или нет	См. урок 2, 8
`for`	Начало цикла перебора элементов набора	См. урок 1
`from`	Указание пакета или модуля, из которого выполняется импорт	См. урок 3
`global`	Значение переменной, присвоенное ей внутри функции, становится доступным вне этой функции	См. урок 3
`if`	Если	См. урок 1
`import`	Импорт модуля	См. урок 4
`in`	Проверка на вхождение	См. урок 1
`is`	Проверка, ссылаются ли два объекта на одно и то же место в памяти	См. урок 1
`lambda`	Определение анонимной функции	См. урок 3
`nonlocal`	Значение переменной, присвоенное ей внутри функции, становится доступным в объемлющей функции	См. урок 3
`not`	Логическое «НЕ»	См. урок 1
`or`	Логическое «ИЛИ»	См. урок 1
`pass`	Заглушка для функции или класса. Используется, когда код класса и функции ещё не определён	См. урок 3
`raise`	Генерация исключения	См. урок 8
`return`	Вернуть результат	См. урок 3

`try`	Выполнить инструкции с перехватом исключения	См. урок 2, 8
`while`	Начало цикла «ПОКА»	См. урок 1
`with`	Использование менеджера контекста	См. урок 5
`yield`	Определение функции-генератора	См. урок 4

Лучшие онлайн-интерпретаторы Python

Для быстрой проверки работоспособности кода можно использовать онлайн-инструменты, например:

1. IdeOne.

Онлайн-интерпретатор, позволяющий напрямую в браузере проверять работу кода более, чем на 60 языках программирования.

2. Koding.

Готовая среда для разработки и тестирования. Включает виртуальную машину под управлением Ubuntu, среду разработки и различные предустановленные сервисы. Доступно напрямую из браузера.

3. PythonAnywhere.

Популярный сервис для запуска Python-скриптов в облаке. Доступна возможность хостинга разработанных проектов.

4. Codenvy.

Облачный онлайн-редактор для разработчиков, в том числе Python-программистов. Предусматривает специальные механизмы быстрого обмена файлами между участниками команды.

Практическое задание

1. Поработайте с переменными, создайте несколько, выведите на экран. Запросите у пользователя некоторые числа и строки и сохраните в переменные, затем выведите на экран.

2. Пользователь вводит время в секундах. Переведите время в часы, минуты, секунды и выведите в формате чч:мм:сс. Используйте форматирование строк.

3. Узнайте у пользователя число n. Найдите сумму чисел n + nn + nnn. Например, пользователь ввёл число 3. Считаем 3 + 33 + 333 = 369.

4. Пользователь вводит целое положительное число. Найдите самую большую цифру в числе. Для решения используйте цикл while и арифметические операции.

5. Запросите у пользователя значения выручки и издержек фирмы. Определите, с каким финансовым результатом работает фирма. Например, прибыль — выручка больше издержек, или убыток — издержки больше выручки. Выведите соответствующее сообщение.

   Если фирма отработала с прибылью, вычислите рентабельность выручки. Это отношение прибыли к выручке. Далее запросите численность сотрудников фирмы и определите прибыль фирмы в расчёте на одного сотрудника.

6. Спортсмен занимается ежедневными пробежками. В первый день его результат составил **a** километров. Каждый день спортсмен увеличивал результат на 10% относительно предыдущего. Требуется определить номер дня, на который результат спортсмена составит не менее **b** километров. Программа должна принимать значения параметров **a** и **b** и выводить одно натуральное число — номер дня.

   Например: a = 2, b = 3.
   Результат:
   1-й день: 2
   2-й день: 2,2
   3-й день: 2,42
   4-й день: 2,66
   5-й день: 2,93
   6-й день: 3,22
   Ответ: на шестой день спортсмен достиг результата — не менее 3 км.

Дополнительная литература

1. Настройка Python path.
2. Список всех операторов.

Используемая литература

Для подготовки методического пособия были использованы следующие ресурсы:

1. Язык программирования Python 3 для начинающих и чайников.
2. Программирование в Python.
3. Учим Python качественно (habr).
4. Самоучитель по Python.
5. Лутц М. Изучаем Python. — М.: Символ-Плюс, 2011 (4-е издание).