

# Курс повышения квалификации "Введение в ROS"

## 3 Python для роботов

### О python

**Python** — язык программирования общего назначения. Имеет библиотеки и фреймворки для разных направлений: веб-разработки (например, `Django` и `Bottle`), научных и математических вычислений (`Orange`, `SymPy`, `NumPy`) для настольных графических пользовательских интерфейсов (`Pygame`, `Panda3D`).

Работать на Python приятно и легко, потому что он позволяет сфокусироваться на задаче, а не продираться сквозь сложный синтаксис.

#### История Python

Разработка Python началась Гвидо Ван Россумом в конце 1980-х., и в феврале 1991 года вышла первая версия.

#### Зачем создан Python?

В конце 1980-ых, Гвидо Ван Россум работал над группой операционных систем Атобеа. Он хотел использовать интерпретируемый язык, такой как ABC. Язык имеющий простой и доступный в понимании синтаксис, который мог бы получить доступ к системным вызовам Атобеа. Поэтому он решил создать масштабируемый язык. Это привело к созданию нового языка, у которого позже появилось название Python.

#### Почему выбрали название Python

Нет. Он не назван в честь змеи. В 70-х Россум был фанатом комедийного сериала "Monty Python's Flying Circus" (Летающий цирк Монти Пайтона). "Python" было взято из названия.

#### Дата выпуска версий языка

Версия	Дата выпуска
Python 1.0 (первый стандартный выпуск) Python 1.6 (последняя выпущенная версия)	Январь 1994 Сентябрь 5, 2000
Python 2.0 (представлены списки) Python 2.7 (последняя выпущенная версия)	Октябрь 16, 2000 Июль 3, 2010
Python 3.0 (Сделан акцент на удаление дублирующих конструкций и модулей) Python 3.9 (Текущая релизная версия)	Декабрь 3, 2008 настоящее время

## Особенности программирования на Python

### 1. Язык простой, легкий и доступный в изучении

У Python читаемый синтаксис. Гораздо проще читать и писать программы на Python по сравнению с другими языками, такими как: C++, Java, C#. Python делает программирование интересным и позволяет сфокусироваться на решении, а не синтаксисе.

Для новичков, отличный выбор — начать изучение программирования с Python.

### 2. Бесплатный и с открытым кодом

Можно свободно использовать и распространять программное обеспечение, написанное на Python, даже для коммерческого использования. Вносить изменения в исходный код Python. Над Python работает большое сообщество, постоянно совершенствуя язык с каждой новой версией.

### 3. Портативность

Перемещайте и запускайте программы на Python с одной платформы на другую без каких-либо изменений кода.

Код работает практически на всех платформах, включая Windows, Mac OS X и Linux.

### 4. Масштабируемый и встраиваемый

Предположим, что приложение требует повышения производительности. Вы можете с легкостью комбинировать фрагменты кода на C/C++ и других языках вместе с кодом Python.

Это повысит производительность приложения, а также дает возможность написания скриптов, создание которых на других языках требует больше настроек и времени.

### 5. Высокоуровневый, интерпретируемый язык

В отличие от C/C++, вам не нужно беспокоиться о таких сложных задачах, как “сборка мусора” или управление памятью.

Так же, когда вы запускаете код Python, он автоматически преобразует ваш код в язык, который понимает компьютер. Не нужно думать об операциях более низкого уровня.

### 6. Стандартные библиотеки для решения общих задач

Python укомплектован рядом стандартных библиотек, что облегчает жизнь программиста, так как нет необходимости писать весь код самостоятельно. Например, что бы подключить базу данных MySQL на Web сервер, используйте библиотеку MySQLdb, добавляя ее в код строкой `import MySQLdb`.

Стандартные библиотеки в Python протестированы и используются тысячами людей. Поэтому будьте уверены, они будут работать именно так, как описано в документации.

### 7. Объектно-ориентированный

В Python все объект. Объектно-ориентированное программирование (ООП) в самой природе питона. Разделяйте сложные задачи на маленькие части, создавая объекты.

## Приложения на Python

### Веб-приложения

Создавайте масштабируемые веб-приложения, с помощью фреймворков и CMS (Систем управления содержимым) на Python. Популярные платформы для создания Web приложений: Django, Flask, Pyramid, Plone, Django CMS.

Кстати, сайты Mozilla, Reddit, Instagram и PBS написаны на Python.

## Научные и цифровые вычисления

У Python много библиотек для научных и математических вычислений. Есть библиотеки, такие как: `SciPy` и `NumPy` которые используются для общих вычислений. И специальные библиотеки, такие как: `EarthPy` для науки о Земле, `AstroPy` для астрономии и так далее.

Python язык номер один в машинном обучении, анализе и сборе данных и прототипировании нейросетей.

## Создание прототипов программного обеспечения

Python менее эффективен, в сравнении с компилированными языками, такими как C++ и Java. Это не очень практичный выбор, если ресурсы ограничены и при этом нужна максимальная эффективность. Тем не менее, Python — прекрасный язык для создания прототипов. Используйте Pygame (библиотека для создания игр), чтобы создать прототип игры. Если прототип понравился, используйте вставки на C/C++ для повышения производительности.

Python обладает рядом преимуществ, помогающим начинающим разработчикам начать писать код быстрее

### 1. Простой элегантный синтаксис

Программировать на Python интересно. Легче понять и написать код на Python. Почему? Синтаксис кажется естественным и простым. Возьмите этот код для примера:

```
1 a = 3
2 b = 5
3 sum = a + b
4 print(sum)
```

Даже если вы не программировали ранее, вы с легкостью поймете, что эта программа складывает две цифры и печатает их сумму на экране.

### 1. Не слишком строгий

Не нужно определять тип переменной в Python. Нет необходимости добавлять ";" в конце строки. Python принуждает следовать методам написания читаемого кода (например, одинаковым отступам). Эти мелочи могут значительно облегчить обучение новичкам.\

### 2. Выразительность языка

Python позволяет писать программы с большей функциональностью и с меньшим количеством строк кода. Вот ссылка на исходный код игры [Tic-tac-toe](#) с графическим интерфейсом и противником в лице смарт-компьютера менее чем на 500 строк кода. Это просто пример. Вы будете удивлены, как много можно сделать с Python, как только начнете изучать основы языка.\

### 3. Большое сообщество и поддержка

У Python большое сообщество с огромной поддержкой. Множество активных форумов в интернете, которые помогут, когда возникают вопросы. Вот некоторые из них:

- [Python на Хабре](#)
- [Вопросы о Python на Тостер](#)
- [Вопросы о Python на Stack Overflow](#)

позволяет писать программы с большей функциональностью и с меньшим количеством строк

кода. Вот ссылка на исходный код игры [Tic-tac-toe](#) с графическим интерфейсом\

#### 4. Большой выбор курсов для обучения

По Python есть огромное количество курсов в интернете, как платных так и бесплатных. Вот некоторые из них:

- <https://www.coursera.org/learn/python-osnovy-programmirovaniya>
- <https://stepik.org/course/67/syllabus>
- <https://skillbox.ru/course/profession-python/>
- <https://netology.ru/programs/python>
- <https://compscicenter.ru/courses/python/2015-autumn/classes/>

## ОСНОВЫ И СИНТАКСИС

### Среда разработки

Программы на Python можно писать в любом текстовом редакторе, главное соблюдать правильный синтаксис. Однако, мы для написания кода будем использовать [Visual Studio Code](#) которую мы установили при подготовке рабочего места. Это бесплатная удобная IDE (integrated development environment) - интегрированная среда разработки. Она удобна тем, что подсвечивает синтаксис кода программы, указывает на ошибки и может исполнять код внутри себя, вызывая встроенный интерпретатор Python. Кроме того у нее есть еще несколько удобных особенностей, о которых мы расскажем позже.

Написанный код на Python сохраняется в файлах с расширением `.py`

### Интерпретатор Python

Для того, чтобы превратить код написанный на языке Python в последовательность команд для процессора, нужно передать этот код интерпретатору Python. Специальной программе, которая превращает написанный и понятный человеку текст кода в команды машинного кода, понятного компьютеру.

Чтобы запустить программу на Python нужно вызвать интерпретатор Python из директории с программой и передать ему название файла с программой.

В текущий момент существует одновременно две версии языка Python (вторая и третья версия). Команда `python` вызывает выполнение интерпретатора второй версии, а команда `python3` третьей версии.

В нашем курсе мы будем использовать актуальную третью версию python.

### Создадим и запустим первую программу

Откройте Visual Studio Code. При первом запуске будет предложено некоторое количество настроек

интерфейса - выберите по своему усмотрению. Создайте новый файл, укажите язык программирования python. Программа предложит вам установить расширения для питона - соглашайтесь. После этого наберите текст первой программы:

```
1 >>> print ("Hello robot")
```

Сохраните файл с названием `test.py` в папке `ros_projects` которую мы создали в предыдущей главе.

Запустим первую программу. Для этого откроем терминал, зайдем в папку `ros_projects` и выполним команду:

```
1 user@userbox:~/ros_projects$ python3 test.py
```

Данная команда запустит программу из файла `test.py` и вы увидите результат ее выполнения - выведенную надпись Hello robot:

```
1 user@userbox:~/ros_projects$ python3 test.py
2 Hello robot
3 user@userbox:~/ros_projects$
```

Если система не выдала ошибок - значит у вас все настроено верно, и можно продолжать работать.

---

## Синтаксис Python

Стоит отметить основную особенность синтаксиса Python. Он не содержит операторных символов (begin..end в pascal или скобок {..} в Си), вместо этого **блоки выделяются отступами**: пробелами или табуляцией, а вход в блок из операторов осуществляется двоеточием. Однострочные комментарии начинаются с «`#`», многострочные — начинаются и заканчиваются тремя двойными кавычками «`"""`».

Пример кода на Python

```
1 a = 2
2 b = 3
3 c = a + b
4 print(c)
```

Код выше, если его запустить выведет в консоль число 5 .

```
1 user@userbox:~/ros_projects$ python3 test.py
2 5
3
```

```
user@userbox:~/ros_projects$
```

В целом синтаксис языка Python крайне прост и мы будем осваивать его по ходу изучения основных особенностей языка.

---

## Внешние модули

Для python написано большое количество внешних модулей реализующих ту или иную функциональность.

Для того чтобы установить модуль из общедоступной библиотеки используйте пакетный менеджер Python `pip`

Установим модуль `pip`

```
1 sudo apt install python3-pip
```

Теперь мы можем установить модуль для Python например `numpy` - модуль для математических вычислений (<https://numpy.org/>)

```
1 pip install numpy
```

В процессе установки вы увидите примерно такое сообщение:

```
1 user@userbox:~$ pip install numpy
2 Collecting numpy
3 Downloading https://files.pythonhosted.org/packages/3a/5f/47e578b3ae79e2624e205445ab77a1848
4 100% |████████████████████████████████████████████████████████████████████████████████| 17.0MB 78kB/s
5 Installing collected packages: numpy
6 Successfully installed numpy-1.16.6
```

---

## Запуск python программ

Очень часто нам может быть не удобно каждый раз при указывать имя интерпретатора для запуска программы. Для решения такой проблемы, есть стандартный механизм linux.

В начало `python` файла, первой строчкой необходимо указать используемый интерпретатор. Например файл (`test.py`):

```
1 #!/usr/bin/env python3
2
3 ~
```

```
3 a = 2
4 b = 3
5 c = a + b
6 print(c)
```

Такая строчка называется [Шебанг] ([https://ru.wikipedia.org/wiki/Шебанг\\_\(Unix\)](https://ru.wikipedia.org/wiki/Шебанг_(Unix))).

Также файлу необходимо установить специальный флаг, что файл может быть исполнен. Для этого существует утилита linux `chmod`

```
1 chmod +x test.py
```

После этих двух операций, мы можем запустить python скрипт, просто набрав в консоле имя скрипта `./test.py` без указания интерпретатора.

## Числа, переменные и операции над ними

### Целые числа

В общем случае программирование это написание заданий для компьютера, по обработке данных. Данные бывают разные, числа, текст, изображения, видео, звук и т.д. Но в конечном итоге все эти типы данных сводятся к числам. Изучение python мы и начнем с чисел.

Самый простой тип чисел это целые числа. Данный тип чисел в python и многих других языках называется `int`.

В python для целых чисел определены операции `+`, `-`, `*` и `**`, т.е. сложения, вычитания, умножения и возведения в степень. Производя эти операции над целыми числами мы получим целое число.

Однако, операции деления `/` для целых чисел и возведения в степень `**` возвращает вещественное число (значение типа `float`), если показатель степени — отрицательное число. О вещественных числах мы поговорим чуть позднее.

Но есть и специальная операция целочисленного деления, выполняющегося с отбрасыванием дробной части, которая обозначается `//`. Она возвращает целое число: целую часть частного. Другая близкая ей операция – это операция взятия остатка от деления, обозначаемая `%` тоже возвращает целое число.

### Действительные числа

Действительные числа это все числа числовой прямой - целые, дробные, иррациональные, отрицательные и положительные. В Python действительные числа имеют тип `float`. Они ограничены по числу знаков после запятой (не более 16), но для наших типовых задач данной точности вполне хватает.

Действительные (вещественные) числа представляются в виде чисел с десятичной точкой (а не запятой, как принято при записи десятичных дробей в русских текстах). Для записи очень больших или очень маленьких по модулю чисел используется так называемая запись «с плавающей точкой» (также называемая «научная» запись). В этом случае число представляется в виде некоторой десятичной дроби, называемой мантиссой, умноженной на целочисленную степень десяти (порядок). Например, расстояние от Земли до Солнца равно  $1.496 \cdot 10^{11}$ , а масса молекулы воды  $2.99 \cdot 10^{-23}$ .

Числа с плавающей точкой в программах на языке Питон, а также при вводе и выводе записываются так: сначала пишется мантисса, затем пишется буква `e`, затем пишется порядок. Пробелы внутри этой записи не ставятся. Например, указанные выше константы можно записать в виде `1.496e11` и `2.99e-23`. Перед самым числом также может стоять знак минус.

Напомним, что результатом операции деления `/` всегда является действительное число (`float`), в то время как результатом операции целочисленного деления `//` является целое число (`int`).

Преобразование действительных чисел к целому производится с округлением в сторону нуля, то есть `int(1.7) == 1`, `int(-1.7) == -1`.

## Операторы сравнения

Что можно делать с числами, кроме математических операций сложения, умножения, вычитания и т.д.? Числа можно сравнивать! Сравнение это очень понятная операция, ну очевидно же, что-то одно, больше или меньше чего-то другого. А что является результатом операции сравнения? Ведь в отличие скажем от того же сложения, где результатом является сумма, у сравнения нет нового значения. На самом деле результатом применения оператора сравнения к двум числам является значение Истина или Ложь. Давайте посмотрим это на примере оператора "Больше" `>`. Это самый простой и понятный оператор, это Что-то одно Больше чего-то другого. А поскольку все в Python объект, то один объект Больше другого. Логично предположить что сравнивать можно, только одинаковые по типу объекты. Сравнивать "мух" с "котлетами" - нельзя! А вот сравнивать длину строки "муха" с длиной строки "котлета" - можно! Но давайте к чему-то более понятному - к числам. Давайте сравним 5 и 3. Применим оператор `>` к паре чисел. И что мы получим, когда нажмем Enter?

```
1 >>> 5 > 3
```

А получим мы ответ в виде `True`, ту самую Истину. Т.е. Python используя формальную математическую логику вычислил результат операции сравнения `5 "Больше" 3`, и определил его как Истина. Ну а если к той же конструкции, мы применим оператор "Больше", но теперь поменяем местами наши цифры `<`



Результат будет следующим:

```
1 >>> 3 > 5
2 False
```

False, переводится на русский как Ложь. По сути это и есть два состояния, которые мы получаем применяя любой оператор сравнения к любым сравнимым объектам. Ложь или Истина. True или False.

**i** Кстати! Переменные, к которым применяются операции сравнения называются **операндами**.

Логично предположить, что если есть оператор сравнения "Больше", то есть и оператор сравнения "Меньше". И он действительно есть, а кроме того есть еще ряд операторов сравнения:

< Меньше — условие верно, если первый операнд меньше второго.

> Больше — условие верно, если первый операнд больше второго.

<= Меньше или равно.

>= Больше или равно.

== Равенство. Условие верно, если два операнда равны.

!= Неравенство. Условие верно, если два операнда неравны.

Например, условие `(x * x < 1000)` означает "значение `x * x` меньше 1000", а условие `(2 * x != y)` означает "удвоенное значение переменной `x` не равно значению переменной `y`".

Для того чтобы определить в каком порядке применять соответствующие операторы, например, в случае `2 * 2 + 2` существует таблица приоритета применения операторов, она очень похожа на привычные нам со школы правила приоритета математических операций. К примеру круглые скобки имеют самый большой приоритет и все что в них выполняется в первую очередь, потом идут операции возведения в степень, применения знака, умножения, затем сложения и вычитания, и в конце, с самым низким приоритетом выполняются операции сравнения. Для практического применения важно помнить, что скобки имеют высший приоритет, и если вы сомневаетесь в том в каком порядке будут выполняться операции - просто расставьте скобки. Например, так `2 * (2 + 2)`, вы тогда операция сложения выполнится первой, а умножение будет уже применяться к результату сложения и финальный результат будет 8, а если бы скобки не стояли, то результат конечно же был бы 6.

Операторы сравнения в Питоне можно объединять в цепочки (в отличии от большинства других языков программирования, где для этого нужно использовать логические связки), например, `1 == 2 == 3` или `1 <= x <= 10`.

## Переменные и оператор присваивания

**Переменная** — это математический объект, который занимает некоторое множество значений (как правило, числовых) и в его пределах может, как «ползунок», изменять своё значение. **Переменные** используются, в частности, в задании математических выражений. Понятие **переменной** широко используется в таких областях, как математика, естественные науки, техника и программирование. Примерами **переменных** могут служить: температура воздуха, параметр функции и многое другое.

Давайте на примерах. Предположим, что есть большое число: `384400`. Оно обозначает расстояние между Солнцем и Землей в километрах.


Предположим, что это значение мы хотим использовать в каких-то наших расчетах. И хотим использовать его достаточно часто. Вместо того, чтобы каждый раз вводить это число, мы можем просто сказать, что теперь каждый раз используя в программе выражение `moon_to_earth` мы имеем ввиду вот это число `384400`. Вот так:

```
1 >>> moon_to_earth = 384400
2 >>> moon_to_earth = moon_to_earth + 1
```

Эти данные могли бы быть и куда объемнее. В них, например, могло бы быть 100 цифр, а обращаться к данным, возможно, нужно было бы в 100 разных местах. Таким образом польза от присваивания имени значению очевидна.

Разберем код примера:

1. Сначала создается переменная `moon_to_earth`, и ей присваивается значение `384400`.

 Вот этот символ `=` оператор присваивания. Сам смысл использования переменных в том, что мы можем менять их значения - "присваивать" им какие-то новые значения. Для этого и используется оператор присваивания. Он берет какое-то выражение или значение справа от себя и присваивает его переменной слева от себя, таким образом, что после операции присваивания в этой переменной оказывается результат вычисления выражения или сразу какое-то значение как в нашем случае.

2. Выполняется какой-то расчет, а точнее увеличение числа `384400` на единицу. И теперь в переменной `moon_to_earth` новое значение — `384401`.

Советы:

1. Название переменной всегда расположено слева при присваивании. А значение, которое нужно присвоить, располагается справа.\
2. В Python (и любом другом языке программирования) очень важно правильно выбирать название переменной, ведь к ней придется часто обращаться:
  - Кое-какие названия являются зарезервированными в Python, поэтому их использовать нельзя: `print`, `true`, `false`, `count`, `sum` и так далее. Со временем вы все их узнаете.

- Названия переменных должны начинаться с символа алфавита или нижнего подчеркивания.
- Если начать переменную с символа или нижнего подчеркивания, то после в нее можно включать и цифры.
- Названия переменных чувствительны к регистру. Если был использован символ в верхнем регистре, то именно так и нужно обращаться к переменной в будущем.\

### 3. Практические соглашения именования переменных:

- Можно добавлять небольшой суффикс, чтобы не путать переменную с ключевым словом Python: `var_number`, `var_true`, `v_start`, `v_close`, `the_class`, `the_one`, `the_name`. Креативность разрешена, главное, чтобы название оставалось понятным.
- Лучше не использовать переменную в один символ, только если это цикл или любая другая короткая операция. Такие имена как `a`, `x`, `y`, `z` начинают путать уже спустя пару минут после того, как их впервые использовали.
- Слова рекомендуется разделять нижним подчеркиванием: `cars_counted`, `doors_closed`, `dogs_saved`, `ships_fixed` и так далее. Это повышает читаемость переменных. Сравните: `timeinmoscow` и `time_in_moscow`.
- Рекомендуется (часто требуется) использовать латиницу и английские слова. Так ваш код будут понимать больше разработчиков.

В одной строке можно присвоить сразу несколько переменных. Вот пример:

```
1 >>> i, j, k = "Hello", 55, 21.0765
```

## Ввод и вывод данных, преобразование типов

### Вывод данных

Для вывода данных в Питоне используется функция `print()`. Внутри круглых скобок через запятую мы пишем то, что хотим вывести на экран.

- i** Все что пишется внутри круглых скобок функции, называется передаваемым аргументом. Или просто аргументом функции. Это и есть, то что функция принимает на вход и с чем будет работать в дальнейшем.

*Подробнее про функции и их аргументы мы поговорим далее*

Вот программа, которая выводит на экран результат нескольких вычислений:

```
1 print(3 + 5)
2 print(2 * 9, (6 - 9) * 4)
3
```

```
3 print(3 ** 4) # две звёздочки (**) - оператор возведения в степень
4 print(69 / 4) # косая черта (/) - оператор деления
5 print(69 // 4) # две косые черты (//) - оператор возвращающий частное от деления нацело
6 print(57 % 7) # процент (%) - оператор остатка от деления нацело
```

## Ввод данных

Для ввода данных мы используем функцию `input()`. Она считывает все, что введет пользователь с клавиатуры до нажатия клавиши `Enter`.

Напишем простую программу, которая будет спрашивать у пользователя его имя и здороваться.

```
1 print('Как вас зовут?')
2 name = input() # считываем строку и "кладем" её в переменную name
3 print('Привет, ' + name + '!')
```

Теперь, попробуем написать программу, которая считывает два числа и выводит их сумму. Для этого считаем два числа и сохраним их в переменные `a` и `b`, пользуясь оператором присваивания `=`. Слева от оператора присваивания `=` в программах на Python ставится имя переменной (подробнее про операторы далее). Справа от оператора присваивания ставится любое выражение (значение, математическая формула, символ, строка, текст, результат работы функции, имя файла и т.д.). После выполнения операции присваивания, в нашем случае, переменная станет указывать на результат умножения переменных `a` и `b`.

- i На самом деле в питоне нет и переменных. Есть лишь имена, которые связаны с какими-нибудь объектами. Можно сначала связать имя с одним объектом, а потом — с другим. Можно несколько имён связать с одним и тем же объектом. Но в нашем курсе это не важно.

```
1 a = input("Введите a")
2 b = input("Введите b")
3 s = a + b
4 print(s)
```

Выполните эту программу и посмотрите на результаты её работы. Давайте в качестве запросов программы введем `3` и `8`.

Мы видим, что программа выводит `38`, хотя мы ожидали, что `3 + 8` будет `11`. Это произошло потому, что команда `input()` в Python, принимает весь введенный текст с клавиатуры как "текст", т.е. на выходе из команды `input()` мы получим введенные данные в формате `str` - String или строка. Таким образом в переменной `a` и `b` оказались текстовые строки, пусть даже и содержащие числа. Ну и как следствие, при операции присваивания, интерпретатор Python, рассчитал

выражение, которое он должен был присвоить переменной `s` , а именно сложил две строки, путем добавления первой ко второй и получил результат `"3" + "8" = "38"`  
В Питоне все данные называются объектами. Число 2 представляется объектом `число 2`, а строка `'Как вас зовут?'` – это объект `строка 'Как вас зовут?'`.

Каждый объект относится к какому-то типу. Строки хранятся в объектах типа `str` , целые числа хранятся в объектах типа `int` , дробные числа (вещественные числа) — в объектах типа `float` . Тип объекта определяет, какие действия можно делать с объектами этого типа. Например, если в переменных `first` и `second` лежат объекты типа `int` , то их можно перемножить, а если в них лежат объекты типа `str` , то их перемножить нельзя, но можно складывать, добавляя вторую к первой.

```
1 first = 8
2 second = 11
3 print(first * second)

4 first = '8'
5 second = '11'
6 print(first * second)
```

Чтобы преобразовать строку из символов цифр в объект число, воспользуемся встроенной в Python функцией `int()` . Подробнее про функции мы расскажем дальше, но сейчас мы просто воспользуемся ей, для этого "скажем" что взять в качестве ввода и куда передать вывод.

Например, `int('34')` преобразует два символа `34` в значение числа 34. Теперь, скажем интерпретатору Python куда передать полученное нашей функцией `int` , значение.

```
1 a = int('23')
```

Что же мы сделали? На ввод функции `int()` мы подали строку `"34"` , которую функция преобразовала в число 34, после чего интерпретатор сохранил в переменной `a` ссылку на это число. Или говоря проще (но не совсем корректно) присвоил переменной `a` - значение 34.

Как мы говорили ранее - в Python всё объект! А значит, в качестве ввода нашей функции, мы можем использовать, например, вывод другой функции, который тоже является объектом. Главное условие для корректной работы любого такого выражения, чтобы типы объектов совпадали с теми, которые ожидают функции на ввод.

Давайте переформатируем нашу первую программу и сделаем ее такой, какой мы хотели, чтобы она была изначально. А именно, чтобы она считывала два числа и выводила их сумму

```
1 a = int(input("Введите a"))
2 b = int(input("Введите b"))
3 s = a + b
4 print(s)
```

Обратите внимание, что если вы хотите считать с клавиатуры действительное число, то результат, возвращаемый функцией `input()` необходимо преобразовывать к типу `float`

```
1 x = float(input())print(x)
```

Чтобы получить на базе одного объекта другой объект другого класса, как-то ему соответствующий, нужно использовать функцию приведения. Имя этой функции совпадает с именем типа данных, к которому мы приводим объект. Например: `int` — класс для целых чисел. Перевод перевода целого числа в действительное осуществляется функцией `float()`. Или например есть строка "5", мы можем перевести строку в число 5, при помощи функции `int()`, вот так: `int("5")`. и т.д.

## Списки и строки

### Списки

Большинство программ работает не с отдельными переменными, а с набором переменных. Например, программа может обрабатывать информацию об учащихся класса, считывая список учащихся с клавиатуры или из файла, и при этом изменение количества учащихся в классе не должно требовать создания новых переменных вручную. Для того, чтобы хранить множество данных в одной переменной, применяются структуры данных, и первые структуры, которые мы будем с вами изучать это списки и строки.

И если со строками, мы уже немного познакомились, когда писали наши первые примеры, то списки это что-то новое. На самом деле исходя из названия "список", это некий единый объект, содержащий в себе записи каких-то других значений. Например список учеников в классе или список товаров в магазине.

Для хранения таких данных мы будем использовать структуру списка (в других языках программирования используется термин "массив"). Список представляет собой последовательность элементов, пронумерованных от 0. Список можно задать перечислением элементов произвольного типа в квадратных скобках, например, вот так:

```
1 Primes = [2, 3, 5, 7, 11, 13]
2 Rainbow = ['Red', 'Orange', 'Yellow', 'Green', 'Blue', 'Indigo', 'Violet']
```

В списке `Primes` — 6 элементов, а именно: `Primes[0] == 2`, `Primes[1] == 3`, `Primes[2] == 5`, `Primes[3] == 7`, `Primes[4] == 11`, `Primes[5] == 13`. Список `Rainbow` состоит из 7 элементов, каждый из которых является строкой.

Также как и символы в строке, элементы списка можно индексировать отрицательными числами с конца, например, `Primes[-1] == 13`, `Primes[-6] == 2`.

- ① Длину списка, то есть количество элементов в нем, можно узнать при помощи функции `len()`, например, `len(Primes) == 6`

Элементы списка можно изменять, присваивая им новые значения.

## Строки

Строка состоит из последовательности символов. На самом деле каждая строка, с точки зрения Питона, — это объект класса `str`. Любой базовый тип данных в Питоне можно перевести к строке, которая ему соответствует. Для этого нужно вызвать функцию `str()`, передав ей в качестве аргумента объект, переводимый в строку.

- ① Узнать длину строки, а так же количество элементов в списке можно при помощи функции `len()` передав ей в качестве аргумента саму строку.

*подробнее о функциях мы поговорим дальше*

```
1 s = input()
2 print(len(s))
3 t = input()
4 number = int(t)
5 u = str(number)
6 print(s * 3)
7 print(s + ' ' + u)
```

---

## Срезы (slices)

Срез (slice) — извлечение из данной строки одного символа или некоторого фрагмента подстроки или подпоследовательности.

Есть три формы срезов. Самая простая форма среза: взятие одного символа строки, а именно, `S[i]` — это срез, состоящий из одного символа, который имеет номер `i`. При этом считается, что нумерация начинается с числа 0. То есть если `S = 'Hello'`, то `S[0] == 'H'`, `S[1] == 'e'`, `S[2] == 'l'`, `S[3] == 'l'`, `S[4] == 'o'`.

Заметим, что в Питоне нет отдельного типа для символов строки. Каждый объект, который получается в результате среза `S[i]` — это тоже строка типа `str`.

Номера символов в строке (а также в других структурах данных: списках, кортежах) называются *индексом*.

Если указать отрицательное значение индекса, то номер будет отсчитываться с конца, начиная с номера `-1`. То есть `S[-1] == 'o'`, `S[-2] == 'l'`, `S[-3] == 'l'`, `S[-4] == 'e'`, `S[-5] == 'H'`.

Или в виде таблицы:

Строка S	H	e	l	l	o
Индекс	<code>S[0]</code>	<code>S[1]</code>	<code>S[2]</code>	<code>S[3]</code>	<code>S[4]</code>
Индекс	<code>S[-5]</code>	<code>S[-4]</code>	<code>S[-3]</code>	<code>S[-2]</code>	<code>S[-1]</code>

Если же номер символа в срезе строки `S` больше либо равен `len(S)`, или меньше, чем `-len(S)`, то при обращении к этому символу строки произойдет ошибка `IndexError: string index out of range`.

Срез с двумя параметрами: `S[a:b]` возвращает подстроку из `b - a` символов, начиная с символа с индексом `a`, то есть до символа с индексом `b`, не включая его. Например, `S[1:4] == 'ell'`, то же самое получится если написать `S[-4:-1]`. Можно использовать как положительные, так и отрицательные индексы в одном срезе, например, `S[1:-1]` — это строка без первого и последнего символа (срез начинается с символа с индексом 1 и заканчивается индексом -1, не включая его).

При использовании такой формы среза ошибки `IndexError` никогда не возникает. Например, срез `S[1:5]` вернет строку `'ello'`, таким же будет результат, если сделать второй индекс очень большим, например, `S[1:100]` (если в строке не более 100 символов).

Если опустить второй параметр (но поставить двоеточие), то срез берется до конца строки. Например, чтобы удалить из строки первый символ (его индекс равен 0), можно взять срез `S[1:]`. Аналогично если опустить первый параметр, то можно взять срез от начала строки. То есть удалить из строки последний символ можно при помощи среза `S[:-1]`. Срез `S[:]` совпадает с самой строкой `S`.

Любые операции среза со строкой создают новые строки и никогда не меняют исходную строку. В Питоне строки вообще являются неизменяемыми. Можно лишь в "старую" переменную указывающую на строку присвоить новую строку.

## Логические операторы, условия и ветвления

### Ход выполнения программы и условия

Программа на питоне выполняется последовательно. Сверху вниз. Т.е. сначала выполняется первая строчка, потом вторая и т.д.



Например:

```
1 a = 5
2 b = 4
3 c = a + b
4 print(c)
```

Т.е. сначала создается переменная `a` и в нее присваивается значение 5, потом создается переменная `b` и в нее присваивается значение 4, потом создается переменная `c` и в нее присваивается результат сложения переменных `a` и `b`, затем значение переменной `c` выводится на экран. Все понятно, все последовательно.

Но не всегда программы должны выполняться последовательно и прямолинейно. Часто мы хотим, чтобы компьютер или робот вели себя в разных ситуациях по-разному. Объезжали препятствия, если они появятся на пути. Включали свет, если становится темно. Подавали сигнал, пока сдают назад. И т.д.

Т.е. делали что-то, при каких-то условиях. В терминологии языков программирования, мы хотим, чтобы программы поддерживали условные ветвления. Т.е. выполнялись по тем или иным веткам кода, в зависимости от условий.

Как и в обычном человеческом языке в Python, условия, при которых программа должна делать "что-то другое" регулируются условными операторами:

`if` - если

`else` - в другом случае

`while` - пока

Выражение, которое мы должны написать в питоне, для того, чтобы наша программа учитывала какие-то условия выглядит так:

```
1 если (условие):
2     делать то-то и то-то
3 если условие не выполняется:
4     то делать то-то и то-то
```

или

```
1 if (condition):
2     какой-то код
3 else:
4     какой-то другой код
```

или на диаграмме



В таком случае при проверке условия программа пойдет выполняться по тому или иному пути в зависимости от результата вычисления условия.

## Ветвления

Давайте поговорим что-такое условие, которое мы проверяем, для определения по какому пути пойдет наша программа. Условие это параметр условного оператора, который перенаправляет поток выполнения нашей программы по тому или иному пути в зависимости от того, что является результатом проверки условия. Фактически, результатом может быть только 2 значения, это уже известные нам Истина или Ложь, true или false. А для проверки условия используются операторы сравнения, результатом которых и является true или false.

Давайте пример,

```
1 if (5 > 3):
2     print("Условие == true")
3     print("Основной путь ветвления")
4 else:
5     print("Условие == false")
6     print("Альтернативный путь ветвления")
```

Т.е. когда мы подставим  $5 > 3$  как условие в конструкцию `if()`., то интерпретатор Python перенаправит выполнение программы по основному пути ветвления. А если бы условие было бы ложью, то стал бы выполняться альтернативный путь ветвления.

Именно этот принцип используется при управлении потоком выполнения программы в питоне. Если условие истинно - то основной путь выполнения, если ложно, то альтернативный.

## Логические операторы

Иногда, чтобы определить верный путь ветвления хода функции нужно проверить одновременно не одно, а несколько условий. Например, проверить, является ли данное число четным? Это можно сделать при помощи условия `(n % 2 == 0)` (остаток от деления `n` на `2` равен `0`), а если необходимо проверить, что два данных целых числа `n` и `m` являются четными, необходимо проверить справедливость обоих условий: `n % 2 == 0` и `m % 2 == 0`, для чего их необходимо объединить при помощи оператора `and` (логическое И): `n % 2 == 0 and m % 2 == 0`.

В Питоне существуют стандартные логические операторы: логическое И, логическое ИЛИ, логическое НЕ (отрицание).

Логическое И является бинарным оператором (то есть оператором с двумя операндами: левым и правым) и называется `and`. Оператор `and` возвращает `True` тогда и только тогда, когда оба его операнда имеют значение `True`.

Логическое ИЛИ является бинарным оператором, называется `or` и возвращает `True` тогда, когда хотя бы один операнд равен `True`.

Логическое НЕ (отрицание) является унарным оператором, то есть оператором с одним операндом и называется `not`, за которым следует единственный операнд. Логическое НЕ возвращает `True`, если операнд равен `False` и наоборот.

## Циклы

### Цикл while

Циклы это, повторяющиеся куски кода, которые будут выполняться пока будет истинно условие их выполнения. Это очень удобно. Действительно, мы можем заставить компьютер считать что-то, пока не получим результат, или считывать какую-то последовательность ввода с клавиатуры, пока не будет введен стоп-символ или вообще, делать что-то за циклено до тех пор пока мы не будем удовлетворены результатом.

Цикл `while` (“пока”) позволяет выполнить одну и ту же последовательность действий, пока проверяемое условие истинно. Условие записывается до тела цикла и проверяется до выполнения тела цикла. Как правило, цикл `while` используется, когда невозможно определить точное значение количества проходов исполнения цикла.

Синтаксис цикла `while` в простейшем случае выглядит так:

```
1 while условие:
2     блок инструкций
```

При выполнении цикла `while` сначала проверяется условие. Если оно ложно, то выполнение цикла прекращается и управление передается на следующую инструкцию после тела цикла `while`. Если условие истинно, то выполняется код внутри цикла, после чего условие проверяется снова и снова выполняется инструкция. Так продолжается до тех пор, пока условие будет истинно. Как только условие станет ложно, работа цикла завершится и управление передастся следующей инструкции после цикла.

Очевидно, что в случае если условие никак не зависит от результатов работы цикла, или программы в целом, то цикл будет выполняться "бесконечно".

Например, следующий фрагмент программы напечатает на экран квадраты всех целых чисел от 1 до 10. Видно, что цикл `while` может заменять цикл `for ... in range(...)`

```
1 i = 1
2 while i <= 10:
3     print(i ** 2)
4     i += 1
```

В этом примере переменная `i` внутри цикла изменяется от 1 до 10. Такая переменная, значение которой меняется с каждым новым проходом цикла, называется счетчиком. Заметим, что после выполнения этого фрагмента значение переменной `i` будет равно 11, поскольку именно при `i == 11` условие `i <= 10` впервые перестанет выполняться.

## Цикл for

Цикл `for`, также называемый циклом с параметром, в языке Питон богат возможностями, однако не совсем похож на циклы `for` из других языков программирования. В цикле `for` указывается переменная и множество значений, по которому будет пробегать переменная. Множество значений может быть задано списком, кортежем, строкой или диапазоном.

- ❗ Множество значений цикла `for`, в любом случае должно задаваться итерируемым множеством или объектом. Т.е. таким множеством, по элементам которого можно последовательно проходить с каждой итерацией. А вот как задать количество итераций числом (или выражением), об этом далее в описании функции `range()`

Вот простейший пример использования цикла, где в качестве множества значений используется кортеж

```
1 i = 1
2 for color in 'red', 'orange', 'yellow', 'green', 'cyan', 'blue', 'violet':
3     print('#', i, ' color of rainbow is ', color, sep = '')
4     i += 1
```

В этом примере переменная `color` последовательно принимает значения `'red'`, `'orange'` и т.д. В теле цикла выводится сообщение, которое содержит название цвета, то есть значение переменной `color`, а также номер итерации цикла число, которое сначала равно 1, а потом увеличивается на один (инструкцией `i += 1` с каждым проходом цикла).

Инструкция `i += 1` эквивалентна конструкции `i = i + 1` (это просто сокращенная запись). Такую сокращенную запись можно использовать для всех арифметических операций: `*`, `-`, `/`, `%` ...

В списке значений могут быть выражения различных типов, например

```
1 for i in 1, 2, 3, 'one', 'two', 'three':
2     print(i)
```

При первых трех итерациях цикла переменная `i` будет принимать значение типа `int`, при последующих трех — типа `str`.

## 2. Функция range

Как правило, циклы `for` используются либо для повторения какой-либо последовательности действий заданное число раз, либо для изменения значения переменной в цикле от некоторого начального значения до некоторого конечного.

Для повторения цикла некоторое заданное число раз `n` можно использовать цикл `for` вместе с функцией `range`

```
1 for i in range(4): # равносильно инструкции
2     for i in 0, 1, 2, 3: # здесь можно выполнять циклические действия
3         print(i)
4     print(i ** 2) # цикл закончился, поскольку закончился блок с отступом
5 print('Конец цикла')
```

В качестве `n` может использоваться числовая константа, переменная или произвольное арифметическое выражение (например, `2 ** 10`). Если значение `n` равно нулю или отрицательное, то тело цикла не выполнится ни разу.

Функция `range` может также принимать не один, а два параметра. Вызов `range(a, b)` означает, что индексная переменная будет принимать значения от `a` до `b - 1`, то есть первый параметр функции `range`, вызываемой с двумя параметрами, задает начальное значение индексной переменной, а второй параметр — первое значение, которое индексная переменная принимать **не будет**. Если же `a ≥ b`, то цикл не будет выполнен ни разу. Например, для того, чтобы просуммировать значения чисел от 1 до `n` можно воспользоваться следующей программой

```
1 sum = 0
2 n = 5
3
```

```
4 for i in range(1, n + 1):
    sum += i
5     print(sum)
```

В этом примере переменная `i` принимает значения 1, 2, ..., `n`, и значение переменной `sum` последовательно увеличивается на указанные значения.

Наконец, чтобы организовать цикл, в котором индексная переменная будет уменьшаться, необходимо использовать функцию `range` с тремя параметрами. Первый параметр задает начальное значение индексной переменной, второй параметр — значение, до которого будет изменяться индексная переменная (не включая его!), а третий параметр — величину изменения индексной переменной. Например, сделать цикл по всем нечетным числам от 1 до 99 можно при помощи функции `range(1, 100, 2)`, а сделать цикл по всем числам от 100 до 1 можно при помощи `range(100, 0, -1)`.

Более формально, цикл `for i in range(a, b, d)` при `d > 0` задает значения индексной переменной `i = a`, `i = a + d`, `i = a + 2 * d` и так для всех значений, для которых `i < b`. Если же `d < 0`, то переменная цикла принимает все значения `i > b`.

## Функции

## Функции

Функции — это такие изолированные участки кода, которые выполняются только тогда, когда это необходимо. Для того, чтобы указать интерпретатору Python, на те участки кода, которые надо выполнять обособленно, нужно определить их как функции. У функции есть имя (обычно), перечень аргументов, которые она принимает (иногда), и значение которое она возвращает (не часто).

Вы уже встречались с функциями `sqrt()`, `len()` и `print()`. Они все обладают общим свойством: они могут принимать аргументы (ноль, один или несколько), и они могут возвращать значение (хотя могут и не возвращать). Например, функция `sqrt()` принимает один аргумент и возвращает значение (корень числа). Функция `print()` принимает переменное число аргументов и ничего не возвращает.

Указание интерпретатору, что данный участок кода надо считать функцией и выполнять при определенных условиях, называется объявлением функции. В Python функции объявляются следующим образом:

```
1 def function_name():
2     // какой-то изолированный код
```

А как сделать так, чтобы интерпретатор выполнил этот участок кода - эту функцию? Указание

интерпретатору выполнять функцию, называется вызовом этой функции и в Python выглядит вот так:

```
1 function_name()
```

Разумеется, функцию надо сначала определить и только потом вызвать. Иначе попытка вызова неопределенной функции, даст ошибку:

```
1 >>> function_name()
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'my_function' is not defined
```

Давайте определим простейшую функцию, а затем вызовем ее:

```
1 def function_print_hello_robot():
2     print("Hello robot")
3     return True
```

Функция имеет имя `function_print_hello_robot`, так как в круглых скобках ничего не написано, то никаких аргументов, она не принимает. И так как в конце текста функции (тела функции) написано `return`, она возвращает значение `True`

**i** Вы уже слышали про отступы. После объявления имени функции и ее аргументов, после двоеточия, команды функции должны быть выделены отступом и TAB или пробелов, на равное расстояние от края.

```
1 def function_name():
2     <Пробелы>print("some text")
```

Или

```
1 def function_name():
2     < TAB >print("some text")
```

Теперь давайте вызовем ее и посмотрим, что она делает:

```
1 Hello robot
```

Функция просто печатает, Hello robot. А вот значение True, она не печатает. Она его возвращает. А куда? Ну если мы вызвали функцию просто указав ее имя, то никуда, а вот если мы присвоили то, что она возвращает какой-то переменной, то значение True, будет передано этой переменной.

```
1 def function_print_hello_robot():
2
```

```
3 print("Hello robot")
4
5 a = function_print_hello_robot()
6 print(a)
```

И вот теперь вывод такой программы будет следующим:

```
1 Hello robot
2 True
```

Давайте разберемся с аргументами функции. Они передаются функции для использования в момент ее вызова, но для этого нам надо указать, как именно функция будет их использовать. Давайте перепишем нашу функцию:

```
1 def function_print_hello_robot(robot_name):
2     print("Hello! From robot ", robot_name)
3     return True
```

Теперь мы передаем в нашу функцию как аргумент некое имя робота, который и будет с нами здороваться.

Давайте вызовем эту функцию и передадим ей как аргумент имя одного знаменитого робота:

```
1 def function_print_hello_robot(robot_name):
2     print("Hello! From robot ", robot_name)
3     return True
4
5 function_print_hello_robot("Fyodor")
```

Как и ожидалось, вывод функции будет следующим:

```
1 Hello! From robot Fyodor!
```

## Область видимости. Глобальные и локальные переменные.

Функции в своей работе активно используют переменные. Но для того, чтобы функции не мешали друг другу в своей работе, переменные объявленные внутри одной функции не видны внутри другой функции. К примеру, объявим две функции и попытаемся получить доступ из одной функции к переменным другой:

```
1 def function1():
2
```



```
3     print(a)
4
5 def function2():
6     print(a)
7
8 function1()
9 function2()
```

Мы получим примерно такую ошибку:

```
1 Traceback (most recent call last):
2   File "c:/projects/test.py", line 9, in <module>
3     function2()
4   File "c:/projects/test.py", line 6, in function2
5     print(a)
6 NameError: name 'a' is not defined
```

Таким образом каждая функция имеет свою область видимости. Т.е. те переменные которые она может использовать в своей работе.

Но, что если я хочу использовать значения переменных из одной функции в другой функции. В таком случае, одним из методов является использования глобальных переменных - т.е. таких переменных, которые не ограничены областью видимостью какой-то одной функции, а доступны всем функциям программы, например:

```
1 def function1():
2     global a
3     a = 5
4     print(a)
5
6 def function2():
7     global a
8     print(a)
9
10 function1()
11 function2()
```

Использование таких глобальных переменных объявляется внутри функций, через ключевое слово `global`. И как показано в примере выше, при запуске, такая программа выдаст:

```
1 5
2 5
```

Подробнее про использование глобальных и локальных переменных мы расскажем в следующем модуле: Базовый курс ROS.

# Стандартная библиотека Python

## Стандартные функции

Стандартная библиотека Python богата различными функциями, для работы с данными, сетевыми протоколами, операционной системой, файлами, XML, регулярными выражениями и прочим. Собственно, это одна из причин по которой Python так популярен. Зачастую, не надо писать какой-то алгоритм с нуля, можно взять готовую функцию из стандартной библиотеки или из большого количества сторонних модулей. Но это десятки встроенных функций и классов, сотни инструментов, входящих в [стандартную библиотеку Python](#), и тысячи сторонних библиотек на [PyPI](#). Держать всё в голове начинающему программисту нереально. Сейчас мы быстро перечислим основные функции, которыми мы будем пользоваться в нашем курсе.

Начнем с уже известных нам `print()`, `len()`, `str()`, `int()`, `float()`, надеюсь вы их помните и повторять мы их не будем.

`list`

Эта функция может очень облегчить задачу, если вы хотите составить список из итераций цикла.

```
1 >>> numbers = [2, 1, 3, 5, 8]
2 >>> squares = (n**2 for n in numbers)
3 >>> squares
4 at 0x7fd52dbd5930>
5 >>> list_of_squares = list(squares)
6 >>> list_of_squares
7 [4, 1, 9, 25, 64]
```

При работе со списком метод `copy` позволяет создать его копию.

```
1 >>> copy_of_squares = list_of_squares.copy()
```

Если вы не знаете, с какими элементами работаете, функция `list` является более общим способом перебора элементов и их копирования.

```
1 >>> copy_of_squares = list(list_of_squares)
```

`tuple`

Эта функция во многом похожа на функцию `list`, за исключением того, что вместо списков она создает кортежи.

```
1 >>> numbers = [2, 1, 3, 4, 7]
2 >>> tuple(numbers)
3 (2, 1, 3, 4, 7)
```

Если вы пытаетесь создать хешируемую коллекцию (например, ключ словаря), стоит отдать предпочтению кортежу вместо списка.

dict

Эта функция создаёт новый словарь.

Подобно спискам и кортежам, `dict` эквивалентна проходу по массиву пар «ключ-значение» и созданию из них словаря.

Дан список кортежей, по два элемента в каждом.

```
1 >>> color_counts = [('red', 2), ('green', 1), ('blue', 3), ('purple', 5)]
```

Выведем его на экран с помощью цикла.

```
1 >>> colors = {}
2 >>> for color, n in color_counts:
3 ...     colors[color] = n
4 ...
5 >>> colors
6 {'red': 2, 'green': 1, 'blue': 3, 'purple': 5}
```

То же самое, но с использованием `dict`.

```
1 >>> colors = dict(color_counts)
2 >>> colors
3 {'red': 2, 'green': 1, 'blue': 3, 'purple': 5}
```

range

Эта функция создаёт объект `range`, который представляет собой диапазон чисел.

```
1 >>> range(10_000)
2 range(0, 10000)
3 >>> range(-1_000_000_000, 1_000_000_000)
4 range(-1000000000, 1000000000)
```

Результирующий диапазон чисел включает начальный номер, но исключает конечный (`range(0, 10)` не включает 10).

Данная функция полезна при переборе чисел.

```
1 >>> for n in range(0, 50, 10):
2 ...     print(n)
3 ...
4 0
5 10
6 20
7 30
8 40
```

Обычный вариант использования — выполнить операцию `n` раз.

```
1 first_five = [get_things() for _ in range(5)]
```

Функция `range` в Python возвращает список.

`bool`

Эта функция проверяет достоверность (истинность) объектов Python. Относительно чисел будет выполняться проверка на неравенство нулю.

```
1 >>> bool(5)
2 True
3 >>> bool(-1)
4 True
5 >>> bool(0)
6 False
```

Применяя `bool` к коллекциям, будет проверяться их длина (больше 0 или нет).

```
1 >>> bool('hello')
2 True
3 >>> bool('')
4 False
5 >>> bool(['a'])
6 True
7 >>> bool([])
8 False
9 >>> bool({})
10 False
11 >>> bool({1: 1, 2: 4, 3: 9})
12 True
13 >>> bool(range(5))
14 True
15
```

```
16 raise bool(range(0))
17 >>> bool(None)
18 False
```

Проверка истинности очень важна в Python. Вместо того, чтобы задавать вопросы о длине контейнера, многие новички задают проверку истинности.

```
1 # Вместо этого
2 if len(numbers) == 0:
3     print("The numbers list is empty")
4
5 # многие делают так
6 if not numbers:
7     print("The numbers list is empty")
```

Данная функция используется редко. Но, если нужно привести значение к логическому типу для проверки его истинности, `bool` вам необходима.

### sum

Эта функция берёт набор чисел и возвращает их сумму.

```
1 >>> sum([2, 1, 3, 4, 7])
2 17
```

В Python есть много вспомогательных функций, которые выполняют циклы за вас (отчасти потому, что они хорошо сочетаются с генератор-выражениями).

```
1 >>> numbers = [2, 1, 3, 4, 7, 11, 18]
2 >>> sum(n**2 for n in numbers)
3 524
```

### min и max

Эти функции выдают минимальное и максимальное число из набора соответственно.

```
1 >>> numbers = [2, 1, 3, 4, 7, 11, 18]
2 >>> min(numbers)
3 1
4 >>> max(numbers)
5 18
```

Данные методы сравнивают элементы, используя оператор `<`. Поэтому, все передаваемые в них значения должны быть упорядочены и сопоставимы друг с другом.

type

Эта функция возвращает тип объекта, который вы ей передаете.

Тип экземпляра класса есть сам класс.

```
1 >>> x = [1, 2, 3]
2 >>> type(x)
3 <class 'list'>
```

Это основные, но далеко не все функции стандартной библиотеки Python, читайте примеры кода и вы встретитесь с еще большим количеством функций.

Наш ознакомительный курс по Python закончен. Мы ни в коем случае не претендуем на его полноту, мы лишь коснулись Python, и то, только в той практической части которая понадобится нам для дальнейшего обучения робототехнике. Продолжаете изучать Python самостоятельно и это поможет вам достигать новых вершин и в робототехнике!

## Модуль math

Для проведения вычислений с действительными числами Python содержит много дополнительных функций, собранных в библиотеку (модуль), которая называется `math`.

Для использования этих функций в начале программы необходимо подключить математическую библиотеку, что делается командой

```
1 import math
```

Например, пусть мы хотим округлять вещественные числа до ближайшего целого числа вверх. Соответствующая функция `ceil` от одного аргумента вызывается, например, так: `math.ceil(x)` (то есть явно указывается, что из модуля `math` используется функция `ceil`). Вместо числа `x` может быть любое число, переменная или выражение. Функция возвращает значение, которое можно вывести на экран, присвоить другой переменной или использовать в выражении:

```
1 import math
2 x = math.ceil(4.2)
3 y = math.ceil(4.8)
4 print(x)
5 print(y)
```

Другой способ использовать функции из библиотеки `math`, при котором не нужно будет при каждом использовании функции из модуля `math` указывать название этого модуля, выглядит так:

```
1 from math import ceil
```

```
2 x = 7 / 2
3 y = ceil(x)
4 print(y)
```

или так:

```
1 from math import *
2 x = 7 / 2
3 y = ceil(x)
4 print(y)
```

Ниже приведен список основных функций модуля `math`. Более подробное описание этих функций можно найти на [сайте с документацией языка Питон](#).

Некоторые из перечисленных функций (`int`, `round`, `abs`) являются стандартными и не требуют подключения модуля `math` для использования.

## Основы ООП

Объектно-ориентированное программирование (ООП) — это парадигма программирования, где различные компоненты компьютерной программы моделируются на основе реальных объектов.

**Объект** — это что-либо, у чего есть какие-либо характеристики и то, что может выполнить какую-либо функцию. При этом сама программа создается как некоторая совокупность объектов, которые взаимодействуют друг с другом и с внешним миром. Каждый **объект** является экземпляром некоторого **класса**.

### Создание классов

Для создания класса необходимо прописать ключевое слово `class` и далее название для класса. Общепринято начинать названия классов с буквы в верхнем регистре, но если этого не сделать, то ошибки не будет.

В любом классе можно создавать поля (переменные), методы (функции), а также конструкторы.

Создав новый класс и поместив туда какую-либо информацию мы можем создавать на основе него новые объекты. Объекты будут иметь доступ ко всем характеристикам класса.

Пример простого класса приведен ниже:

```
1 class Book:
2     pass # Класс может ничего не возвращать
```

Класс состоит из объявления (инструкция `class`), имени класса (нашем случае это имя *Book*) и тела класса, которое содержит **атрибуты** и **методы** (в нашем минимальном классе есть только одна инструкция `pass`).

На основе такого класса мы можем создать множество объектов. Каждый объект в данном случае будет представлять из себя конкретную книжку. Для каждого объекта мы можем указать уникальные данные.

Чтобы создать объект нам потребуется следующий код:

```
1 book_new = Book() # Создание объекта
2 book_second = Book() # Создание 2 объекта
```

Рассмотрим более сложный пример

```
1 class Person:
2     name = "Ivan"
3     age = 10
4
5     def set(self, name, age):
6         self.name = name
7         self.age = age
8
```

Переменные `name` и `age` в классе называются атрибуты класса. Функция `set()` методом класса.

Мы можем создать несколько объектов, и продемонстрировать как это все работает

```
1 vlad = Person()
2 vlad.set("Влад", 25)
3 print (vlad.name + " " + str(vlad.age))
4
5 ivan = Person()
6 ivan.set("Иван", 56)
7 print (ivan.age)
```

### Конструктор класса — метод `__init__`

Большинство классов имеют специальный метод, который автоматически запускается при создании объекта.

**Такой метод называется конструктором класса** и в языке программирования Python носит имя `__init__`. (В начале и конце по два знака подчеркивания.)

Первым параметром, как и у любого другого метода, у `__init__` является `self`, на место которого подставляется объект в момент его создания. Второй и последующие (если есть) параметры заменяются аргументами, переданными в конструктор при вызове класса. Ранее мы рассмотрели пример без конструктора, теперь перепишем наш класс с использованием конструктора.



```

2 class Person:
3     name = "Ivan"
4     age = 10
5     def __init__(self, name, age):
6         self.name = name
7         self.age = age
8

```

И примеры создания экземпляра класса с использованием конструктора.

```

1 vlad = Person ("Влад", 25)
2 print (vlad.name + " " + str(vlad.age))

```

### Что такое `_self_`?

До этого момента вы уже успели познакомиться с ключевым словом `self`. `self` – это ссылка на текущий экземпляр класса. В таких языках как *Java*, *C#* аналогом является ключевое слово *this*. Через `self` вы получаете доступ к атрибутам и методам класса внутри него самого:

```

1 class Person:
2     name = "Ivan"
3     age = 10
4
5     def __init__(self, name, age):
6         self.name = name
7         self.age = age
8
9     def print_data(self):
10         print (self.name + " " + str(self.age))
11

```

Пример использования

```

1 vlad = Person ("Влад", 25)
2 vlad.print_data()

```

## Концепции ООП

Мы разберем три основных концепции ООП. И хотя мы не будем в полной мере их использовать в рамках нашего курса, тем не менее знать об их существовании необходимо.

### Наследование

Наследование является одним из ключевых понятий ООП. За счёт наследования можно создать один общий класс (класс родитель) и создать множество других классов (классы наследники), что будут наследовать все поля, методы и конструкторы из главного класса.

### За чем использовать наследование?

Предположим что у нас есть один большой класс «Транспорт». В классе описываются базовые характеристики для всех транспортных средств:

- поля: скорость, вес, запас хода и тому подобное;
- методы: получение информации из полей, установка новых значений;
- конструктор: пустой и по установке всех полей.

На основе класса мы спокойно можем создать объект легковой машины, объект грузовика, объект самолета и так далее. У всех объектов будут одинаковые характеристики и методы.

Мы явно понимаем, что у объекта машина и самолёт будут разные поля и характеристики. Как можно поступить:

1. Можно создать два отдельных класса: «Car» и «Airplane». В каждом классе будут все методы, поля и конструкторы повторно переписанные из класса «Транспорт», а также будут новые методы, что важны только для конкретного класса;
2. Можно создать два класса наследника: «Car» и «Airplane». Оба класса будут наследовать всё от класса «Транспорт» и при этом будут содержать свои дополнительные функции. Таким образом повторения кода не будет и код станет меньше и чище.

### Создание классов наследников

Для создания класса наследника требуется создать класс и указать наследование от главного класса.

Пример класса наследника:

```
1 class Cars:
2     wheels = 4 # Общее значение для всех объектов,
3     # так как все машины имеют колеса
4
5 class BMW (Cars):
6     is_m_series = true # Является ли модель "М" серии?
7     # Переменная нужна только в классе BMW
```

### Инкапсуляция

Инкапсуляция позволяет ограничить доступ к какой-либо функции в классе. Благодаря такому подходу злоумышленники или же мы сами не сможем случайно или намерено вызвать или изменить метод.

Пример:

```
1 class Some:
2     def __printWords(self):
3         print ("Попробуй меня вызвать")
4
5 obj = Some()
6 obj.__printWords() # Вызов функции ничего не даст
```

Если метод класса начинается с `__` то такой метод или является приватным и его не возможно вызвать. Тоже правило работает и для атрибутов класса.

## Полиморфизм

Полиморфизм позволяет одинаково обращаться с объектами, имеющими однотипный интерфейс, независимо от внутренней реализации объекта. Например, с объектом класса “грузовой автомобиль” можно производить те же операции, что и с объектом класса “автомобиль”, т.к. первый является наследником второго, при этом обратное утверждение неверно (во всяком случае не всегда). Другими словами полиморфизм предполагает разную реализацию методов с одинаковыми именами. Это очень полезно при наследовании, когда в классе наследнике можно переопределить методы класса родителя.

Пример:

```
1 class Cars:
2     def nothing(self): # Пустая функция
3         pass
4
5 class BMW (Cars):
6     def nothing(self, word):
7         print (word + "!") # Функция теперь будет работать по новому
8
9 a = BMW()
10 a.nothing("Some")
```

## Практическое задание

### Вводная часть

1. Создайте программу, которая выводит строку "Hello, robot"
2. Сохраните эту программу в файл `greet.py`
3. Запустите файл `greet.py` интерпретатором Python

## Ввод и вывод данных

1. Создайте программу, которая складывает три числа (пользователь вводит числа)
2. Создайте программу, которая считает площадь прямоугольника, принимая на вход значения двух его сторон

## Математические функции

Напишите программу, которая бы вычисляла заданное арифметическое выражение при заданных переменных. Ввод переменных осуществляется с клавиатуры. Вывести результат Z с 2-мя знаками после запятой.

$$Z = \frac{9\pi t + 10 \cos(x)}{\sqrt{t} - |\sin(t)|} * e^x$$

## Условия и логические операторы

1. Напишите программу, которая принимает ввод двух чисел и печатает значение большего из них.
2. Напишите программу, которая принимает ввод трех чисел и печатает значение большего из них.

## Строки и списки

Вам предстоит реализовать две функции, которые "вращают" тройку влево и вправо. Как это выглядит, вы можете понять из пары примеров:

```
1 >>> triple = ('A', 'B', 'C')
2 >>> rotate_left(triple)
3 ('B', 'C', 'A')
4 >>> rotate_right(triple)
5 ('C', 'A', 'B')
```

## Функции

### Расчет разницы углов.

Напишите функцию `diff`, которая принимает два угла (int) и возвращает **наименьшую** разницу между ними.

```
1 >>> diff(0, 45)
2 45
3 >>> diff(0, 180)
4 180
5 >>> diff(0, 190) # не 190, а 170, потому что 170 меньше
6 170
7 >>> diff(120, 280)
```

```
8 160
9 >>>
```

## Расчет числа Фибоначчи

Реализуйте функцию `fib()`, находящую положительные [Числа Фибоначчи](#). Аргументом функции является порядковый номер числа.

## Формула вычисления числа

```
1 f(0) = 0
2 f(1) = 1
3 f(n) = f(n-1) + f(n-2)
```

## Пример

```
1 >>> fib(3)
2 2
3 >>> fib(5)
4 5
5 >>> fib(10)
6 55
7 >>>
```

## Основы ООП

Реализуйте решение задачи расчета числа Фибоначчи используя классы. Реализация должна проверяться примером. При этом расчёт должен начинаться в методе `get_data`

```
1 f = fib(5)
2 f.get_data()
3 f.print_result() // 5
```