Python

python

Данный курс предназначен для тех, кто только начинает и хочет изучать Python. Мы расскажем как начать обучение с нуля: установка окружения и первая программа, синтаксис языка, типы и объекты, циклы, функции и многое другое.

- **О возможностях языка Python**
- **Установка Python (Anaconda Jupyter)**
- **Синтаксис**
- **Типы и объекты**
- **Типы операторов**
- **Условные операторы**
- **Циклы**
- **Как не надо называть переменные**

- **Числа**
- **Строки**
- **Списки**
- **Кортежи**
- **Словари**
- **Множества**
- **Функции**
- **Работа с файлами**
- **Подключение модулей**

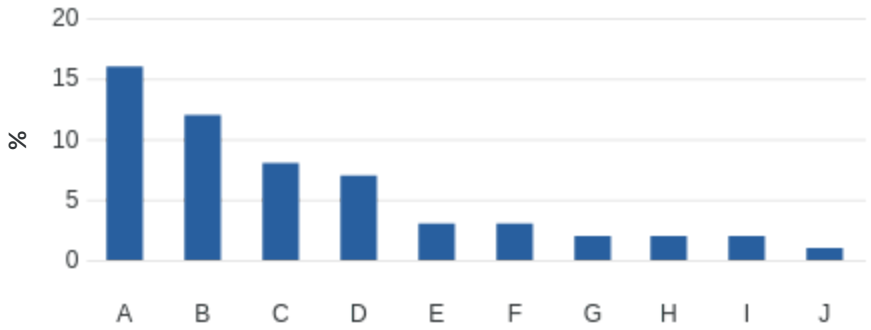
О возможностях языка Python

Python – активно развивающийся язык программирования предназначенный для решения большого числа различных задач. С его помощью можно:

- создавать веб-приложения,
- разрабатывать игры,
- решать бизнес-задачи,
- заниматься математическими вычислениями и анализом данных,
- работать с текстовыми файлами, изображениями, аудио и видео файлами,
- реализовывать графический интерфейс пользователя

и многое другое.

Рейтинг языков программирования на февраль 2019 года



B. C | **12**

C. Python | **8**

D. C++ | **7**

E. Visual Basic .NET | **3**

F. JavaScript | **3**

G. C# | **2**

H. PHP | **2**

I. SQL | **2**

J. Objective-C | **1**

Как видно из рейтинга, Python входит в тройку популярных языков программирования. Такой успех можно объяснить возможностью выполнения широкого спектра задач и удобством языка. Удобство заключается в том, что Python – высокоуровневый язык. Это означает, что сложные описания структур машинного кода выполнены в удобно читаемом для человека виде. Стоит отметить, что при изучении языка необходимо уделять больше времени пониманию того, как работают функции, поскольку это позволит быстрее прокачивать свой навык программирования.

Установка Python (Anaconda Jupyter)

Так как в рамках программы мы планируем работать в Anaconda Jupyter, ниже расскажем как установить данный инструмент. Скачать ее можно с официального [сайта]

Перейдя по ссылке Вы попадете на страницу, где необходимо нажать кнопку Download (рис. 1)

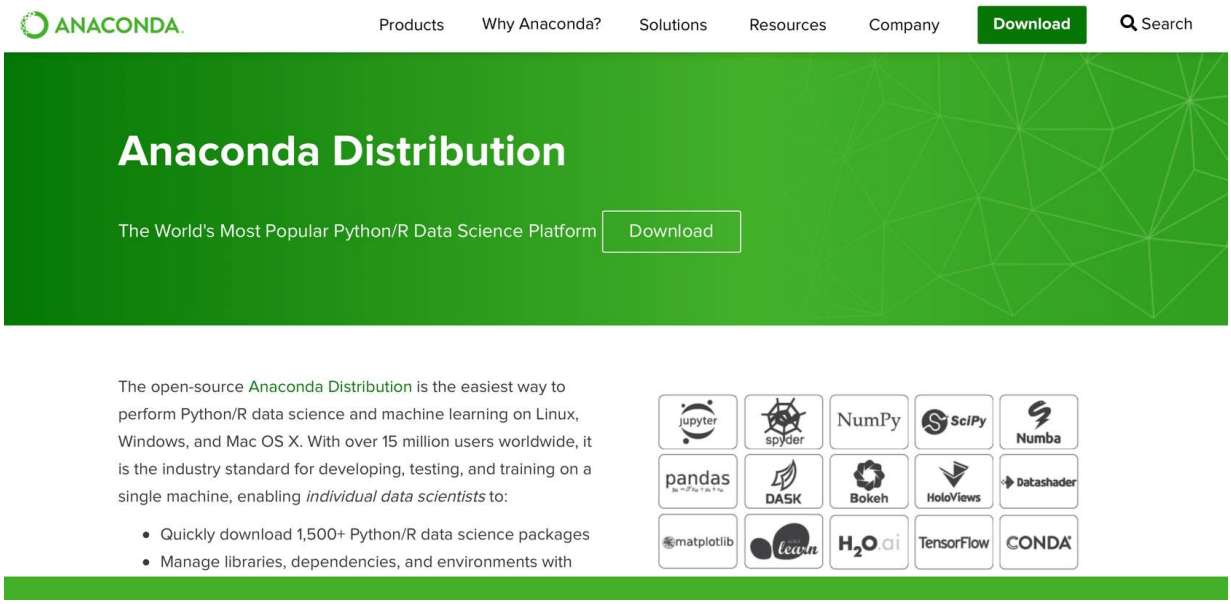


Рис. 1

Перед следующим шагом следует выбрать операционную систему, которая установлена на Вашем компьютере:

- Windows
- macOS
- Linux

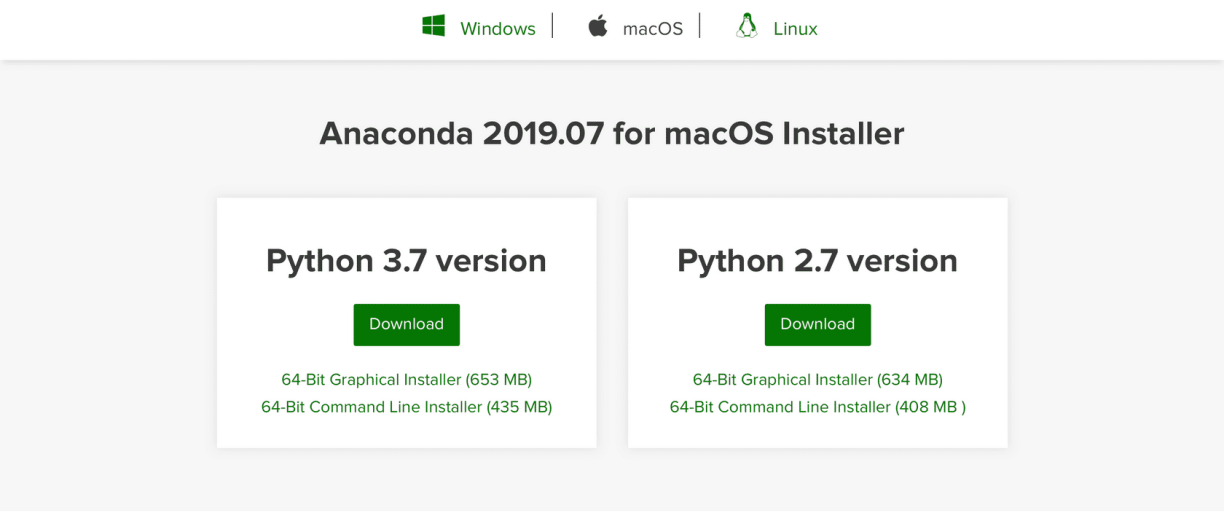


Рис. 2

После чего Вам надо в разделе Python 3.7 нажать кнопку Download (рис. 3) и начнется загрузка установочного файла. После загрузки необходимо запустить установочный файл и следовать инструкции по установке.

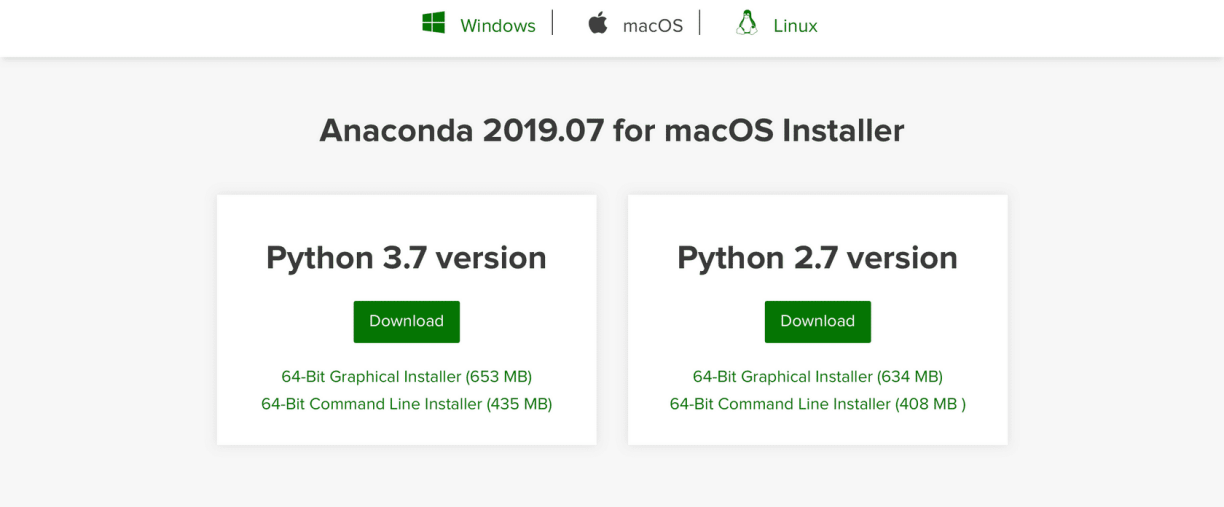


Рис. 3

После запуска Anaconda, первое окно будет выглядеть примерно так (рис. 4).

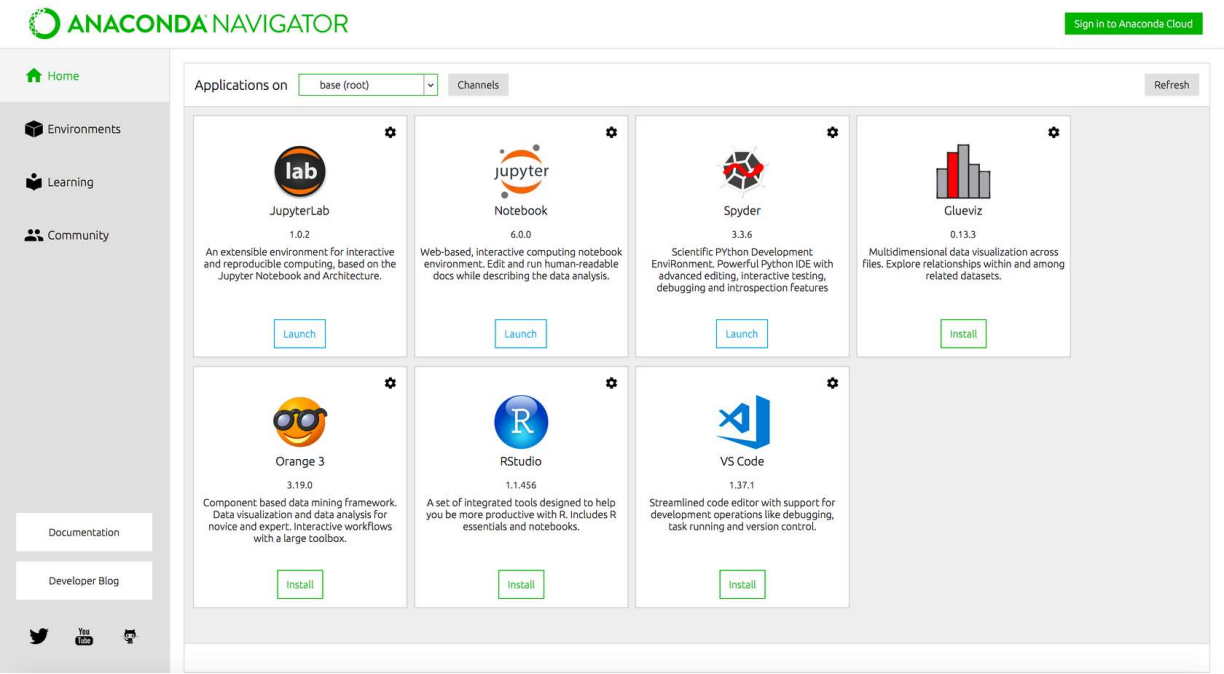


Рис. 4

Далее запустим Jupyter Notebook нажав на кнопку Launch (рис. 5).

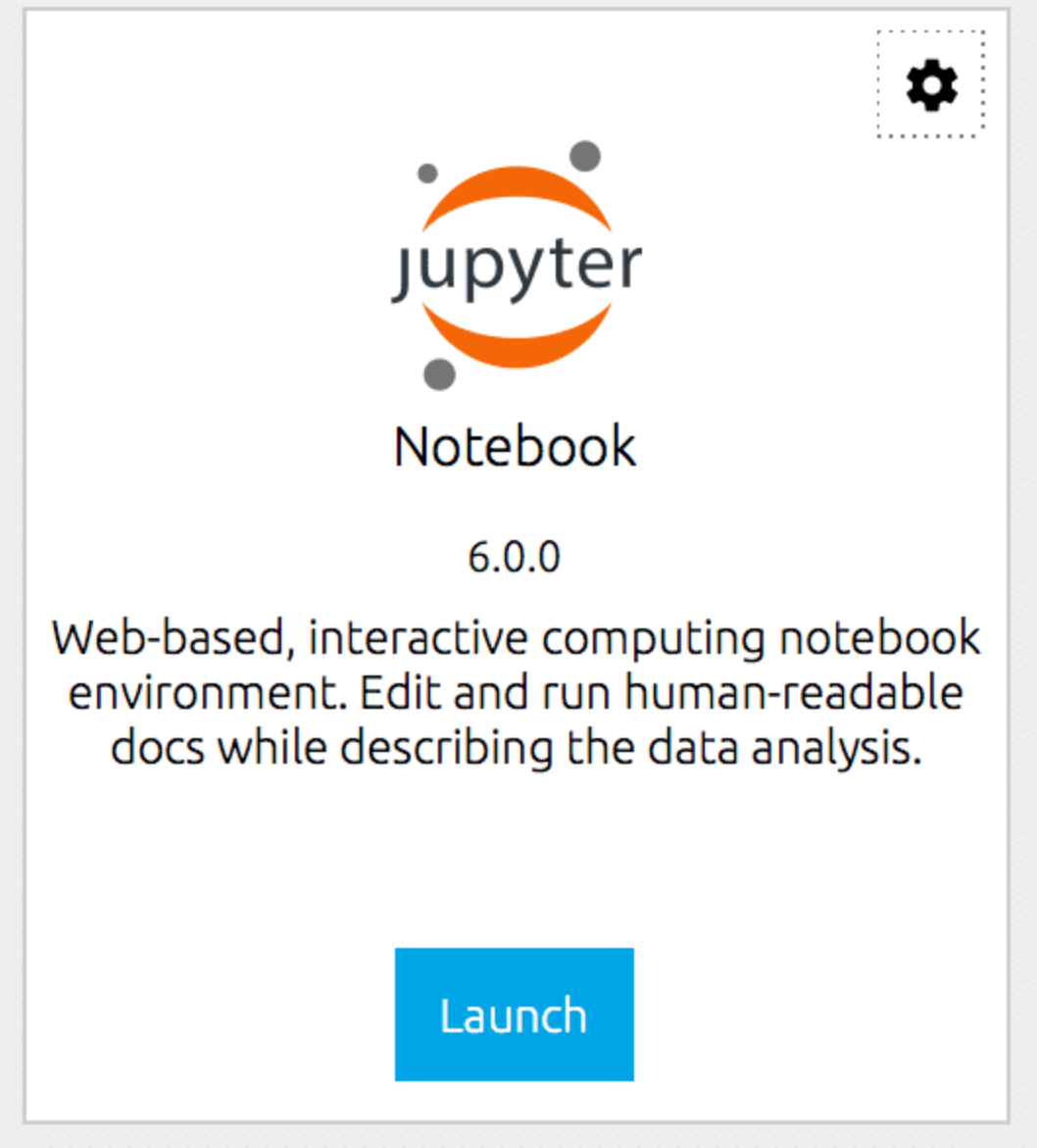


Рис. 5

После старта jupyter, откроется страница на которой будет отображена файловая структура вашего ПК. Давайте выберем папку Documents (или Мои Документы) нажав на нее в списке (рис. 6)

Рис. 6

Далее, перейдя в папку Документы, отобразится ее содержимое (рис. 7)

Рис. 7

Давайте создадим отдельную папку для наших проектов, выбрав контекстное меню New в правом верхнем углу и нажав на опцию Folder (рис. 8)

Рис. 8

После этого в списке файлов, появится папка с наименованием Untitled Folder, давайте выберем ее, кликнув на пустой квадратик слева от названия и выбрав опцию Rename для переименования (рис. 9)

Рис. 9

Откроется новое окно, в котором надо будет присвоить имя, которое Вам больше нравится (рис. 10)

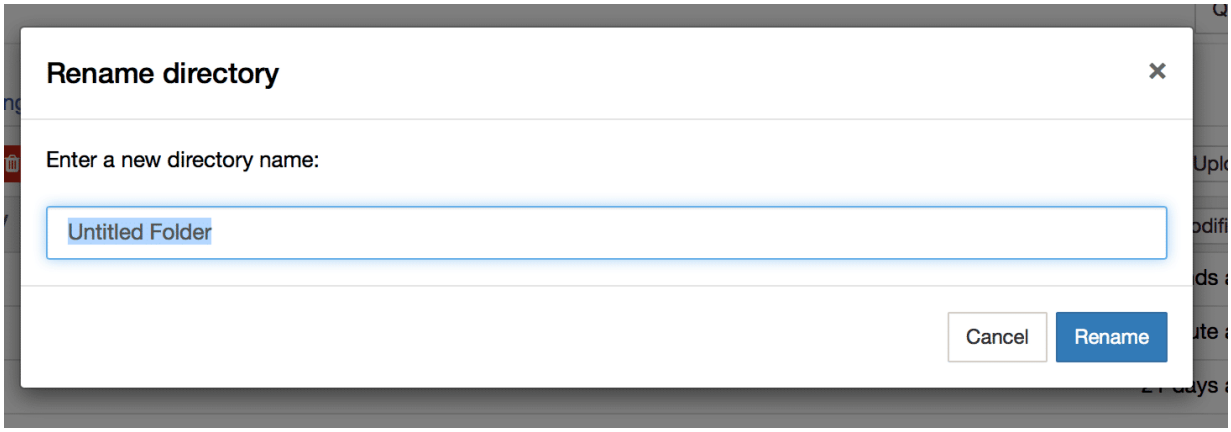


Рис. 10

Далее перейдем в нашу новую папку и как мы видим пока она пуста (рис. 11)

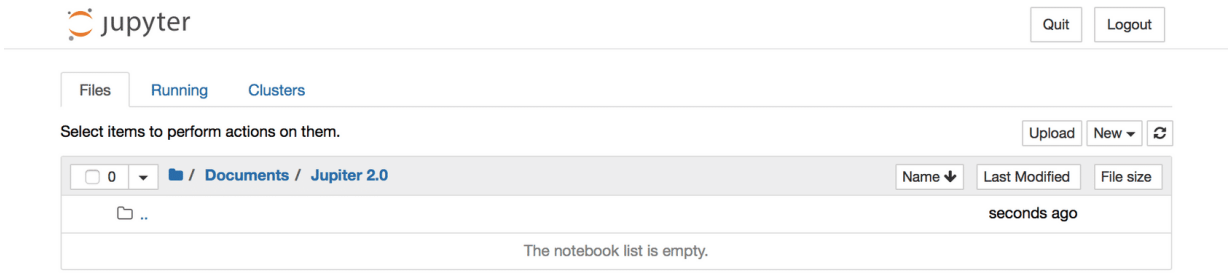


Рис. 11

Для того, чтобы приступить к работе, давайте выберем в контекстном меню New, опцию Python 3 (рис. 12)

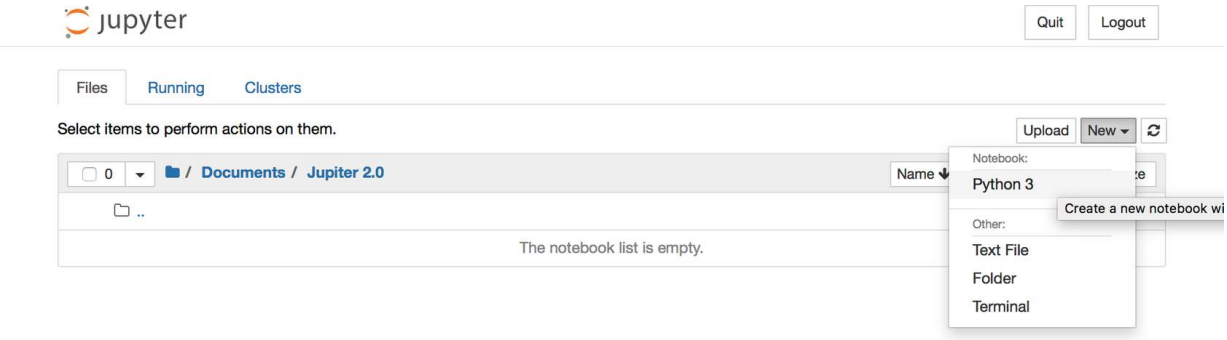


Рис. 12

Перед нами открылась рабочая область, где в дальнейшем мы будем выполнять наши задания, код будем писать внутри прямоугольного поля. Давайте сохраним наш файл и переименуем его(рис. 13)

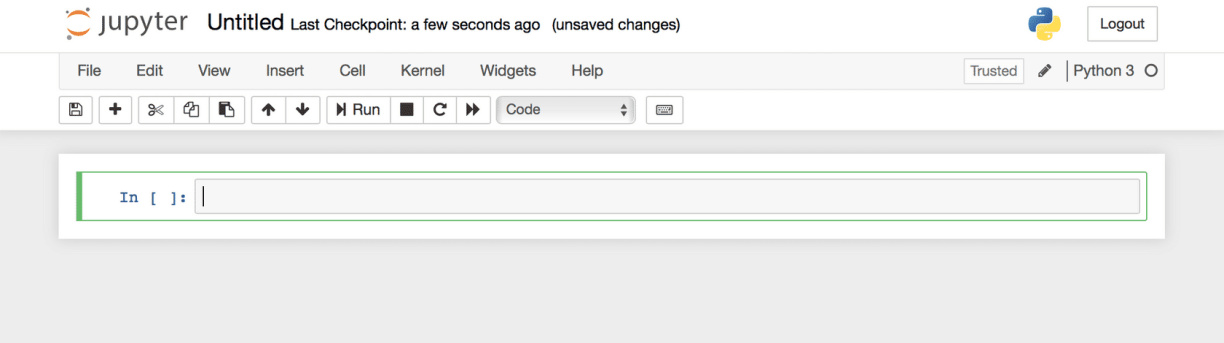


Рис. 13

Для переименования выберите поле Untitled сверху и должно открыться поле для ввода своего названия файла, после чего необходимо подтвердить действие нажав Rename и после этого надо нажать на значок дискеты, чтобы сохранить файл(рис. 14)

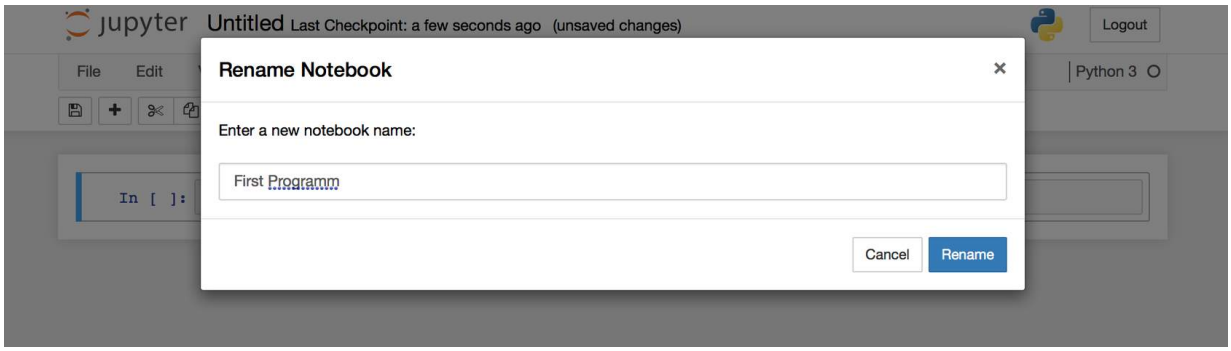


Рис. 14

Как можно убедиться, наш файл появился в структуре папки и теперь мы можем разработать нашу первую программу (рис. 15)

Рис. 15

Попробуйте переписать код с картинки и скомпилировать программу, нажав на Run сверху (рис. 16)

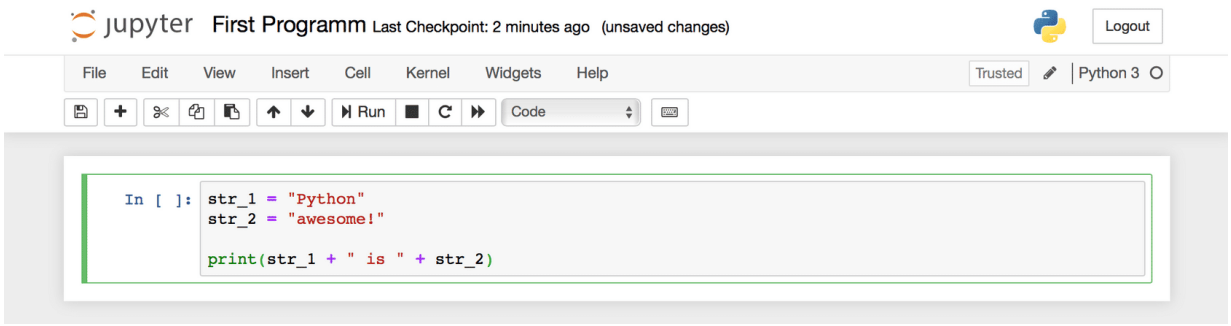


Рис. 16

Под нашим кодом появилось сообщение, которое показывает результат выполнения программы, именно это нам и требовалось, чтобы программа вывела данное сообщение на экран (рис. 17)

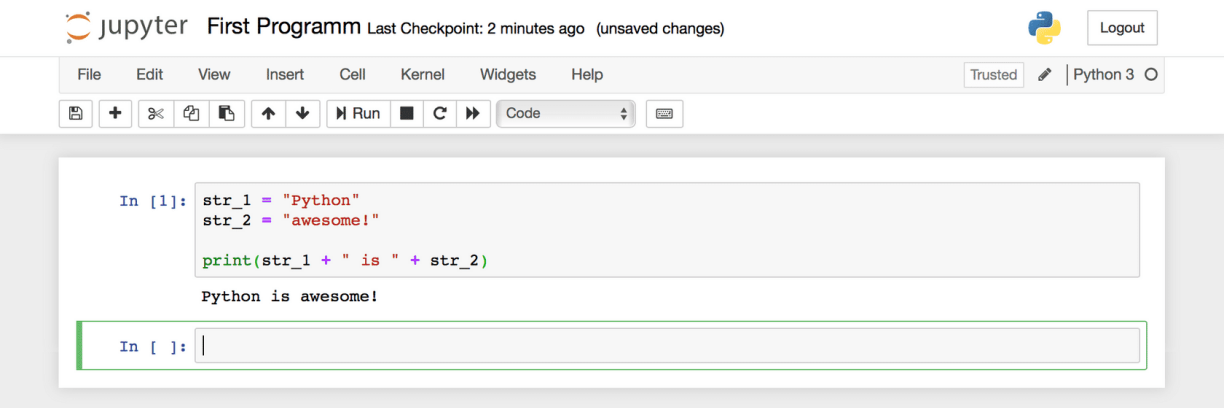


Рис. 17

После того, как программа была выполнена, создался новый блок программ, в котором мы можем написать новый код, или можно вернуться к предыдущему, просто выбрав его и редактировать ил просто удалить, если он не нужен, нажав на ножницы сверху (рис. 18)

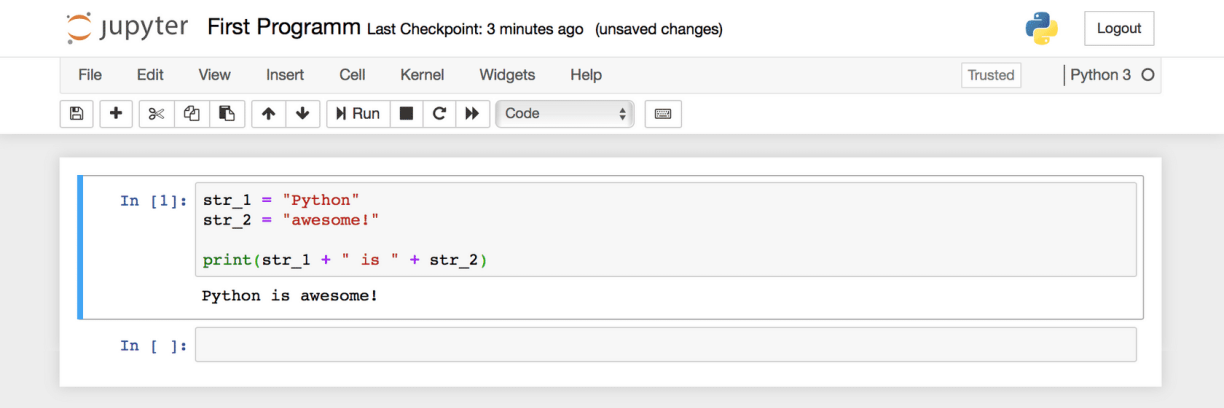


Рис. 18

Немного о работе Python.

После всех проделанных операций с написанием и запуском первой программы стоит поговорить о том, как же на самом деле работает Python. После того, как Вы запускаете команду на выполнение своей программы, Python переводит (компилирует) исходный текст программы в машинный код (байт-код). То есть происходит операция перехода с высокого уровня на более низкий - машинный уровень. Это делается для повышения производительности, так как выполнения программы на машинном коде происходит гораздо быстрее. Далее выполнение файла с машинным кодом выполняет виртуальная машина PVM (интерпретатор), после чего результат процесса выводится на экран (рис. 19).

Рис. 19

Эта часть теории редко где объясняется, но мы считаем нужным дать более глубокое представление о том с чем Вам предстоит работать. Самое время переходить к практике.

Синтаксис

Синтаксис языка Python очень прост, главное помнить основные моменты:

- Конец строки выражения является окончанием инструкции, никаких знаков на конце не требуется, исключением являются вложенные инструкции.

В примере ниже инструкция `var = "first string"` и `var = 2 + 9` называются заданием переменных. Переменная в этих случаях называется `var`, в принципе мы можем использовать практически любое название (исключения прописаны в уроке "Как не надо называть переменные"). Переменные используются для хранения и передачи информации по ходу выполнения программы. Данные, которые передаются в переменную бывают разных типов, об этом мы расскажем подробнее в уроке "Типы и объекты".

```
>>> print("Python is awesome!")

>>> var = "first string"

>>> var = 2 + 9
```

Строковые выражения могут заключаться в одинарные, двойные и тройные кавычки. Также может встречаться комбинация из одинарных и двойных кавычек. Строковое выражение обрамленное в тройные кавычки может состоять из нескольких строчек.

В случае, когда используются комбинации из двойных и одинарных кавычек, одна пара должна быть внешней – она обозначает начало и конец строкового выражения, а внутренняя пара является частью этого выражения. Попробуйте поэкспериментировать с командой print() различные варианты.

```
>>> string = "String"
>>> print(string)
```

```
>>>string = 'String'
>>> print(string)
```

```
>>>string = '''Str
ing'''
>>> print(string)
```

```
>>>string = 'Stri"ng'
>>> print(string)
```

```
>>>string = "Str 'ing' "
>>> print(string)
```

- Комментарии – вспомогательные строки, не обрабатываемые программой, обозначаются знаком # перед началом строки и действует до конца строки. Иногда бывает полезно записать комментарий, чтобы оставить напоминание о том, что выполняется в программе в данный момент.

```
# пишу тут комментарий, чтобы не забыть что делает программа
```

- Вложенная инструкция – часть общей инструкции выполняющаяся при определенных условиях, в этом случае условие заканчивается двоеточием, а само вложенное выражение должно отступать на 4 пробела от места, откуда начинается условие.

```
#инструкция объявляющая переменную var и присваивающая ей
#значение равное 5
var = 5
#основная инструкция 1
if var < 3:
#вложенная инструкция, отступает от основной инструкции 1
    print("less")
#основная инструкция 2
else:
#вложенная инструкция, отступает от основной инструкции 2
    print("more")
```

Отступ в 4 пробела необходим для того, чтобы программа понимала, где начинается вложенная инструкция и к какой основной она относится. Можно и использовать

меньшее число отступов или табуляцию, но в среде разработчиков считается правильным тоном совершать отступ в 4 пробела, так, код становится наиболее читаемым.

- В программе может быть несколько уровней вложенных инструкций, в таком случае надо на каждом уровне делать отступы от начала предыдущей вложенной инструкции.

```
# объявляем переменную var равную 5
var = 5
# основная инструкция
if var > 3:
# вложенная инструкция уровня 1
    if var > 4:
# вложенная инструкция уровня 2, отступает на 4 пробела от
# вложенной инструкции уровня 1
        print("more than 4")
```

Одноуровневые инструкции могут быть написаны в одну строчку:
`if var < 3: print("like this")`
так же, как и обычные инструкции:
`a = 5; b = 6; print(a, b)`
но такие случаи лучше избегать, так как код перестает быть читаемым.

В данном уроке проговорены основные примеры, которые нужны для начала работы, в дальнейшем мы будем обращать внимание на новые элементы, которые будут возникать в процессе обучения. Также мы рекомендуем почитать PEP 8 – руководство по написанию кода на Python.

Типы и объекты

Язык Python характерен своей неявной динамической типизацией. Это означает, что при задании какой-либо переменной, нам не надо объявлять ее тип (число, строка, и т.д.), как это сделано в языке C. То есть достаточно просто присвоить ей значение и в зависимости от того, какое это значение, Python сам определит тип переменной.

Существует несколько видов типов данных - встроенные и неустроенные. Встроенные - те типы, которые встроены в интерпретатор, не встроенные - типы данных, которые можно импортировать из других модулей. В данном курсе нам достаточно рассмотреть только встроенные типы:

- None - неопределенное значение переменной

Зачастую, чтобы отловить какую-то ошибку записи значения куда-либо, мы применяем проверку на отсутствие значения в переменной, ячейке базы данных, таблицы и т.д. Попробуйте повторить код приведенный ниже и посмотрите какой будет результат:

```
null_variable = None
not_null_variable = 'something'

if null_variable == None:
    print('null_variable is None')
else:
    print('null_variable is not None')

if not_null_variable == None:
    print('not_null_variable is None')
else:
    print('not_null_variable is not None')
```

Логический тип данных (bool) удобно использовать, когда в условии может быть только "да" или "нет". В математическом представлении True = 1, False = 0.

- True - логическая переменная, истина
- False - логическая переменная, ложь

Попробуйте выполнить код выше и посмотрите, что выведется на экран.

```
a = 0
b = 0

print(a < b)

print(a > b)

print(a == b)
```

Целые числа используются для стандартных арифметических операций, когда нас не интересует точность, до n-го знака

- int - целое число

Напротив, числа, применяющиеся для точных вычислений до n-го знака после запятой - **числа с плавающей точкой**. При арифметическом взаимодействии двух типов (int и float), результат всегда будет иметь тип float.

- float - число с плавающей точкой

Комплексные числа предназначены для более сложных математических вычислений, они состоят из вещественной и мнимой части.

- complex – комплексное число

Более подробно об этих типах будет рассказано в уроке "Числа".

Списки являются своего рода хранилищем данных разного типа, другими словами списки это массивы, только хранить они могут данные разных типов. Более подробно об этом типе будет рассказано в уроке "Списки".

- list – список

Кортеж – это список, который после создания нельзя изменить, очень полезно его использовать для защиты "от дурака", чтобы по ошибке данные не были изменены.
Более подробно об этом типе будет рассказано в уроке "Кортежи"

- tuple – кортеж

Диапазон используется в циклах, или когда нам нужно вывести последовательность целых чисел. Более подробно об этом типе будет рассказано в уроке "Циклы".

- range – диапазон, неизменяемая последовательность целых чисел.

Строковые переменные мы используем для формирования сообщений, каких либо сочетаний символов. Более подробно об этом типе будет рассказано в уроке "Строки".

- str – строка

Следующие два типа перечислены для ознакомления, в дальнейшем мы не будем уделять им практического значения в рамках данного курса.

Байт это минимальная единица хранения и обработки цифровой информации. Данный тип допускает возможность производить изменение кодировки символов в зависимости от задач.

- bytes - байты

Последовательность байт представляет собой некую информацию (текст, картинка и т.д.). Помимо изменении кодировки, имеет дополнительные возможности применять методы к перекодированным строкам и вносить изменения.

- bytearray - массивы байт

Кроме этого используются следующие объекты

- set - множество

По сути это контейнер для неповторяющихся данных, хранящий эти данные в случайном порядке.

- frozenset - неизменяемое множество

То же, что и множество, без возможности изменить данные внутри контейнера. Более подробно об этих типа будет рассказано в уроке "Множества"

- dict - словарь

Словари являются набором пар "ключ"-"значение", довольно удобный тип данных для формирований структур. Более подробно об этом типе будет рассказано в уроке "Словари".

Задания.

Используя функцию type(), попробуйте выполнить следующие операции и посмотрите что выведется на экран:

- type(None)
- type(True)
- type(False)
- type(1)
- type(5.3)
- type(5 + 4j)
- type([1, 5.3, False, 4])
- type((1, True, 3, 5+4j))
- type(range(5))
- type('Hello')
- type(b'a')
- type(bytearray([1,2,3]))
- type(memoryview(bytearray('XYZ', 'utf-8')))
- type({'a', 3, True})
- type(frozenset({1, 2, 3}))
- type({'a' : 32}).

Типы операторов

Простым языком, оператором является элемент выражения, который указывает на то, какое действие необходимо произвести между элементами. То есть, в выражении "21 - 4" знак "-" является оператором, указывающим на то, что нужно произвести вычитание. "21" и "4" при этом называются операндами.

В языке Python существуют следующие типы операторов:

- Арифметические операторы:

+	Оператор суммы	>>> print(5 + 8) 13
-	Оператор разности	>>> print(31 - 2) 29
*	Оператор произведения	>>> print(12 * 9) 108
/	Оператор деления	>>> print(6 / 4) 1.5
%	Оператор получения остатка от деления	>>> print(6 % 4) 2
**	Оператор возведения в степень	>>> print(9 ** 2) 81

//	Оператор целочисленного деления	>>> print(6 // 4) 1

- Операторы сравнения (реляционные)

==	Проверяет равны ли операнды между собой. Если они равны, то выражение становится истинным.	>>> print(5 == 5)
		True
		>>> print(6 == 44)
		False

!=	Проверяет равны ли операнды между собой. Если они не равны, то выражение становится истинным.	>>> print(12 != 12)
		False
		>>> print(1231 != 0.4)
		True

>	Проверяет больше ли левый операнд чем правый, если больше, то выражение становится истинным.	>>> print(53 > 23)
		True
		>>> print(432 >500)
		False

<	Проверяет меньше ли левый операнд чем правый, если меньше, то выражение становится истинным.	>>> print(5 < 51)
		True
		>>> print(6 < 4)
		False

<>	Проверяет равны ли операнды между собой. Если они не равны, то выражение становится истинным.	>>> print(132 <> 11)
		True
		>>> print(44 <> 44)
		False

>=	Проверяет больше или равен левый операнд, чем правый, Если больше, то выражение становится истинным.	>>> print(5 >= 5)
		True
		>>> print(6 >=44)
		False

<=	Проверяет меньше или равен левый операнд, чем правый, Если меньше, то выражение становится истинным.	>>> print(32 <= 232)
		True
		>>> print(65 <= 9)
		False

- Операторы присваивания

=	Присваивает значение правого операнда левому	>>> var = 5 >>>print(var) 5

+=	Прибавляет значение правого операнда к левому и присваивает левому. a += b эквивалентно записи
	a = a + b
		>>> var = 5
		>>> var += 4
		>>>print(var)
		9

-=	Отнимает значение у левого операнда правое и присваивает левому. a -= b эквивалентно записи
	a = a -b
		>>> var = 5
		>>> var -= 2
		>>>print(var)
		3

*=	Умножает значение левого операнда на правое и присваивает левому. a *= b эквивалентно записи
	a = a * b
		>>> var = 5
		>>> var *= 10
		>>>print(var)
		50

/=	Делит значение левого операнда на правое и присваивает левому. a /= b эквивалентно записи
	a = a / b
		>>> var = 5
		>>> var /= 4
		>>>print(var)
		1.25

%=	Делит значение левого операнда по остатку на правое и присваивает левому. a %= b эквивалентно записи a = a % b	>>> var = 5 >>> var %= 10

		>>>print(var) 5

**=	Возводит значение левого операнда в степень правого и присваивает левому. a **= b эквивалентно записи a = a ** b	>>> var = 5
		>>> var **= 8
		>>>print(var)
		390625

//=	Целочисленно делит значение левого операнда на правое и присваивает левому. a //= b эквивалентно записи a = a // b	>>> var = 5
		>>> var //= 30
		>>>print(var)
		0

- Побитовые операторы

Данные операторы работают с данными в двоичной системе счисления. Например число 13 в двоичной системе будет равно 1101

&	Бинарный "И" оператор, копирует бит в результат только если бит присутствует в обоих операндах	0 & 0 = 0 1 & 0 = 0 0 & 1 = 0 1 & 1 = 1 101 & 011 = 001
|	Бинарный "ИЛИ" оператор копирует бит, если тот присутствует в хотя бы в одном операнде	0 | 0 = 0 1 | 0 = 1 0 | 1 = 1 1 | 1 = 1 101 | 011 = 111
^	Бинарный "Исключительное ИЛИ" оператор копирует бит только если бит присутствует в одном из операндов, но не в обоих сразу	0 ^ 0 = 0 1 ^ 0 = 1 0 ^ 1 = 1 1 ^ 1 = 0 101 ^ 011 = 110

~	Бинарный комплиментарный оператор. Является унарным (то есть ему нужен только один операнд) меняет биты на обратные, там где была единица становиться ноль и наоборот	~1 = 0 ~0 = 1 ~101 = 010
>>	Побитовый сдвиг вправо. Значение левого операнда "сдвигается" вправо на количество бит указанных в правом операнде	100 >> 2 = 001
<<	Побитовый сдвиг влево. Значение левого операнда "сдвигается" влево на количество бит указанных в правом операнде	100 >> 2 = 10000

- Логические операторы

and	Логический оператор "И". Условие будет истинным если оба операнда истина	True and True = True. True and False = False. False and True = False. False and False = False.
or	Логический оператор "ИЛИ". Если хотя бы один из операндов истинный, то и все выражение будет истинным	True or True = True. True or False = True. False or True = True. False or False = False.
not	Логический оператор "НЕ". Изменяет логическое значение операнда на противоположное	not True = False. not False = True.

- Операторы членства

Данные операторы участвуют в поиске данных в некоторой последовательности данных.

in	Возвращает истину, если элемент присутствует в последовательности, иначе возвращает ложь	>>> print('he' in 'hello')
		True
		>>> print(5 in [1, 2, 3, 4, 5])
		True
		>>> print(12 in [1, 2, 4, 56])
		False

not in	Возвращает истину если элемента нет в последовательности	результаты обратны предыдущему примеру

- Операторы тождественности

Данные операторы сравнивают размещение двух объектов в памяти компьютера

is	Возвращает истину, если оба операнда указывают на один объект	>>> a = 12
		>>>b = 12
		>>> a is b
		True
		>>> c = 22
		>>> a is c
		False

is not	Возвращает ложь, если оба операнда указывают на один объект	результаты обратны предыдущему примеру

- Условные операторы

Особый вид операторов, про который мы расскажем в уроке "Условные операторы".

Условные операторы

Условные операторы нужны для проверки условий и в зависимости от результата вести логику выполнения программы в своём направлении.

Оператор if

Условный оператор `if` ("если") является основным оператором проверки выполнения условия. Для того, чтобы выполнить простую вложенную инструкцию, необходимо проверить условие на соответствие использовав оператор `if` и прописав после него соответствующее условие:

```
# объявляем переменную
var = 5
# выполняем проверку условия
if var < 10:
# если условие выполняется, то выполняется вложенная инструкция
    print("var less than 10")
```

В данном случае мы объявляем переменную `var` и присваиваем ей значение равное `5`, далее выполняем проверку условия – если переменная меньше `10`, то вывести соответствующее сообщение.

Оператор else

Условный оператор else ("иначе") является продолжением основной инструкции. Else используется тогда, когда необходимо перебрать все оставшиеся варианты, не вошедшие в if:

```
# объявляем переменные
var_1 = 10
var_2 = 9
# выполняем проверку условия
if var_1 == var_2:
# если условие выполняется, то выполняется вложенная инструкция
    print("var_1 equal var_2")
# иначе при любых других значениях
else:
# выполняется другая вложенная инструкция
    print("var_1 not equal var_2")
```

Как показано в примере выше, мы задаем переменной var_1 значение равное 10, а переменной var_2 значение 9, затем производим сравнение наших переменных на равенство, если var_1 равно var_2, тогда следует вывести сообщение, что они равны, иначе, вывести сообщение, что переменные не равны. Таким образом, оператор else позволяет выполнить инструкцию print("var_1 not equal var_2") при любых значениях var_1 и var_2 кроме тех, при которых они равны.

Оператор elif

Оператор elif является сокращенным вариантом от конструкции else if, которая позволяет добавить дополнительные условия в логику выполнения программы:

```
# объявляем переменные
var_1 = 10
var_2 = -10
```

```
# выполняем проверку условия
if var_1 == var_2:
# если условие выполняется, то выполняется вложенная инструкция
    print("var_1 equal var_2")
# иначе если выполняется другое условие
elif var_1 < var_2:
# выполняется другая вложенная инструкция
    print("var_1 less than var_2")
# при любом другом случае
else:
    print("var_1 more than var_2")
```

Здесь мы объявляем переменные var_1 и var_2 и присваиваем значения 10 и -10 соответственно, после этого выполняем проверку первого условия, равны ли эти переменные, если они равны, то выводим соответствующее сообщение. Если они не равны, то пытаемся понять, какая из этих переменных больше. Выполняется проверка второго условия (elif), если var_1 меньше var_2, тогда выводим соответствующее сообщение, и наконец, остается последний вариант, который и выводит сообщение о том, что var_1 больше чем var_2.

Главное, на что стоит обратить внимание - после использования конструкций if и elif всегда необходимо записывать условие проверки (в нашем случае var_1 == var_2 и var_1 < var_2 соответственно), а else всегда используется без условий, потому что означает выполнение инструкции при любых других вариантах, которые не были рассмотрены операторами if и elif.

Количество конструкций elif может быть множество, тогда, как if и else используются по 1 разу на одном уровне:

```
if (условие):
    (выполнение условия)
elif (другое условие):
    (выполнение другого условия)
elif (третье условие):
    (выполнение третьего условия)
elif (четвертое условие):
...
```

```
...
...
else:
    (выполнение при всех других не рассмотренных ранее случаях)
```

Можно использовать эту конструкцию на разных уровнях:

```
if (условие):
    if (дополнительное условие):
        (выполнение дополнительного условия)
    elif (другое дополнительное условие):
         (выполнение другого дополнительного условия)
    elif ...
         ...
    ...
    else:
        ...
elif (другое условие):
    (выполнение другого условия)
elif ...
    ...
else:
    ...
```

В случае, если код выглядит следующим образом:

```
var = 10
if var == 10:
    print("var equal 10")
if var < 10:
    print("var less than 10")
```

```
else:
    print("var more than 10")
```

То после его выполнения вы получите следующие сообщения:

```
var equal 10
var more than 10
```

Смысл в том, что при такой записи у нас имеется 2 независимых блока проверки условий. 1 блок состоит из первого if и вывода сообщения, второй состоит из конструкции if и else. То есть наша переменная var обязательно будет проходить проверку на условия в каждом из этих блоков.

Перед тем как приступить к выполнению заданий, мы расскажем об одной функции, которая пригодится для их выполнения.

- input() – принимает значения введенные с клавиатуры.

Данная функция останавливает ход выполнения программы до тех пор, пока пользователь не введет данные с клавиатуры, после чего подтверждает их. Функция принимает эти данные и выполнение программы продолжается. Давайте попробуем:

```
var = input()
print(var)
```

После того, как мы скомпилировали код, нам предлагается ввести данные, после этого наши данные присваиваются переменной var, тип этих данных - строки. После этого программа продолжает выполняться и выводит на экран значение переменной var. Потренируйтесь перед началом выполнения заданий.

Задания

Задание 1. Даны два целых числа. Выведите значение наименьшего из них.

1) Входные данные: -15, -8. Выходные данные: -15
2) Входные данные: 15, -8. Выходные данные: -8
3) Входные данные: 3, 7. Выходные данные: 3
4) Входные данные: 2, 2. Выходные данные: 2

Давайте выполним первое задание вместе:

```
# Создаем переменные и присваиваем им значения, введенные с клавиатуры
x = int(input())
y = int(input())
#  Производим сравнение чисел
if x < y:
    print(x)
else:
    print(y)
```

Если скомпилировать код написанный выше, то он поочередно предложит ввести 2 числа, после чего произведет их сравнение. Можно подставить данные из примеров 1,2,3 и 4 и убедиться, что результат совпадает с выходными данными заявленными в примере. Метод int(), который примерялся в этой задаче, позволяет перевести из типа "строка" в тип целых чисел. Это нужно было сделать, потому что, как говорилось выше - метод input() возвращает строковые данные.

В целом Вы могли и не использовать функцию input(), а просто записать следующим образом, каждый раз меняя данные:

```
# Создаем переменные и присваиваем им значения
x = -8
y = -12
#  Производим сравнение чисел
if x < y:
    print(x)
else:
    print(y)
```

Дальнейшие задачи предлагаются для самостоятельного решения.

Задание 2. Даны три целых числа. Выведите значение наименьшего из них.

1) Входные данные: -10, 5, 10. Выходные данные: -10

2) Входные данные: 10, 30, 4. Выходные данные: 4

3) Входные данные: -5, -3, -3. Выходные данные: -5

4) Входные данные: 1, 10, 20. Выходные данные: 1

Задание 3. Даны три целых числа. Определите, сколько среди них совпадающих. Программа должна вывести одно из чисел: 3 (если все совпадают), 2 (если два совпадает) или 0 (если все числа различны).

1) Входные данные: 10, 5, 10. Выходные данные: 2

2) Входные данные: 17, 17, -9. Выходные данные: 2

3) Входные данные: 4, -82, -82. Выходные данные: 2

4) Входные данные: 100, 100, 100. Выходные данные: 3

Циклы

Циклы это специальные выражения, которые позволяют выполнять часть кода несколько раз, если оно того требует. Циклы очень полезны в программировании, поскольку это позволяет не прописывать вручную одну и ту же операцию несколько раз. Допустим нам надо вывести на экран 13 чисел от 1 до 13, как бы это выглядело без циклов:

```
print(1)
print(2)
print(3)
print(4)
print(5)
print(6)
print(7)
print(8)
print(9)
print(10)
print(11)
print(12)
print(13)
```

Согласитесь, утомительно. Давайте посмотрим далее что можно с этим сделать, применив циклы. В языке Python есть несколько видов циклических выражений.

Цикл While

Самый простой вид циклов, выполняет инструкцию в блоке до тех пор, пока условие не станет истинным.

```
# создаем переменную, равную 1
var = 1
# прописываем цикл с условием - выполнять до тех пор, пока переменная
# меньше или равна 13
while var <= 13:
# выводим значение переменной
    print(var)
# увеличиваем переменную на 1
    var += 1
```

Попробуйте выполнить пример выше и посмотрите что получится, можете не переписывать комментарии. Как видно из примера, у нас появляются вложенные инструкции, а значит их надо отделять пробелами.

> **Необходимо всегда помнить, что при использовании цикла while, нужно следить за тем, чтобы условие выполнялось. Если бы в примере выше мы не увеличивали переменную var на 1, тогда цикл бы стал бесконечным, и программа бы выполнялось до тех пор, пока не закончится память компьютера.**

Цикл For

Цикл for менее универсальный, но работает быстрее, чем цикл while. Он способен проходить по любому итерируемому объекту, будь то списки, словари, кортежи, строки.

```
for i in [1, 2, 3, 4, 5 ,6 ,7 ,8 ,9 , 10, 11, 12, 13]:
    print(i)
```

В примере выше написана программа с использованием цикла for. Мы указываем, что наша вложенная инструкция print(i) должна выполняться до тех пор, пока переменная i принимает каждое значение из массива [1, 2, 3, 4, 5 ,6 ,7 ,8 ,9 , 10, 11, 12, 13]. То есть данный цикл описывает конструкцию:

```
 i = 1
print(i)
 i = 2
print(i)
 i = 3
print(i)
 ...
 i = 13
print(i)
```

Но в нашем примере все равно отсутствует элегантность, присущая языку Python. Давайте попробуем что-нибудь сделать с массивом из 13 чисел. Что если нам понадобится выводить 100 или 100000 чисел? Для этих целей применим функцию range(), которая поможет нам автоматически сгенерировать последовательность чисел.

Давайте выполним пару команд и посмотрим на результат:

Здесь и далее в тексте могут встречаться записи следующего вида:

```
>>> print(5)

5
```

Это значит, что примеры были выполнены в среде разработки IDLE для наглядности и более кратком изложении . Выражение стоящее после >>> означает инструкцию, выражение без стрелочек - результат компиляции программы. Соответственно, когда вы повторяете примеры, писать в своей программе >>> не надо. Когда кусок кода приведен без стрелочек, значит, что он показан для самостоятельной компиляции.

```
>>> var = range(1, 14)
>>> print(var)
range(1, 14)

>>> print(var[0])
1
>>> print(var[1])
2
>>> print(var[12])
13
>>> print(var[14])
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    print(var[14])
IndexError: range object index out of range
```

Мы присвоили переменной значение выражения range(1,14), после того, как мы решили вывести содержимое переменной var, обнаружилось, что в ней заложено это выражение - range(1, 14). Как говорилось ранее, range это особый тип, который содержит в себе последовательность данных. Чтобы это показать, выполним команду print(var[0]), которая выводит содержимое переменной, стоящей в 0 ячейке памяти. Ячейки памяти берут отсчет с 0, поэтому на 12 позиции будет храниться число 13, как видно из примера. Тем самым мы видим, что переменная var содержит в себе последовательность чисел от 1 до 13. Но как быть с 14 позицией, ведь в команде range() второй параметр у нас равен 14, особенность range() состоит в том, что последнее число последовательности не включается в нее, то есть мы говорим, что нам надо сформировать последовательность чисел от 1 до 14, не включая 14. range() работает так, потому что чаще всего используется для перебора индексов в списках, отсчет которых начинается с 0, а задача перебора обычных чисел редко применяется на практике.

Вернемся к циклу for. Теперь мы можем написать нашу программу максимально компактно:

```
for i in range(1,14):
    print(i)
```

Оператор continue

Оператор continue позволяет начать следующий проход цикла, минуя оставшиеся инструкции.

```
for var in 'Python':
    if var == 'h':
        continue
    print(var)
```

На примере выше мы перебираем последовательность символов и когда наша переменная хранит в себе символ 'h', мы используем оператор continue, чтобы пропустить дальнейшую инструкцию print(var).

Оператор break

Оператор break досрочно прерывает цикл. Повторите пример ниже и посмотрите, что получится.

```
for var in 'Python':
    if var == 'h':
```

```
            break
        print(var)
```

Оператор else

Оператор else проверяет цикл на экстренный выход (break). Если экстренного выхода не было, т.е. оператор break не был выполнен, блок инструкций вложенный в оператор else - выполняется.

```
for var in 'Python':
    if var == 'a':
        break
else:
    print('Символа a нет в слове Python')
```

Операторы continue, break и else работают с циклами for и while.

Задания.

Задание 1. Дано 10 целых чисел. Вычислите их сумму.

1) Входные данные: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Выходные данные: 45

2) Входные данные: 1, 1, 1, 1, 1, 1, 1, 1, 1, 1. Выходные данные: 10

3) Входные данные: 8, 4, 5, 3, 9, 2, 3, 4, 5, 1. Выходные данные: 44

4) Входные данные: 758, 483, 893, 393, 293, 292, 292, 485, 828, 182. Выходные данные: 4899

Задание 2. Подсчитайте количество нулей среди введенных чисел и выведите это количество. Вам нужно подсчитать количество чисел, равных нулю, а не количество цифр.

1) Входные данные: 0, 1, 2, 3, 4. Выходные данные: 1

2) Входные данные: 1, 0, 1, 0, 1, 0, 1, 0, 1, 0. Выходные данные: 5

3) Входные данные: 8, 4, 0, 3, 9, 2, 3, 0. Выходные данные: 2

4) Входные данные: 700, 8, 0, 20, 13. Выходные данные: 1

Задание 3. По данному натуральному $n \leq 9$ выведите лесенку из n ступенек, i-я ступенька состоит из чисел от 1 до i без пробелов.

1) Входные данные: 1.

Выходные данные:

1

2) Входные данные: 3.

Выходные данные:

1

12

123

3) Входные данные: 5.

Выходные данные:

1

12

123

1234

12345

Как не надо называть переменные

Цель данного урока – перечислить ключевые слова и встроенные функции в стандартном пакете Python 3, чтобы в будущем вы избежали совпадений в названии своих переменных или функций и вместе с этим последующих ошибок.

Ключевые слова:

- False - показатель ложности для булева типа;
- True - показатель истинности для булева типа;
- None - "пустой" объект;
- and - логический оператор И;
- with/as - менеджер контекста;
- assert - условие, вызывающее исключение, если условие ложно;
- break - оператор выхода из цикла;
- class - тип, состоящий из методов и атрибутов;
- continue - оператор перехода на следующую итерацию цикла;
- def - обозначение функции;
- del - определение функции;
- elif - условный оператор в противном случае-если;
- else - условный оператор в противном случае;
- except - оператор перехвата исключения;
- finally - выполняет инструкцию вне зависимости от исключения;
- for - оператор цикла for;

- from - оператор импорта из модуля;
- global - оператор создания доступности обращения к переменной за пределами функции;
- if - условный оператор если;
- import - оператор импорта модуля;
- in - оператор проверки на вхождение;
- is - оператор проверки ссылаются ли 2 объекта на одно место в памяти;
- lambda - определение анонимной функции;
- nonlocal - оператор создание доступности обращения к переменной в объемлющей инструкции;
- not - логический оператор не;
- or - логический оператор или;
- pass - ничего не выполняющий оператор;
- raise - оператор вызова исключения;
- return - оператор возвращения результата;
- try - выполнить инструкции, перехватив исключения;
- while - условно-циклический оператор до тех пор;
- yield - определение функции-генератора.

Встроенные функции:

- bool(x) - преобразование к типу bool, использующая стандартную процедуру проверки истинности. Если x является ложным или опущен, возвращает значение False, в противном случае она возвращает True.
- bytearray([источник [, кодировка [ошибки]]]) - преобразование к bytearray. Bytearray - изменяемая последовательность целых чисел в диапазоне 0≤X<256. Вызванная без аргументов, возвращает пустой массив байт.
- bytes([источник [, кодировка [ошибки]]]) - возвращает объект типа bytes, который является неизменяемой последовательностью целых чисел в диапазоне 0≤X<256. Аргументы конструктора интерпретируются как для bytearray().
- complex([real[, imag]]) - преобразование к комплексному числу.
- dict([object]) - преобразование к словарю.

- float([X]) – преобразование к числу с плавающей точкой. Если аргумент не указан, возвращается 0.0.
- frozenset([последовательность]) – возвращает неизменяемое множество.
- int([object], [основание системы счисления]) – преобразование к целому числу.
- list([object]) – создает список.
- memoryview([object]) – создает объект memoryview.
- object() – возвращает безликий объект, являющийся базовым для всех объектов.
- range([start=0], stop, [step=1]) – арифметическая прогрессия от start до stop с шагом step.
- set([object]) – создает множество.
- slice([start=0], stop, [step=1]) – объект среза от start до stop с шагом step.
- str([object], [кодировка], [ошибки]) – строковое представление объекта. Использует метод ___str___.
- tuple(obj) – преобразование к кортежу.
- abs(x) – возвращает абсолютную величину (модуль числа).
- all(последовательность) – возвращает True, если все элементы истинные (или, если последовательность пуста).
- any(последовательность) – возвращает True, если хотя бы один элемент – истина. Для пустой последовательности возвращает False.
- ascii(object) – как repr(), возвращает строку, содержащую представление объекта, но заменяет не-ASCII символы на экранированные последовательности.
- bin(x) – преобразование целого числа в двоичную строку.
- callable(x) – возвращает True для объекта, поддерживающего вызов (как функции).
- chr(x) – возвращает односимвольную строку, код символа которой равен x.
- classmethod(x) – представляет указанную функцию методом класса.
- compile(source, filename, mode, flags=0, dont_inherit=False) – компиляция в программный код, который впоследствии может выполниться функцией eval или exec. Строка не должна содержать символов возврата каретки или нулевые байты.
- delattr(object, name) – удаляет атрибут с именем 'name'.
- dir([object]) – список имен объекта, а если объект не указан, список имен в текущей локальной области видимости.
- divmod(a, b) – возвращает частное и остаток от деления a на b.

- enumerate(iterable, start=0) – возвращает итератор, при каждом проходе предоставляющем кортеж из номера и соответствующего члена последовательности.
- eval(expression, globals=None, locals=None) – выполняет строку программного кода.
- exec(object[, globals[, locals]]) – выполняет программный код на Python.
- filter(function, iterable) – возвращает итератор из тех элементов, для которых function возвращает истину.
- format(value[,format_spec]) – форматирование (обычно форматирование строки).
- getattr(object, name ,[default]) – извлекает атрибут объекта или default.
- globals() – словарь глобальных имен.
- hasattr(object, name) – имеет ли объект атрибут с именем 'name'.
- hash(x) – возвращает хеш указанного объекта.
- help([object]) – вызов встроенной справочной системы.
- hex(x) – преобразование целого числа в шестнадцатеричную строку.
- id(object) – возвращает "адрес" объекта. Это целое число, которое гарантированно будет уникальным и постоянным для данного объекта в течение срока его существования.
- input([prompt]) – возвращает введенную пользователем строку. Prompt – подсказка пользователю.
- isinstance(object, ClassInfo) – истина, если объект является экземпляром ClassInfo или его подклассом. Если объект не является объектом данного типа, функция всегда возвращает ложь.
- issubclass(класс, ClassInfo) – истина, если класс является подклассом ClassInfo. Класс считается подклассом себя.
- iter(x) – возвращает объект итератора.
- len(x) – возвращает число элементов в указанном объекте.
- locals() – словарь локальных имен.
- map(function, iterator) – итератор, получившийся после применения к каждому элементу последовательности функции function.
- max(iter, [args ...] * [, key]) – максимальный элемент последовательности.
- min(iter, [args ...] * [, key]) – минимальный элемент последовательности.
- next(x) – возвращает следующий элемент итератора.
- oct(x) – преобразование целого числа в восьмеричную строку.

- open(file, mode='r', buffering=None, encoding=None, errors=None, newline=None, closefd=True) – открывает файл и возвращает соответствующий поток.
- ord(c) – код символа.
- pow(x, y[, r]) – идентично выражению (x ** y) % r.
- reversed(object) – итератор из развернутого объекта.
- repr(obj) – представление объекта.
- print([object, ...], *, sep=" ", end='\n', file=sys.stdout) – вывод результата на экран.
- property(fget=None, fset=None, fdel=None, doc=None) – возвращает специальный объект дескриптора.
- round(X [, N]) – округление до N знаков после запятой.
- setattr(объект, имя, значение) – устанавливает атрибут объекта.
- sorted(iterable[, key][, reverse]) – отсортированный список.
- staticmethod(function) – статический метод для функции.
- sum(iter, start=0) – сумма членов последовательности.
- super([тип [, объект или тип]]) – доступ к родительскому классу.
- type(object) – возвращает тип объекта.
- type(name, bases, dict) – возвращает новый экземпляр класса name.
- vars([object]) – словарь из атрибутов объекта. По умолчанию – словарь локальных имен.
- zip(*iters) – итератор, возвращающий кортежи, состоящие из соответствующих элементов аргументов-последовательностей.

Числа

Целые числа

Целые числа (int) поддерживают обычный набор математических операций.

a+ b	Сложение
a - b	Вычитание
a * b	Умножение
a / b	Деление
a // b	Получение целой части от деления
a % b	Остаток от деления
-a	Смена знака числа
abs(a)	Модуль числа
divmod(a,b)	Получение пары чисел (a // b, a % b)
a ** b	Возведение в степень

pow(a, b[, c])	a^b по модулю, если модуль задан

```
>>> 255 + 34
289
>>> 5 * 2
10
>>> 20 / 3
6.666666666666667
>>> 20 // 3
6
>>> 20 % 3
2
>>> 3 ** 4
81
>>> pow(3, 4)
81
>>> pow(3, 4, 27)
0
>>> 3 ** 150
369988485035126972924700782451696644186473100389722973815184405301748249
```

Над целыми числами также можно производить битовые операции (&, | , ^, <<, >>, ~).

Также целые числа можно переводить в другие системы счисления используя методы:

- bin(a) - перевод числа в двоичную систему счисления
- hex(a) - перевод числа в 16-тиричную систему счисления
- oct(a) - перевод числа в 8-миричную системы счисления

```
>>> bin(3)
'0b11'
>>> hex(123)
'0x7b'
>>> oct(15)
'0o17'
```

После того как мы перевели число в другую систему счисления, перед самим числом обычно записывается символьное обозначение системы, в которой записано число. Так двоичная система обозначается символами "0b", 16-тиричная - "0x", 8-ричная - "0o"

Числа с плавающей точкой (float)

Вещественные числа поддерживают те же операции, что и целые, однако из-за компьютерного представления, могут возникать неточности.

Для округления вещественного числа можно применить метод round().

Если к вещественному числу применить метод int(), т.е. привести его к целочисленному типу, тогда дробная часть просто отсекается.

```
>>> round(16.76)
17
>>> int(123.823)
123
```

Строки

Строки это упорядоченные последовательности символов, для работы с текстовой информацией. Чтобы присвоить переменной строковое значение, достаточно приравнять выражение в кавычках.

```
s = 'ain"t it fun?'
s = "ain't it fun?"
```

Как говорилось ранее, разницы между одинарными и двойными кавычками нет, они были введены для того, чтобы можно было не экранировать кавычки внутри предложения, как показано на примере выше. То есть, если бы мы в первом варианте с одинарными кавычками в слове ant"t применили также одинарную кавычку, тогда бы мы получили сообщение об ошибке, потому что наша строка состояла бы только из символов 'ain', остальной текст не был бы воспринят как часть строки, и в конце стояла бы одинокая открывающая кавычка. Но и этого можно избежать применив экранирование, или другими словами – отмену действия прямого назначения символа:

```
s = 'ain\'t it fun?'
```

В данном случае символ \ отменяет действие кавычки, то есть она не закрывает строку, а воспринимается как часть выражения и заканчивается оно в самом конце, однако не всегда такой вид удобночитаем.

Существуют различные экранированные последовательности, которые позволяют вставить символы, которые трудно ввести с клавиатуры, в рамках данного курса можно рассмотреть следующую последовательность:

- `\n` – перевод строки

```
>>> s = 'ain"t \nit \nfun?'
>>> print(s)
ain"t
it
fun?
```

Что можно делать со строками.

Строки можно складывать:

```
>>> s_1 = 'Python'
>>> s_2 = ' '
>>> s_3 = 'is awesome!'
>>> print(s_1 + s_2 + s_3)
'Python is awesome!'
```

Или дублировать:

```
>>> print('spam' * 3)
spamspamspam
```

С помощью метода len() можно узнать количество символов в строке:

```
>>> len('Python')
6
```

Можно обращаться к элементам по их индексу (**индексация ведется от нуля**):

```
>>> s = 'Python'
>>> print(s[0])
P
>>> print(s[5])
n
```

По индексу можно извлекать несколько символов, тогда это будет называться **срез**:

```
>>> s = 'Python'
>>> print(s[1:2])
```

```
y
>>> print(s[0:])
Python
>>> print(s[:3])
Pyt
>>> print(s[:])
Python
```

Методов работы со строками довольно много, мы перечислим часть наиболее полезных на наш взгляд:

- find(str, [start],[end]) - Поиск подстроки в строке. Возвращает номер первого вхождения или -1

```
>>> s = 'PythonohtyP'
>>> s.find('t')
2
```

- rfind(str, [start],[end]) - Поиск подстроки в строке. Возвращает номер последнего вхождения или -1

```
>>> s = 'PythonohtyP'
>>> s.rfind('t')
8
```

- index(str, [start],[end]) – Поиск подстроки в строке. Возвращает номер первого вхождения или вызывает ValueError

```
>>> s = 'Python'
>>> s.index('t')
2
```

- rindex(str, [start],[end]) – Поиск подстроки в строке. Возвращает номер последнего вхождения или вызывает ValueError

```
>>> s = 'PythonohtyP'
>>> s.rindex('t')
8
```

- replace(шаблон, замена) – Замена шаблона

```
>>> s = 'Python'
>>> s.replace('P', 'AAA')
'AAAython'
```

- split(символ) – Разбиение строки по разделителю

```
>>> s = 'Python'
>>> s.split('t')
['Py', 'hon']
```

- isdigit() – Состоит ли строка из цифр

```
>>> s = 'Python'
>>> s.isdigit()
False
```

- isalpha() – Состоит ли строка из букв

```
>>> s = 'Python'
>>> s.isalpha()
True
```

- isalnum() – Состоит ли строка из цифр или букв

```
>>> s = 'Python'
>>> s.isalnum()
True
```

- islower() – Состоит ли строка из символов в нижнем регистре

```
>>> s = 'Python'
>>> s.islower()
False
```

- isupper() – Состоит ли строка из символов в верхнем регистре

```
>>> s = 'Python'
>>> s.isupper()
False
```

- istitle() – Начинаются ли слова в строке с заглавной буквы

```
>>> s = 'Python'
>>> s.istitle
True
```

- upper() – Преобразование строки к верхнему регистру

```
>>> s = 'Python'
>>> s.upper()
'PYTHON'
```

- lower() – Преобразование строки к нижнему регистру

```
>>> s = 'Python'
>>> s.lower()
'python'
```

- startswith(str) – Начинается ли строка S с шаблона str

```
>>> s = 'Python'
>>> s.startswith('P')
True
```

- endswith(str) – Заканчивается ли строка S шаблоном str

```
>>> s = 'Python'
>>> s.endswith('a')
False
```

- join(список) – Сборка строки из списка с разделителем S

```
>>> s = 'Python'
>>> s.join(['a','b','c'])
'aPythonbPythonc'
```

Задачи.

Дана строка: Abrakadabra

Сначала выведите третий символ этой строки.

Во второй строке выведите предпоследний символ этой строки.

В третьей строке выведите первые пять символов этой строки.

В четвертой строке выведите всю строку, кроме последних двух символов.

В пятой строке выведите все символы с четными индексами (считая, что индексация начинается с 0, поэтому символы выводятся начиная с первого).

В шестой строке выведите все символы с нечетными индексами, то есть начиная со второго символа строки.

В седьмой строке выведите все символы в обратном порядке.

В восьмой строке выведите все символы строки через один в обратном порядке, начиная с последнего.

В девятой строке выведите длину данной строки.

Входные данные: Abrakadabra

Выходные данные:

r

r

Abrak

Abrakadab

Arkdba

baaar

arbadakarbA

abdkrA

11

Списки

Для создания списка автоматически можно использовать метод list():

```
>>> list('Python')
['P','y','t','h','o','n']
```

Также можно это сделать напрямую, присвоив переменной значение типа list:

```
 # Пустой список
>>> s = []
# список с данными разных типов
>>> l = ['s', 'p', ['isok'], 2]
>>> s
[]
>>> l
['s', 'p', ['isok'], 2]
```

Как видно из примера выше, можно хранить список в списке(вложенный список).

Также можно использовать генераторы списков:

```
>>> a = [a * 3 for a in 'Python']
>>> a
['PPP', 'yyy', 'ttt', 'hhh', 'ooo', 'nnn']
```

Методы списков.

Методы списков вызываются по схеме: list.method(). Ниже будут перечислены полезные методы для работы со списками:

- append(a) – добавляет элемент a в конец списка

```
>>> var = ['l', 'i', 's', 't']
>>> var.append('a')
>>> print(var)
['l', 'i', 's', 't', 'a']
```

- extend(L) – расширяет список, добавляя к концу все элементы списка L.

```
>>> var = ['l', 'i', 's', 't']
>>> var.extend(['l', 'i', 's', 't'])
>>> print(var)
['l', 'i', 's', 't', 'l', 'i', 's', 't']
```

- insert(i, a) - вставляет на i позицию элемент a.

```
>>> var = ['l', 'i', 's', 't']
>>> var.insert(2,'a')
>>> print(var)
['l', 'i', 'a', 's', 't']
```

- remove(a) - удаляет первый элемент в списке со значением a, возвращает ошибку, если такого элемента не существует.

```
>>> var = ['l', 'i', 's', 't', 't']
>>> var.remove('t')
>>> print(var)
['l', 'i', 's', 't']
```

- pop(i) - удаляет i-ый элемент и возвращает его, если индекс не указан, удаляет последний элемент.

```
>>> var = ['l', 'i', 's', 't']
>>> var.pop(0)
'l'
>>> print(var)
['i', 's', 't']
```

- index(a) – возвращает индекс элемента a (индексация начинается с 0).

```
>>> var = ['l', 'i', 's', 't']
>>> var.index('t')
3
```

- count(a) – возвращает количество элементов со значением a.

```
>>> var = ['l', 'i', 's', 't']
>>> var.count('t')
1
```

- sort([key = функция]) – сортирует список на основе функции, можно не прописывать функцию, тогда сортировка будет происходить по встроенному алгоритму.

```
>>> var = ['l', 'i', 's', 't']
>>> var.sort()
>>> print(var)
['i', 'l', 's', 't']
```

- reverse() – разворачивает список.

```
>>> var = ['l', 'i', 's', 't']
>>> var.reverse()
>>> print(var)
['t', 's', 'i', 'l']
```

- copy() – поверхностная копия списка, при присвоении переменной копии списка, значение данного списка не изменяется в случае изменения первого. Если переменной присвоить список через "=", тогда значение этой переменной будет меняться при изменении оригинала.

```
>>> var = ['l', 'i', 's', 't']
>>> asd = var.copy()
>>> print(asd)
['l', 'i', 's', 't']
```

```
>>> var = ['l', 'i', 's', 't']
>>> asd = var
>>> print(asd)
['l', 'i', 's', 't']
>>> print(var)
['l', 'i', 's', 't']
>>> var.reverse()
>>> print(asd)
['t', 's', 'i', 'l']
>>> print(var)
['t', 's', 'i', 'l']

>>> var = ['l', 'i', 's', 't']
>>> asd = var.copy()
>>> print(asd)
['l', 'i', 's', 't']
>>> print(var)
['l', 'i', 's', 't']
>>> var.reverse()
>>> print(asd)
['l', 'i', 's', 't']
>>> print(var)
['t', 's', 'i', 'l']
```

- clear() – очищает список.

```
>>> var = ['l', 'i', 's', 't']
>>> var.clear()
>>> print(var)
[]
```

Задания.

Задание 1. Выведите все элементы списка с четными индексами (то есть A[0], A[2], A[4], ...).

1) Входные параметры: 1 2 3 4 5. Выходные параметры: 1, 3, 5.

2) Входные параметры: 9 4 5 2 3. Выходные параметры: 9, 5, 3.

3) Входные параметры: 7 8. Выходные параметры: 7.

4) Входные параметры: 90 45 3 43. Выходные параметры: 90, 3.

Задание 2. Дан список чисел. Выведите все элементы списка, которые больше предыдущего элемента.

1) Входные параметры: 1, 5, 2, 4, 3. Выходные параметры: 5, 4.

2) Входные параметры: 1, 2, 3, 4, 5. Выходные параметры: 2, 3, 4, 5.

3) Входные параметры: 5, 4, 3, 2, 1. Выходные параметры:

4) Входные параметры: 1, 5, 1, 5, 1. Выходные параметры: 5, 5.

Задание 3. В списке все элементы различны. Поменяйте местами минимальный и максимальный элемент этого списка.

1) Входные параметры: 3, 4, 5, 2, 1. Выходные параметры: 3, 4, 1, 2, 5.

2) Входные параметры: -3000, 3000. Выходные параметры: 3000, -3000.

3) Входные параметры: 1, 2, 3, 4, 5 ,6 ,7. Выходные параметры: 7, 2, 3, 4, 5, 6, 1

4) Входные параметры: -5, 5, 10. Выходные параметры: 10, 5, -5.

Кортежи

Как говорилось ранее кортежи это неизменяемые списки, нужны они для защиты от дурака. Создать кортеж можно следующим образом:

```
>>> a = (1, 2, 3, 4, 5, 6)
>>> print(a)
(1, 2, 3, 4, 5, 6)
```

Чтобы создать пустой кортеж, необходимо применить метод tuple().

```
>>> a = tuple()
>>> print(a)
()
```

Если вам нужен кортеж из одного элемента, то он создается следующим образом:

```
 >>> a = (1,)
>>> print(a)
(1,)
```

Если не указать на конце запятую, тогда мы получим не кортеж, а элемент того типа, который мы указали:

```
 >>> a = (1)
>>> print(a)
1
```

Кстати необязательно даже указывать скобки, кортеж можно создать и без них:

```
 >>> a = 1, 2, 3, 4
>>> print(a)
(1, 2, 3, 4)
```

Над кортежами работают все операции, работающие со списками, которые не вносят изменения в список.

Словари

Чтобы создать словарь можно использовать метод dict():

```
>>> d = dict(short='dict', long='dictionary')
>>> print(d)
{'short': 'dict', 'long': 'dictionary'}
>>> d = dict([(1, 1), (2, 4)])
>>> print(d)
{1: 1, 2: 4}
```

В созданных выше словарях мы получили пары "ключ-значение", в частности ключ short и соответствующее ему значение dict.

Также можно создать словарь следующим образом:

```
>>> d = {}
>>> print(d)
{}
>>> d = {'dict': 1, 'dictionary': 2}
>>> print(d)
{'dict': 1, 'dictionary': 2}
```

Еще один способ – использовать метод fromkeys():

```
>>> d = dict.fromkeys(['a', 'b'])
>>> print(d)
{'a': None, 'b': None}
>>> d = dict.fromkeys(['a', 'b'], 100)
>>> print(d)
{'a': 100, 'b': 100}
```

Также можно использовать генератор словарей:

```
>>> d = {a: a ** 2 for a in range(7)}
>>> print(d)
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36}
```

Как можно работать со словарями:

```
>>> d = {1: 2, 2: 4, 3: 9}
>>> print(d[1])
2
>>> d[4] = 4 ** 2
>>> print(d)
{1: 2, 2: 4, 3: 9, 4: 16}
>>> d['1']
Traceback (most recent call last):
  File "", line 1, in
```

```
    d['1']
KeyError: '1'
```

В первом случае мы обратились к ключу "1", после чего получили вывод на экран его значения "2". Затем мы создали новый ключ "4" и присвоили ему значение "16", после чего эта пара добавилась к нашему словарю. В последнем примере мы попробовали обратиться к несуществующему ключу, поскольку значение '1' стоит в кавычках, а значит это другой тип данных, которого нет в нашем словаре, после чего получили сообщение о то, что такого ключа нет.

Методы словарей.

Методы словарей вызываются по схеме: dict.method(). Ниже будут перечислены полезные методы для работы со словарями:

- clear() – очищает словарь

```
>>> d = {'a': 1, 'b': 2}
>>> d.clear()
>>> print(d)
{}
```

- copy() – возвращает копию словаря

```
>>> d = {'a': 1, 'b': 2}
>>> b = d.copy()
>>> print(b)
{'a': 1, 'b': 2}
```

- fromkeys(seq[,value]) – создает словарь с ключами из seq и значением value

```
>>> d.fromkeys(['a', 'b'], 10)
{'a' : 10, 'b' : 10}
```

- get(key[, default]) – возвращает значение ключа, но если его нет, возвращает default

```
>>> d = {'a': 1, 'b': 2}
>>> d.get('a')
1
```

- items() – возвращает пары (ключ, значение)

```
>>> d = {'a': 1, 'b': 2}
>>> d.items()
dict_items([('a', 1), ('b', 2)])
```

- keys() – возвращает ключи в словаре

```
>>> d = {'a': 1, 'b': 2}
>>> print(d.keys())
dict_keys(['a', 'b'])
```

- pop(key[, default]) – удаляет ключ и возвращает значение. Если ключа нет, возвращает default

```
>>> d = {'a': 1, 'b': 2}
>>> d.pop('a')
1
>>> print(d)
{'b': 2}
```

- popitem() – удаляет и возвращает пару (ключ, значение) с конца

```
>>> d = {'a': 1, 'b': 2}
>>> d.popitem()
('b', 2)
>>> print(d)
{'a': 1}
```

- setdefault(key[, default]) – возвращает значение ключа, но если его нет, создает ключ с значением default

```
>>> d = {'a': 1, 'b': 2}
>>> d.setdefault('e', 6)
6
>>> d.setdefault('f')
>>> print(d)
{'a': 1, 'b': 2, 'e': 6, 'f': None}
```

- update([other]) – обновляет словарь, добавляя пары (ключ, значение) из other. Существующие ключи перезаписываются

```
>>> d = {'a': 1, 'b': 2}
>>> d.update({'d':5})
```

```
>>> print(d)
{'a': 1, 'b': 2, 'd': 5}
```

- values() – возвращает значения в словаре

```
>>> d = {'a': 1, 'b': 2}
>>> d.values()
dict_values([1, 2])
```

Задания.

Вам дан словарь, состоящий из пар слов. Каждое слово является синонимом к парному ему слову. Все слова в словаре различны.

Для слова из словаря, записанного в последней строке, определите его синоним.

1) Входные параметры:

{'Hello':'Hi', 'Bye': 'Goodbye', 'List' : 'Array'}

'Bye'

Выходные параметры:

'Goodbye'

2) Входные параметры:

{'beep':'car'}

'beep'

Выходные параметры:

'car'

3) Входные параметры:

{'a': 1, 'b': 2, 'c' : 3, 'd' : 4, 'e' : 5}

'c'

Выходные параметры:

3

Множества

set

Как говорилось ранее, множества содержат неповторяющиеся данные в произвольном порядке. Создадим множество несколькими способами:

```
>>> a = set()
>>> print(a)
set()
>>> a = set('hello')
>>> print(a)
{'h', 'o', 'l', 'e'}
>>> a = {'a', 'b', 'c', 'd'}
>>> print(a)
{'b', 'c', 'a', 'd'}
>>> a = {i ** 2 for i in range(10)}
>>> print(a)
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}
```

Множества удобно использовать для удаления повторяющихся элементов:

```
>>> words = ['hello', 'daddy', 'hello', 'mum']
>>> set(words)
{'hello', 'daddy', 'mum'}
```

Методы множеств.

Методы множеств, в основном, вызываются по схеме: set.method(). Ниже будут перечислены полезные методы для работы с множествами:

- len(s) – число элементов в множестве (размер множества)

```
>>> a = {'a', 'b', 'c', 'd'}
>>> len(a)
4
```

- x in s – принадлежит ли x множеству s

```
>>> a = {'a', 'b', 'c', 'd'}
>>> 'a' in a
True
```

- isdisjoint(other) – истина, если set и other не имеют общих элементов

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.isdisjoint('a')
False
>>> a.isdisjoint('f')
True
```

- issubset(other) или set <= other – истина, если все элементы set принадлежат other

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.issubset({'a', 'b', 'c', 'd','f','e'})
True
```

- issuperset(other) или set >= other – аналогично

- union(other, ...) или set | other | ... – возвращает объединение нескольких множеств

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.union({'f','d'})
{'b', 'a', 'c', 'f', 'd'}
```

- intersection(other, ...) или set & other & ... – возвращает пересечение множеств

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.intersection({'f','a'})
{'a'}
```

- difference(other, ...) или set - other - ... –возвращает множество из всех элементов set, не принадлежащие ни одному из other

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.difference({'a','f','d'})
{'b', 'c'}
```

- symmetric_difference(other); set ^ other – возвращает множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.symmetric_difference({'a','d'})
{'b', 'c'}
```

- copy() – копия множества

```
>>> a = {'a', 'b', 'c', 'd'}
>>> d = a.copy()
>>> print(d)
{'d', 'b', 'a', 'c'}
```

- update(other, ...); set |= other | ... – объединение множеств. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.update({'w','z'})
>>> print(a)
{'z', 'b', 'a', 'c', 'w', 'd'}
```

- intersection_update(other, ...); set &= other & ... – пересечение множеств. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.intersection_update({'a','d'})
>>> print(a)
{'a', 'd'}
```

- difference_update(other, ...); set -= other | ... – вычитание множеств. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.difference_update({'a','d'})
>>> print(a)
{'b', 'c'}
```

- symmetric_difference_update(other); set ^= other – множество из элементов, встречающихся в одном множестве, но не встречающиеся в обоих. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.symmetric_difference_update({'a','b'})
>>> print(a)
{'c', 'd'}
```

- add(elem) – добавляет элемент в множество. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.add('r')
>>> print(a)
{'r', 'c', 'a', 'd', 'b'}
```

- remove(elem) – удаляет элемент из множества. KeyError, если такого элемента не существует. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.remove('b')
>>> print(a)
{'c', 'a', 'd'}
```

- discard(elem) – удаляет элемент, если он находится в множестве. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.discard('c')
```

```
>>> print(a)
{'a', 'b', 'd'}
```

- pop() – удаляет первый элемент из множества. Так как множества не упорядочены, нельзя точно сказать, какой элемент будет первым. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.pop()
'c'
>>> print(a)
{'a', 'b', 'd'}
```

- clear() – очистка множества. Метод, вносящий изменения в множество

```
>>> a = {'a', 'b', 'c', 'd'}
>>> a.clear()
>>> print(a)
set()
```

frozenset.

Единственное отличие от set заключается в том, что frozenset не меняется, соответственно, к frozenset можно применить только те методы,которые не меняют множество.

Задания.

Задание 1. Дан список чисел. Определите, сколько в нем встречается различных чисел.

1) Входные значения: 1, 2, 3, 2, 1. Выходные значения: 3

2) Входные значения: 1, 2, 3, 4, 5, 6, 7. Выходные значения: 7

3) Входные значения: 1, 1, 1, 1, 1. Выходные значения: 1

4) Входные значения: 1, 2, 3, 1, 1. Выходные значения: 3

Задание 2. Даны два списка чисел. Посчитайте, сколько чисел содержится одновременно как в первом списке, так и во втором.

1) Входные значения:

1, 2, 3

1, 4, 5

Выходные значения: 1

2) Входные значения:

1, 2, 3, 4, 5, 6, 7

10, 2, 3, 4, 8

Выходные значения: 3

3) Входные значения:

1, 10, 223, 413, 2

2, 40, 12, 100, 10

Выходные значения: 2

Функции

Функции в Python это объекты, принимающие аргументы и возвращающие значение. Функции определяются с помощью инструкции def:

```
def sum(x, y):
    return x + y
```

Мы определили функцию с именем sum, которая принимает 2 аргумента и возвращает их сумму. Мы можем вызвать нашу функцию по ее имени, задав значения параметров:

```
>>> sum(34, 12)
46
>>> sum('abc', 'def')
'abcdef'
```

Функция может быть любой сложности, внутри конструкции def -> return, мы можем написать любой код. Смысл в функциях заключается в том, чтобы не писать один и тот же код повторно, а просто в нужный момент вызывать заранее написанную функцию. Так же функция может быть без параметров не обязательно возвращать какое-то конкретное значение или не заканчиваться инструкцией return вовсе:

```
def fun():
    var = 'Python'
    if len(var) >= 6:
        print(var)
    return
```

Код под инструкцией def будет относиться к функции до тех пор, пока он вложен в эту инструкцию, то есть отступает от def.

Функции могут принимать произвольное количество аргументов:

```
>>> def func(*args):
      return args

>>> func(1, 2, 3, 'abc')
(1, 2, 3, 'abc')
```

Как мы видим в таком случае образуется кортеж из этих аргументов. Также можно принимать аргументы в виде словаря:

```
>>> def func(**kwargs):
         return kwargs

>>> func(a=1, b=2, c=3)
{'a': 1, 'c': 3, 'b': 2}
```

Помимо обычных функций, существуют анонимные или lambda функции, их смысл в том, что они могут содержать всего одно выражение, но выполняются они гораздо быстрее:

```
>>> func = lambda x, y: x + y
>>> func(1, 2)
3
>>> func('a', 'b')
'ab'
>>> (lambda x, y: x + y)(1, 2)
3
>>> (lambda x, y: x + y)('a', 'b')
'ab'
```

Работа с файлами

Python позволяет работать с файлами. Самыми простыми являются текстовые файлы. Прежде чем начать работать с файлом необходимо его открыть, для этого есть функция open():

```
f = open('test.txt', 'r')
```

Мы присваиваем переменной f результат выполнения команды открытия файла, параметры этой команды – адрес файла, путь до места где он расположен на компьютере с названием файла(test) и его расширением(.txt), говорящим что он текстовый. Второй параметр 'r' означает, что файл открыт для чтения, то есть мы не сможем вносить изменения в этот файл до тех пор, пока он открыт только для чтения.

У функции open() существует много аргументов:

- 'r' – открытие на чтение(значение по умолчанию)
- 'w' – открытие на запись, содержимое файла удаляется и перезаписывается заново, если файла не существует, создается новый
- 'x' – открытие на запись, только если файла не существует
- 'a' – открытие на дозапись, информация добавляется в конец файла
- 'b' – открытие файла в двоичном виде
- 't' – открытие в текстовом режиме(значение по умолчанию)

- '+' – открытие на чтение и запись

Возможно сочетание режимов, например 'rb', чтение в бинарном виде, по умолчанию установлен 'rt'.

После открытия файла можно прочитать из него информацию, благодаря методу read():

```
>>> f = open('test.txt')
>>> f.read()
'Python is\nAwesome!.\n\n'
```

Также можно прочитать файл построчно:

```
>>> f = open('test.txt')
>>> for line in f:
        print(line)

'Python is\n'
'\n'
'Awesome\n'
'\n'
```

Записать информацию в файл можно открыв файл на запись:

```
>>> f = open('test.txt', 'w')
>>> for i in range(1, 4):
     f.write(i + '\n')
```

```
1
2
3
```

По окончании работы с файлом его необходимо закрыть, используя метод close():

```
>>> f.close()
```

Если попробовать открыть и прочитать информацию из файла, который мы записали, можно убедиться в присутствии информации в нем.

Подключение модулей

Модулем в Python называется любой файл с программой. То есть любой Ваш код заключенный в файл является модулем. Когда разрабатывается любая программа, она редко ограничивается одним файлом. Обычно это набор файлов. Для того, чтобы не писать один и тот же код в каждом файле, к каждому файлу можно подключить другой файл. Подключив другой модуль, из него можно достать полезный метод, который может понадобиться.

Давайте в качестве примера подключим стандартный модуль datetime, чтобы достать оттуда метод datetime.today():

```
>>> import datetime
>>> print(datetime.datetime.today())
2019-08-13 12:34:49.444292
```

Подключив стандартный модуль, мы достали метод получения текущей даты и времени, после чего вызвали его и получили вывод на экран. На языке Python очень много дополнительных библиотек, которые можно доустановить и затем вызвать необходимые методы для улучшения работоспособности своего кода, а зачастую и раскрывая новые возможности. Прежде чем начать работать с новой для себя библиотекой, стоит ознакомиться с ее методами, чтобы эффективно использовать в своей работе.

Также можно использовать псевдонимы для модулей, чтобы сократить их названия:

```
>>> import datetime as m
>>> print(m.datetime.today())
2019-08-13 12:34:49.444292
```

Можно сделать подключение модуля еще удобнее использовав инструкцию from:

```
>>> from datetime import datetime as m
>>> print(m.today())
2019-08-13 12:34:49.444292

>>> from datetime import *
```

Последняя запись со звездочкой ("*") означает, что мы подключаем все методы, находящиеся в модуле datetime.