

devpractice.ru

Python

Уроки

Второе издание

Книга создана в рамках проекта devpractice.ru.

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Автор и коллектив проекта devpractice.ru не несет ответственности за возможные ошибки и последствия, связанные с использованием материалов из данной книги.

Коммерческое распространение данной книги запрещено. Автор и коллектив проекта devpractice.ru оставляет за собой право коммерческого распространения книги.

Издание второе: исправленное и дополненное.

© devpractice.ru, 2019

© Абдрахманов М.И., 2019

Дорогие друзья! Мы дарим вам книгу “Python. Уроки” абсолютно БЕСПЛАТНО! Если вы хотите поддержать коллектив авторов, то можете помочь проекту devpractice.ru любой суммой.

Yandex.Кошелек: 410011113064717

Сайт визитка для перевода: <https://money.yandex.ru/to/410011113064717>

Страница для поддержки проекта на нашем сайте <http://devpractice.ru/donation/>

Спасибо за помощь, это очень важно для нас!

Предлагаем познакомиться с другими нашими проектами.

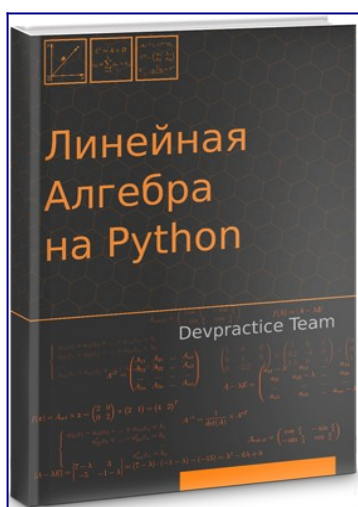
"Pandas. Работа с данными"

Книга посвящена библиотеке для работы с данным *pandas*. Помимо базовых знаний о структурах *pandas*, вы получите информацию о том как работать с временными рядами, считать статистики, визуализировать данные и т.д. Большое внимание уделено практике, все рассматриваемые возможности библиотеки сопровождаются подробными примерами.



"Линейная алгебра на Python"

Книга “Линейная алгебра на *Python*” - это попытка соединить две области: математику и программирование. В ней вы познакомитесь с базовыми разделами линейной алгебры и прекрасным инструментом для решения задач - языком программирования *Python*. Основные разделы книги посвящены матрицам и их свойствам, решению систем линейных уравнений, векторам, разложению матриц и комплексным числам.



Оглавление

Урок 1. Установка.....	7
1.1 Версии <i>Python</i>	7
1.2 Установка <i>Python</i>	7
1.2.1 Установка <i>Python</i> в <i>Windows</i>	7
1.2.2 Установка <i>Python</i> в <i>Linux</i>	11
1.3 Установка <i>Anaconda</i>	11
1.3.1 Установка <i>Anaconda</i> в <i>Windows</i>	11
1.3.2 Установка <i>Anaconda</i> в <i>Linux</i>	15
1.4 Установка <i>PyCharm</i>	18
1.4.1 Установка <i>PyCharm</i> в <i>Windows</i>	18
1.4.2 Установка <i>PyCharm</i> в <i>Linux</i>	21
1.5 Проверка работоспособности.....	22
1.5.1 Проверка интерпретатора <i>Python</i>	22
1.5.2 Проверка <i>Anaconda</i>	23
1.5.3 Проверка <i>PyCharm</i>	25
Урок 2. Запуск программ на <i>Python</i>	28
2.1 Интерактивный режим работы.....	28
2.2 Пакетный режим работы.....	31
Урок 3. Типы и модель данных.....	33
3.1 Кратко о типизации языков программирования.....	33
3.2 Типы данных в <i>Python</i>	33
3.3 Модель данных.....	34
3.4 Изменяемые и неизменяемые типы данных.....	37
Урок 4. Арифметические операции.....	39
4.1 Арифметические операции с целыми и вещественными числами.....	39
4.2 Работа с комплексными числами.....	42
4.3 Битовые операции.....	43
4.4 Представление чисел в других системах счисления.....	44
4.5 Библиотека (модуль) <i>math</i>	44
Урок 5. Условные операторы и циклы.....	47
5.1 Условный оператор ветвления <i>if</i>	47
5.1.1 Конструкция <i>if</i>	47
5.1.2 Конструкция <i>if – else</i>	48
5.1.3 Конструкция <i>if – elif – else</i>	49
5.2 Оператор цикла <i>while</i>	50
5.3 Операторы <i>break</i> и <i>continue</i>	50
5.4 Оператор цикла <i>for</i>	51
Урок 6. Работа с <i>IPython</i> и <i>Jupyter Notebook</i>	53
6.1 Установка и запуск.....	53
6.2 Примеры работы.....	54
6.3 Основные элементы интерфейса <i>Jupyter notebook</i>	56
6.4 Запуск и прерывание выполнения кода.....	58
6.5 Как сделать ноутбук доступным для других людей?.....	59
6.6 Вывод изображений в ноутбуке.....	59
6.7 Магия.....	60
Урок 7. Работа со списками (<i>list</i>).....	62
7.1 Что такое список (<i>list</i>) в <i>Python</i> ?.....	62
7.2 Как списки хранятся в памяти?.....	62

7.3 Создание, изменение, удаление списков и работа с его элементами.....	63
7.4 Методы списков.....	66
7.5 <i>List Comprehensions</i>	69
7.6 <i>List Comprehensions</i> как обработчик списков.....	70
7.7 Слайсы / Срезы.....	72
7.8 “List Comprehensions” ... в генераторном режиме.....	73
Урок 8. Кортежи (<i>tuple</i>).....	75
8.1 Что такое кортеж (<i>tuple</i>) в <i>Python</i> ?.....	75
8.2 Зачем нужны кортежи в <i>Python</i> ?.....	75
8.3 Создание, удаление кортежей и работа с его элементами.....	76
8.3.1 Создание кортежей.....	76
8.3.2 Доступ к элементам кортежа.....	77
8.3.3 Удаление кортежей.....	77
8.3.4 Преобразование кортежа в список и обратно.....	78
Урок 9. Словари (<i>dict</i>).....	79
9.1 Что такое словарь (<i>dict</i>) в <i>Python</i> ?.....	79
9.2 Создание, изменение, удаление словарей и работа с его элементами.....	79
9.2.1 Создание словаря.....	79
9.2.2 Добавление и удаление элемента.....	80
9.2.3 Работа со словарем.....	80
9.3 Методы словарей.....	81
Урок 10. Функции в <i>Python</i>	84
10.1 Что такое функция в <i>Python</i> ?.....	84
10.2 Создание функций.....	84
10.3 Работа с функциями.....	84
10.4 <i>Lambda</i> -функции.....	86
Урок 11. Работа с исключениями.....	87
11.1 Исключения в языках программирования.....	87
11.2 Синтаксические ошибки в <i>Python</i>	88
11.3 Исключения в <i>Python</i>	88
11.4 Иерархия исключений в <i>Python</i>	89
11.5 Обработка исключений в <i>Python</i>	90
11.6 Использование <i>finally</i> в обработке исключений.....	94
11.7 Генерация исключений в <i>Python</i>	94
11.8 Пользовательские исключения (<i>User-defined Exceptions</i>) в <i>Python</i>	95
Урок 12. Ввод-вывод данных. Работа с файлами.....	96
12.1 Вывод данных в консоль.....	96
12.2 Ввод данных с клавиатуры.....	97
12.3 Работа с файлами.....	98
12.3.1 Открытие и закрытие файла.....	98
12.3.2 Чтение данных из файла.....	99
12.3.3 Запись данных в файл.....	100
12.3.4 Дополнительные методы для работы с файлами.....	100
Урок 13. Модули и пакеты.....	102
13.1 Модули в <i>Python</i>	102
13.1.1 Что такое модуль в <i>Python</i> ?.....	102
13.1.2 Как импортировать модули в <i>Python</i> ?.....	102
13.2 Пакеты в <i>Python</i>	104
13.2.1 Что такое пакет в <i>Python</i> ?.....	104
13.2.2 Использование пакетов в <i>Python</i>	105

Урок 14. Классы и объекты.....	106
14.1 Основные понятия объектно-ориентированного программирования.....	106
14.2 Классы в <i>Python</i>	107
14.2.1 Создание классов и объектов.....	107
14.2.2 Статические и динамические атрибуты класса.....	108
14.2.3 Методы класса.....	110
14.2.4 Конструктор класса и инициализация экземпляра класса.....	111
14.2.5 Что такое <i>self</i> ?.....	111
14.2.6 Уровни доступа атрибута и метода.....	112
14.2.7 Свойства.....	114
14.3 Наследование.....	116
14.4 Полиморфизм.....	119
Урок 15. Итераторы и генераторы.....	122
15.1 Итераторы в языке <i>Python</i>	122
15.2 Создание собственных итераторов.....	123
15.3 Генераторы.....	125
Урок 16. Установка пакетов в <i>Python</i>	126
16.1 Где взять отсутствующий пакет?.....	126
16.2 Менеджер пакетов в <i>Python</i> – <i>pip</i>	126
16.3 Установка <i>pip</i>	127
16.4 Обновление <i>pip</i>	128
16.5 Использование <i>pip</i>	128
Урок 17. Виртуальные окружения.....	130
17.1 Что такое виртуальное окружение и зачем оно нужно?.....	130
17.2 ПО позволяющее создавать виртуальное окружение в <i>Python</i>	130
17.3 <i>virtualenv</i>	131
17.3.1 Установка <i>virtualenv</i>	131
17.3.2 Создание виртуального окружения.....	132
17.3.3 Активация виртуального окружения.....	133
17.3.4 Деактивация виртуального окружения.....	134
17.4 <i>venv</i>	134
17.4.1 Создание виртуального окружения.....	134
17.4.2 Активация виртуального окружения.....	134
17.4.3 Деактивация виртуального окружения.....	135
17.5 Полезные ссылки.....	135
Урок 18. Аннотация типов в <i>Python</i>	136
18.1 Зачем нужны аннотации?.....	136
18.2 Контроль типов в <i>Python</i>	136
18.3 Обзор <i>PEP</i> ’ов регламентирующий работу с аннотациями.....	137
18.4 Использование аннотаций в функциях.....	138
18.4.1 Указание типов аргументов и возвращаемого значения.....	138
18.4.2 Доступ к аннотациям функции.....	139
18.5 Аннотация переменных.....	139
18.5.1 Создание аннотированных переменных.....	139
18.5.2 Контроль типов с использованием аннотаций.....	140
18.6 Отложенная проверка аннотаций.....	141
Урок 19. Декораторы функций в <i>Python</i>	143
19.1 Что нужно знать о функциях в <i>Python</i> ?.....	143
19.1.1 Функция как объект.....	143
19.1.2 Функция внутри функции.....	144

19.2 Что такое декоратор функции в <i>Python</i> ?	145
19.2.1 Создание декоратора	145
19.2.2 Передача аргументов в функцию через декоратор	147
19.2.3 Декораторы для методов класса	148
19.2.4 Возврат результата работы функции через декоратор	148
Урок 20. Объектная модель в <i>Python</i>	150
20.1 Что такое объектная модель <i>Python</i> ?	150
20.2 Специальные методы	151
20.3 Примеры	153

Урок 1. Установка

1.1 Версии *Python*

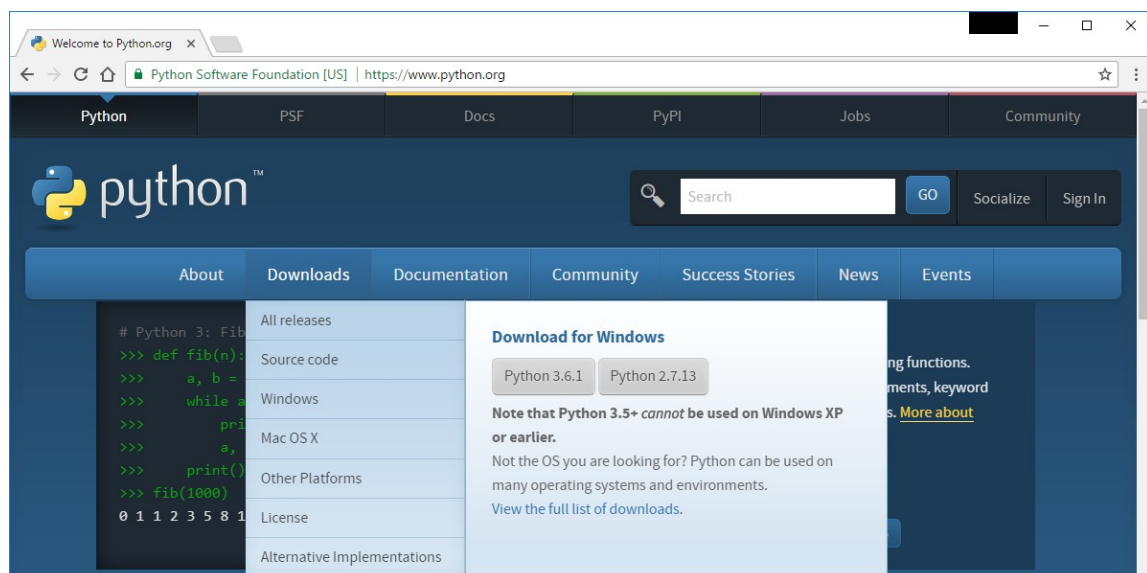
На сегодняшний день существуют две версии *Python* – это *Python 2* и *Python 3*, у них отсутствует полная совместимость друг с другом. На момент написания книги вторая версия *Python* ещё широко используется, но, судя по изменениям, которые происходят, со временем, она останется только для запуска старого кода. В нашей с вами работе, мы будем использовать *Python 3*, и, в дальнейшем, если где-то будет встречаться слово *Python*, то под ним следует понимать *Python 3*. Случаи применения *Python 2* будут специально оговариваться.

1.2 Установка *Python*

Для установки интерпретатора *Python* на ваш компьютер, первое, что нужно сделать – это скачать дистрибутив. Загрузить его можно с официального сайта, перейдя по ссылке <https://www.python.org/downloads/>.

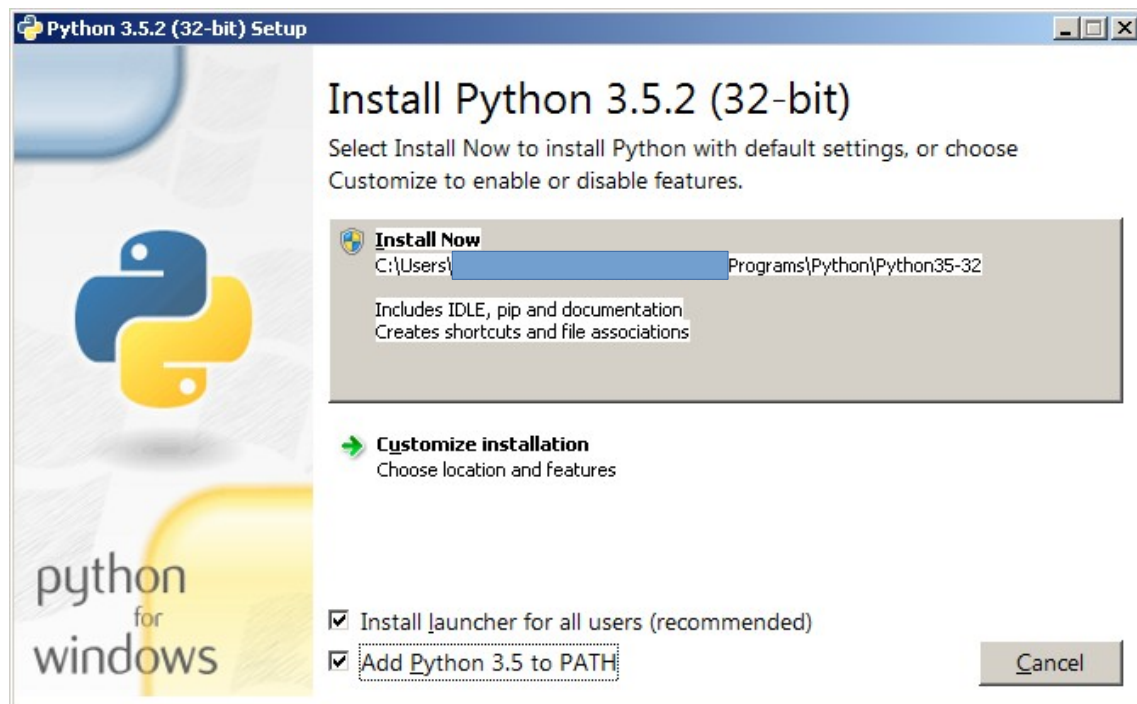
1.2.1 Установка *Python* в *Windows*

Для операционной системы *Windows* дистрибутив распространяется либо в виде исполняемого файла, либо в виде архивного файла. **Если вы используете *Windows 7*, не забудьте установить Service Pack 1!**



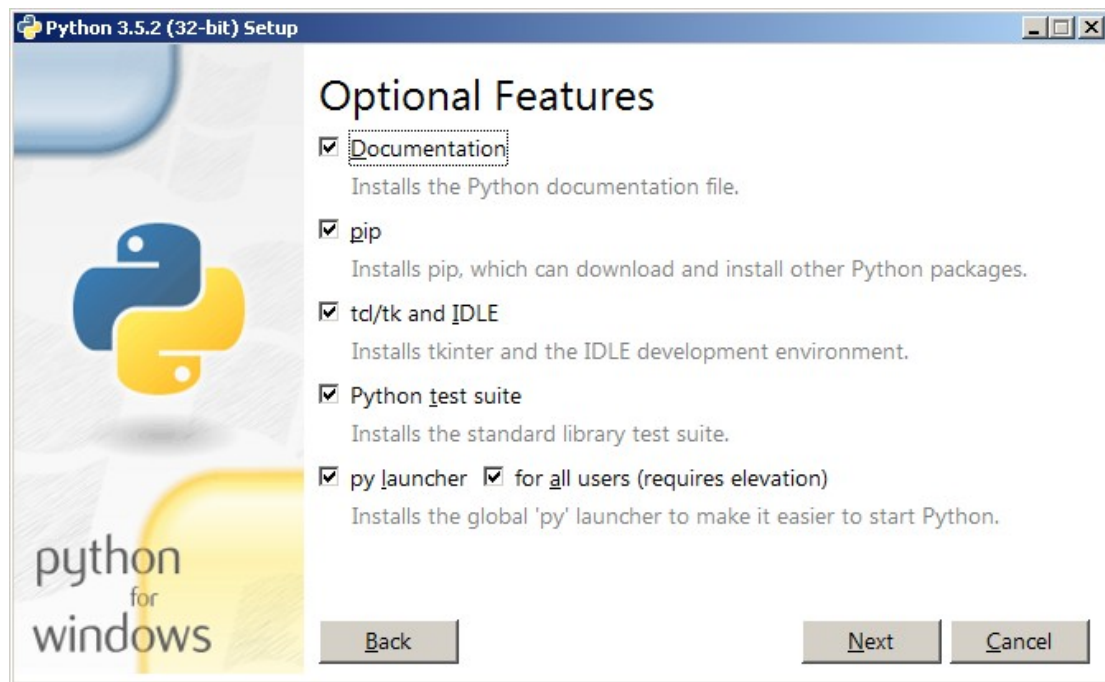
Порядок установки.

1. Запустите скачанный установочный файл.
2. Выберите способ установки.



В данном окне предлагается два варианта *Install Now* и *Customize installation*. При выборе *Install Now*, *Python* установится в папку по указанному пути. Помимо самого интерпретатора будет установлен *IDLE* (интегрированная среда разработки), *pip* (пакетный менеджер) и документация, а также будут созданы соответствующие ярлыки и установлены связи файлов, имеющие расширение *.py* с интерпретатором *Python*. *Customize installation* – это вариант настраиваемой установки. Опция *Add python 3.5 to PATH* нужна для того, чтобы появилась возможность запускать интерпретатор без указания полного пути до исполняемого файла при работе в командной строке.

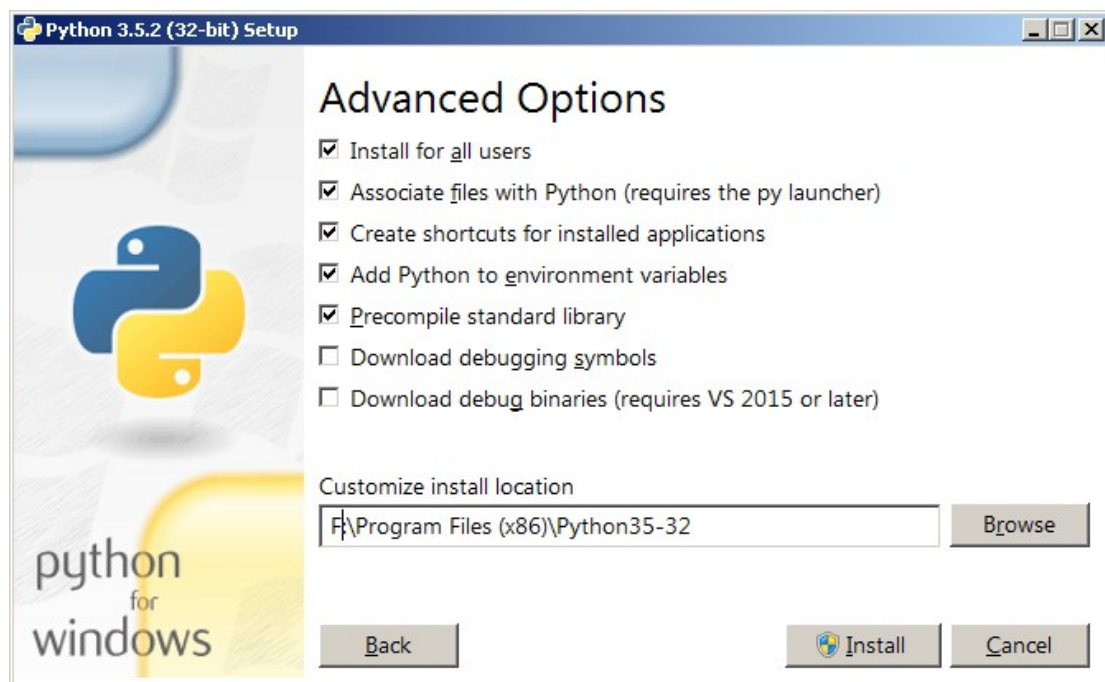
3. Отметьте необходимые опций установки (доступно при выборе *Customize installation*)



На этом шаге нам предлагается отметить дополнения, устанавливаемые вместе с интерпретатором *Python*. Рекомендуем выбрать все опции.

- *Documentation* – установка документаций.
- *pip* – установка пакетного менеджера *pip*.
- *tcl/tk and IDLE* – установка интегрированной среды разработки (*IDLE*) и библиотеки для построения графического интерфейса (*tkinter*).

4. Выберите место установки (доступно при выборе *Customize installation*)



Помимо указания пути, данное окно позволяет внести дополнительные изменения в процесс установки с помощью опций:

- *Install for all users* – Установить для всех пользователей. Если не выбрать данную опцию, то будет предложен вариант инсталляции в папку пользователя, устанавливающего интерпретатор.
- *Associate files with Python* – Связать файлы, имеющие расширение *.py*, с *Python*. При выборе данной опции будут внесены изменения в *Windows*, позволяющие запускать *Python* скрипты по двойному щелчку мыши.
- *Create shortcuts for installed applications* – Создать ярлыки для запуска приложений.
- *Add Python to environment variables* – Добавить пути до интерпретатора *Python* в переменную *PATH*.
- *Precompile standard library* – Провести прекомпиляцию стандартной библиотеки.

Последние два пункта (*Download debugging symbols*, *Download debug binaries*) связаны с загрузкой компонентов для отладки, их мы устанавливать не будем.

5. После успешной установки вас ждет следующее сообщение.



1.2.2 Установка *Python* в *Linux*

Чаще всего интерпретатор *Python* уже входит в состав дистрибутива. Это можно проверить набрав в терминале

```
> python
```

или

```
> python3
```

В первом случае, вы запустите *Python 2* во втором – *Python 3*. В будущем, скорее всего, во всех дистрибутивах *Linux*, включающих *Python*, будет входить только третья версия. Если у вас, при попытке запустить *Python*, выдается сообщение о том, что он не установлен, или установлен, но не тот, что вы хотите, то у вас есть два пути: а) собрать *Python* из исходников; б) взять из репозитория.

Для установки из репозитория в *Ubuntu* воспользуйтесь командой:

```
> sudo apt-get install python3
```

Сборку из исходников рассматривать не будем.

1.3 Установка *Anaconda*

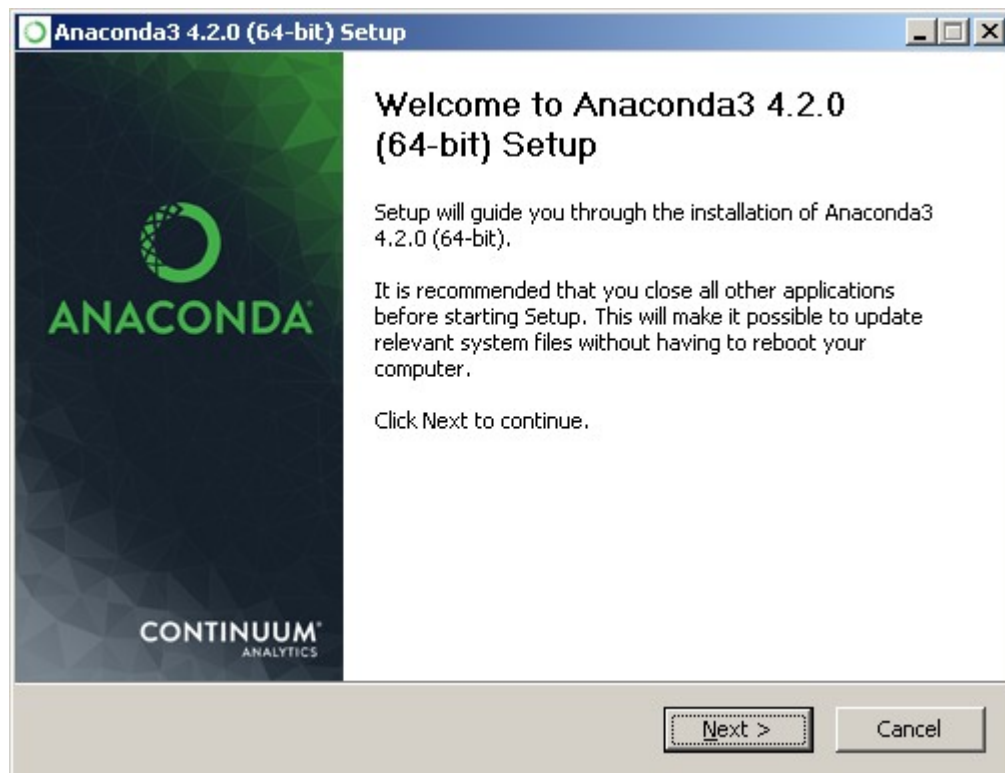
Для удобства запуска примеров и изучения языка *Python*, советуем установить на свой ПК пакет *Anaconda*. Этот пакет включает в себя интерпретатор языка *Python* (есть версии 2 и 3), набор наиболее часто используемых библиотек и удобную среду разработки и исполнения, запускаемую в браузере.

Для установки этого пакета, предварительно нужно скачать дистрибутив <https://www.continuum.io/downloads>.

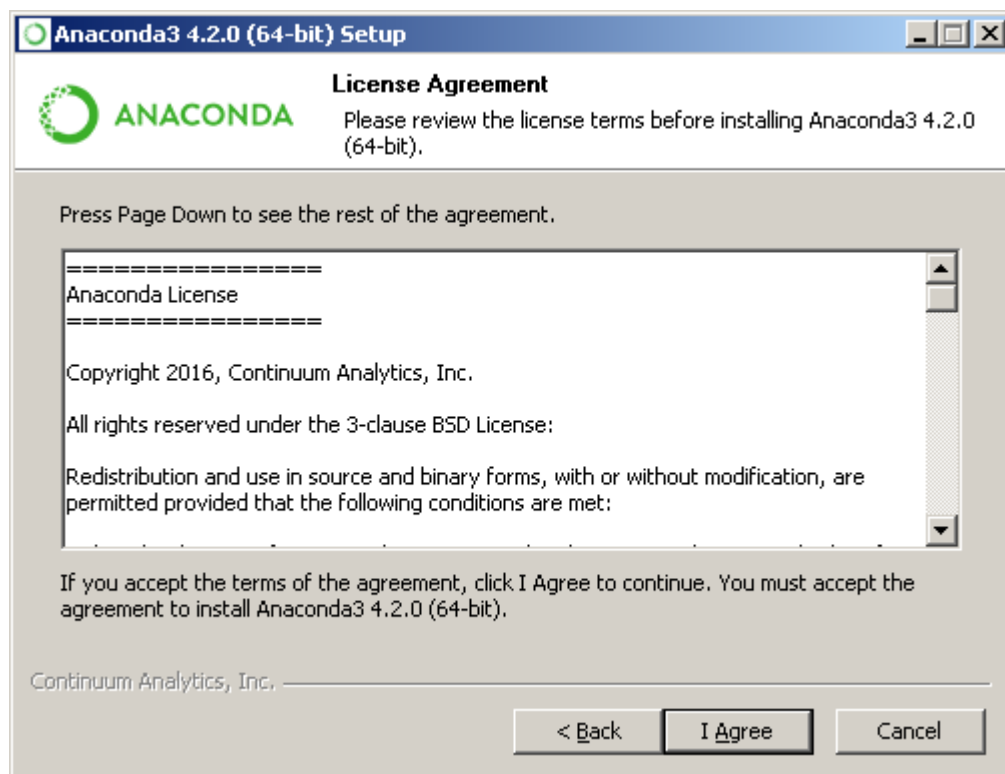
Есть варианты под *Windows*, *Linux* и *Mac OS*.

1.3.1 Установка *Anaconda* в *Windows*

1. Запустите скачанный инсталлятор. В первом появившемся окне необходимо нажать “*Next*”.

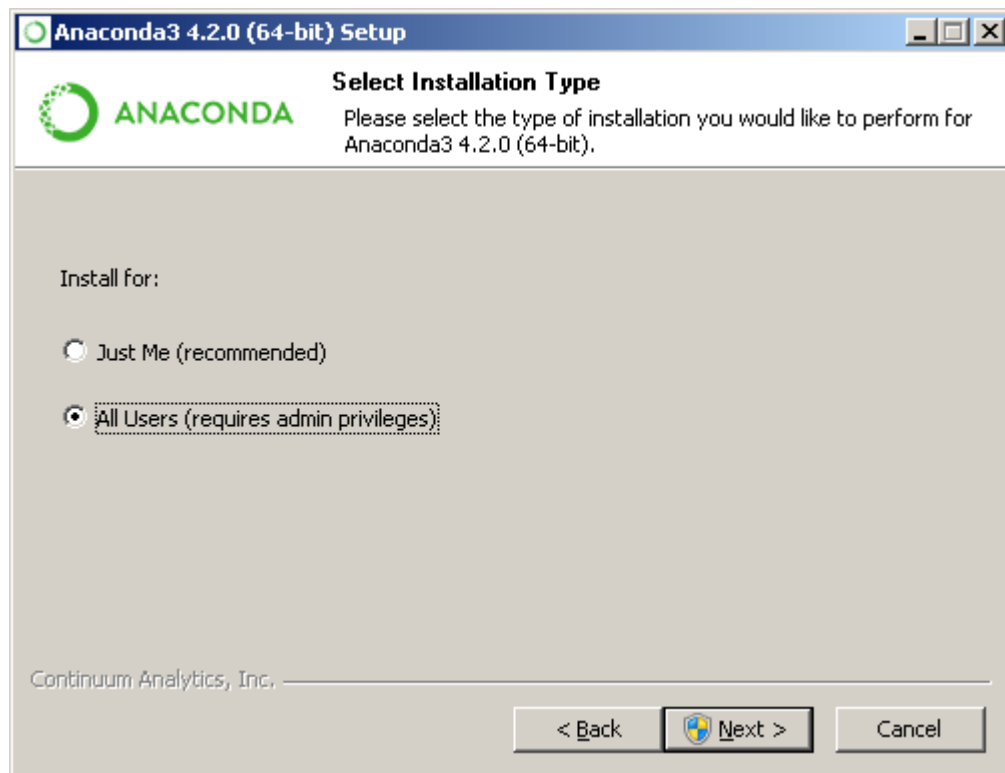


2. Далее следует принять лицензионное соглашение.

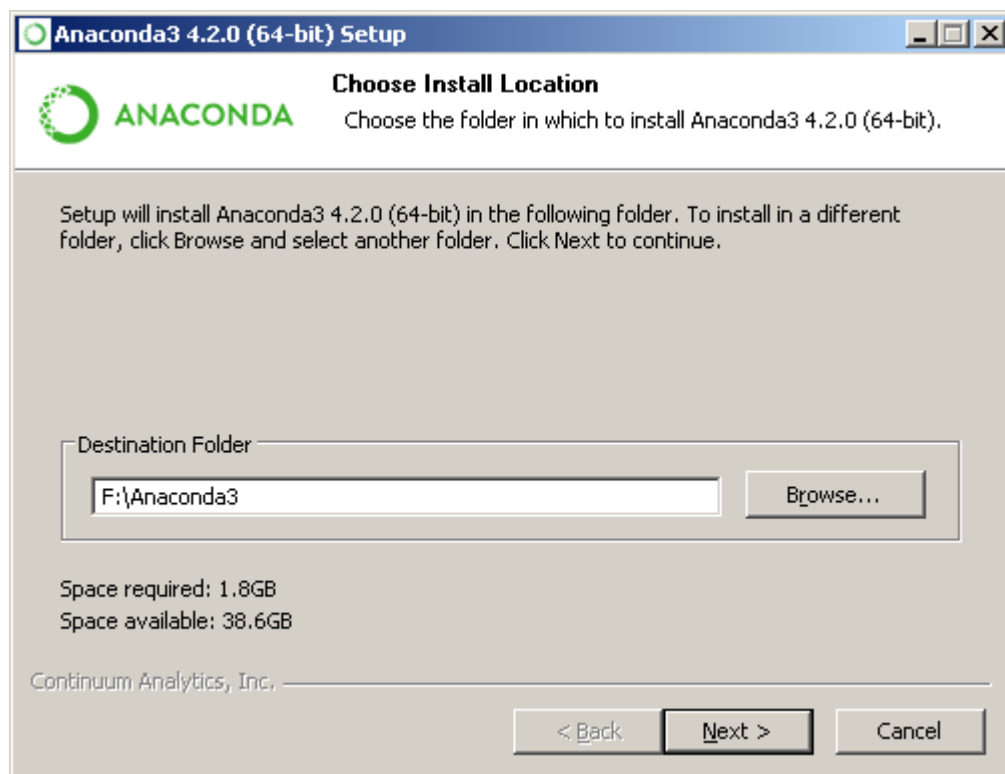


3. Выберите одну из опций установки:

- *Just Me* – только для пользователя, запустившего установку;
- *All Users* – для всех пользователей.



4. Укажите путь, по которому будет установлена *Anaconda*.

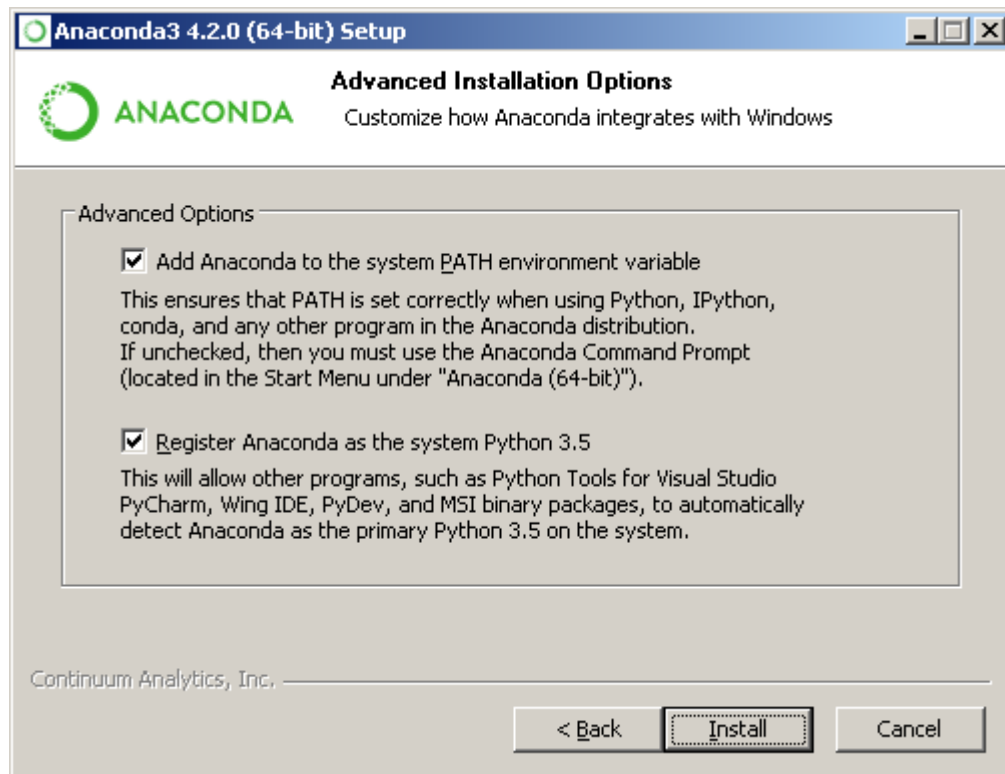


5. Укажите дополнительные опции:

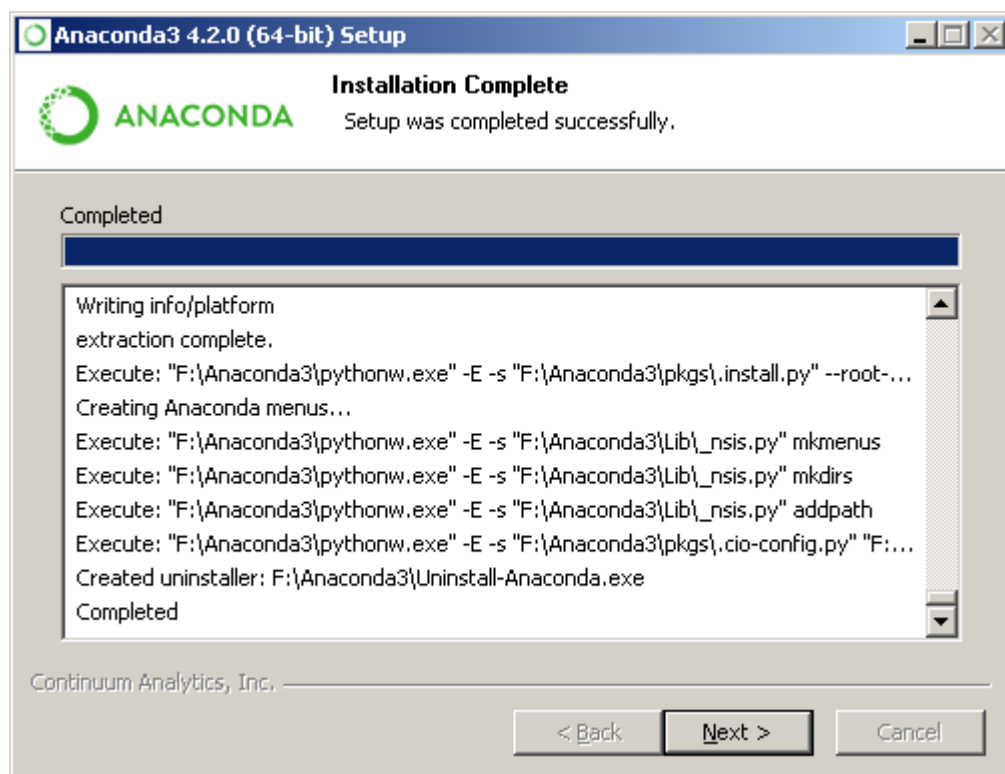
- *Add Anaconda to the system PATH environment variable* – добавить *Anaconda* в системную переменную *PATH*

- *Register Anaconda as the system Python 3.5* – использовать *Anaconda*, как интерпретатор *Python 3.5* по умолчанию.

Для начала установки нажмите на кнопку *"Install"*.



5. После этого будет произведена установка *Anaconda* на ваш компьютер.

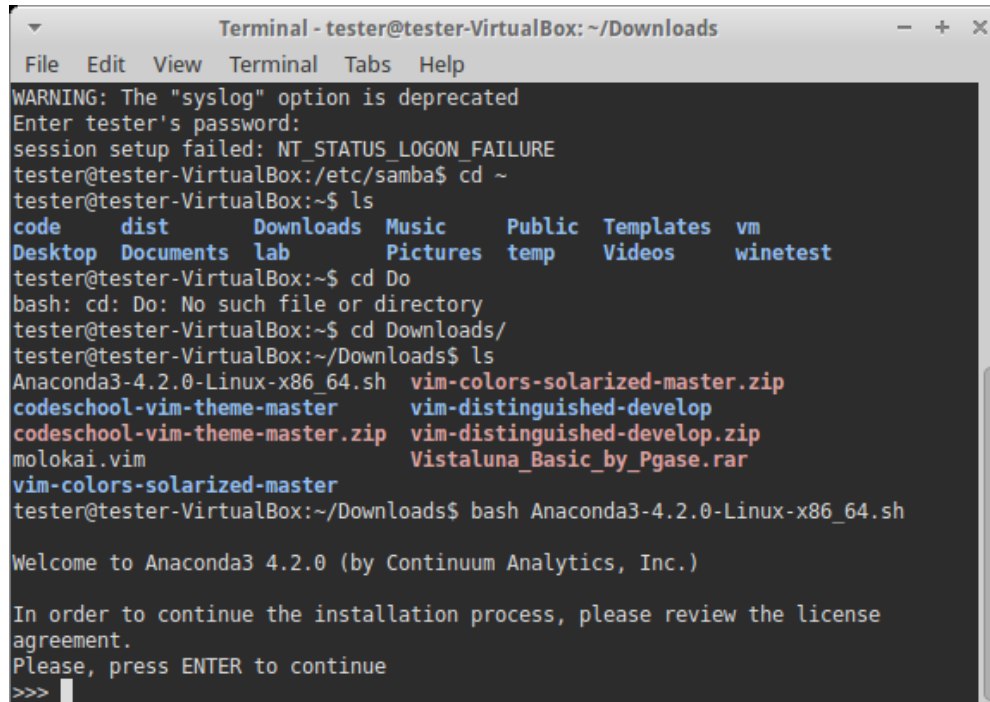


1.3.2 Установка *Anaconda* в *Linux*

1. Скачайте дистрибутив *Anaconda* для Linux, он будет иметь расширение *.sh*, и запустите установку командой:

```
> bash имя_дистрибутива.sh
```

В результате вы увидите приглашение к установке. Для продолжения процессе нажмите “*Enter*”.

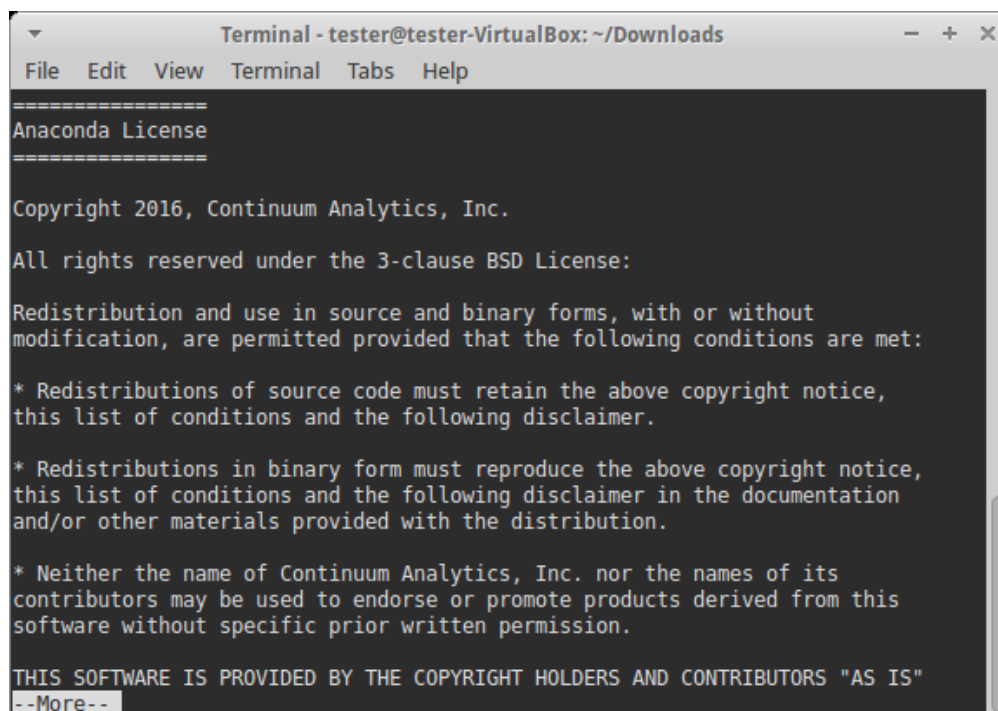


```
Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
WARNING: The "syslog" option is deprecated
Enter tester's password:
session setup failed: NT_STATUS_LOGON_FAILURE
tester@tester-VirtualBox:/etc/samba$ cd ~
tester@tester-VirtualBox:~$ ls
code  dist  Downloads  Music  Public  Templates  vm
Desktop  Documents  lab  Pictures  temp  Videos  winetest
tester@tester-VirtualBox:~$ cd Do
bash: cd: Do: No such file or directory
tester@tester-VirtualBox:~$ cd Downloads/
tester@tester-VirtualBox:~/Downloads$ ls
Anaconda3-4.2.0-Linux-x86_64.sh  vim-colors-solarized-master.zip
codeschool-vim-theme-master      vim-distinguished-develop
codeschool-vim-theme-master.zip  vim-distinguished-develop.zip
molokai.vim                      Vistaluna_Basic_by_Pgase.rar
vim-colors-solarized-master
tester@tester-VirtualBox:~/Downloads$ bash Anaconda3-4.2.0-Linux-x86_64.sh

Welcome to Anaconda3 4.2.0 (by Continuum Analytics, Inc.)

In order to continue the installation process, please review the license
agreement.
Please, press ENTER to continue
>>> |
```

2. Прочитайте лицензионное соглашение, его нужно пролистать до конца.



```
Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
=====
Anaconda License
=====

Copyright 2016, Continuum Analytics, Inc.

All rights reserved under the 3-clause BSD License:

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

* Redistributions of source code must retain the above copyright notice,
this list of conditions and the following disclaimer.

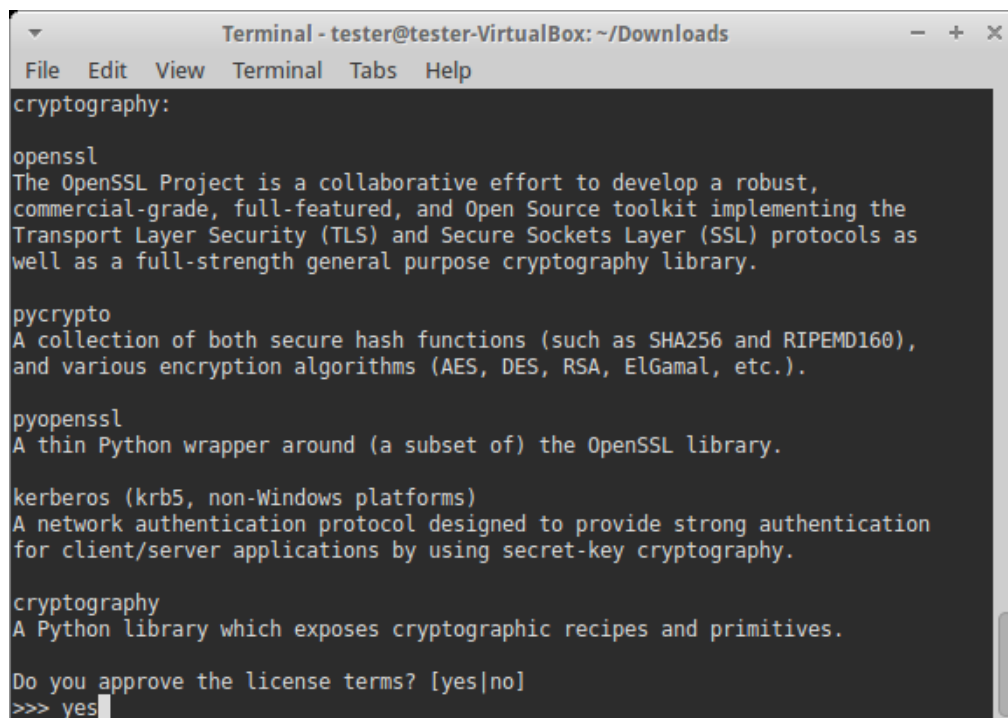
* Redistributions in binary form must reproduce the above copyright notice,
this list of conditions and the following disclaimer in the documentation
and/or other materials provided with the distribution.

* Neither the name of Continuum Analytics, Inc. nor the names of its
contributors may be used to endorse or promote products derived from this
software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
--More--
```


Согласитесь с ним, для этого требуется набрать в командной строке “yes”, в ответе на вопрос инсталлятора:

Do you approve the license terms? [yes|no]



```
Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
cryptography:
openssl
The OpenSSL Project is a collaborative effort to develop a robust,
commercial-grade, full-featured, and Open Source toolkit implementing the
Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols as
well as a full-strength general purpose cryptography library.

pycrypto
A collection of both secure hash functions (such as SHA256 and RIPEMD160),
and various encryption algorithms (AES, DES, RSA, ElGamal, etc.).

pyopenssl
A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
A network authentication protocol designed to provide strong authentication
for client/server applications by using secret-key cryptography.

cryptography
A Python library which exposes cryptographic recipes and primitives.

Do you approve the license terms? [yes|no]
>>> yes
```

3. Выберите место установки. Можно выбрать один из следующих вариантов:

- *Press ENTER to confirm the location* – нажмите *ENTER* для принятия предложенного пути установки. Путь по умолчанию для моей машины: */home/tester/anaconda3*, он представлен чуть выше данного меню.
- *Press CTRL-C to abort the installation* – нажмите *CTRL-C* для отмены установки.
- *Or specify a different location below* – или укажите другой путь в строке ниже.

Нажмите *ENTER*.

```
Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
A collection of both secure hash functions (such as SHA256 and RIPEMD160),
and various encryption algorithms (AES, DES, RSA, ElGamal, etc.).

pyopenssl
A thin Python wrapper around (a subset of) the OpenSSL library.

kerberos (krb5, non-Windows platforms)
A network authentication protocol designed to provide strong authentication
for client/server applications by using secret-key cryptography.

cryptography
A Python library which exposes cryptographic recipes and primitives.

Do you approve the license terms? [yes|no]
>>> yes

Anaconda3 will now be installed into this location:
/home/tester/anaconda3

- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below

[/home/tester/anaconda3] >>> 
```

4. После этого начнется установка.

```
Terminal - tester@tester-VirtualBox: ~/Downloads
File Edit View Terminal Tabs Help
installing: xz-5.2.2-0 ...
installing: yaml-0.1.6-0 ...
installing: zeromq-4.1.4-0 ...
installing: zlib-1.2.8-3 ...
installing: anaconda-4.2.0-np11py35_0 ...
installing: ruamel_yaml-0.11.14-py35_0 ...
installing: conda-4.2.9-py35_0 ...
installing: conda-build-2.0.2-py35_0 ...
Python 3.5.2 :: Continuum Analytics, Inc.
creating default environment...
installation finished.
Do you wish the installer to prepend the Anaconda3 install location
to PATH in your /home/tester/.bashrc ? [yes|no]
[no] >>>
You may wish to edit your .bashrc or prepend the Anaconda3 install location:

$ export PATH=/home/tester/anaconda3/bin:$PATH

Thank you for installing Anaconda3!

Share your notebooks and packages on Anaconda Cloud!
Sign up for free: https://anaconda.org

tester@tester-VirtualBox:~/Downloads$ 
```

1.4 Установка *PyCharm*

Если в процессе разработки вам необходим отладчик и вообще вы привыкли работать в *IDE*, а не в текстовом редакторе, то тогда одним из лучших вариантов будет *IDE PyCharm* от *JetBrains*. Для скачивания данного продукта нужно перейти по ссылке <https://www.jetbrains.com/pycharm/download/>

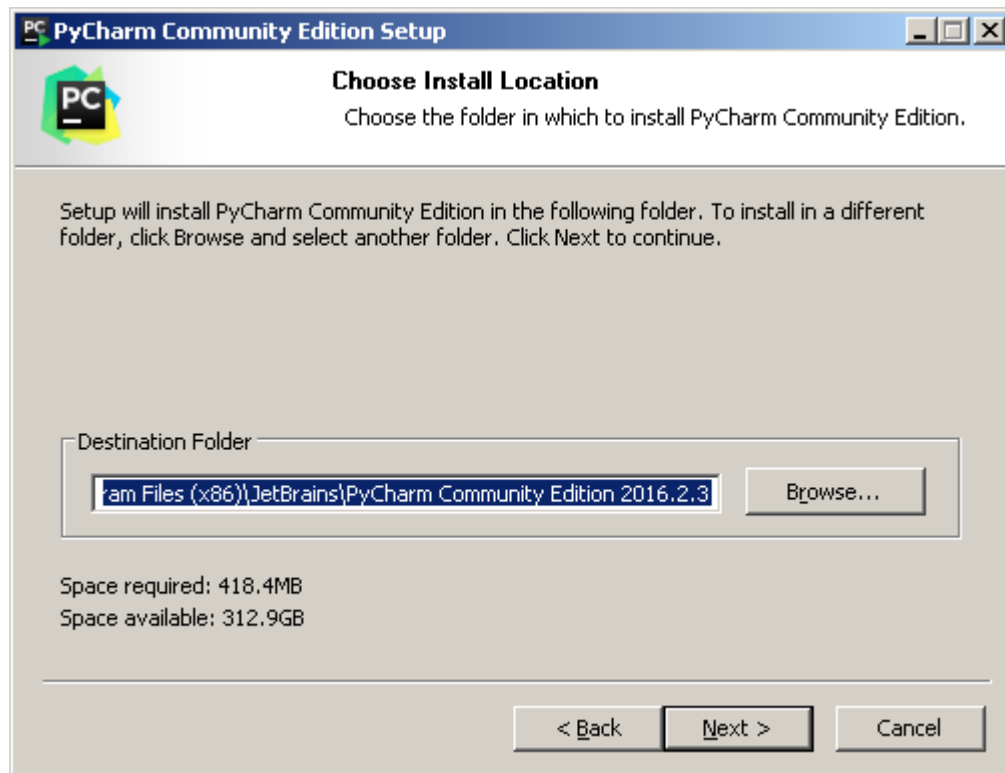
IDE доступна для *Windows*, *Linux* и *Mac OS*. Существуют два вида лицензии *PyCharm* – это *Professional* и *Community*. Мы будем использовать версию *Community*, так как она бесплатна и её функционала более чем достаточно для наших задач.

1.4.1 Установка *PyCharm* в *Windows*

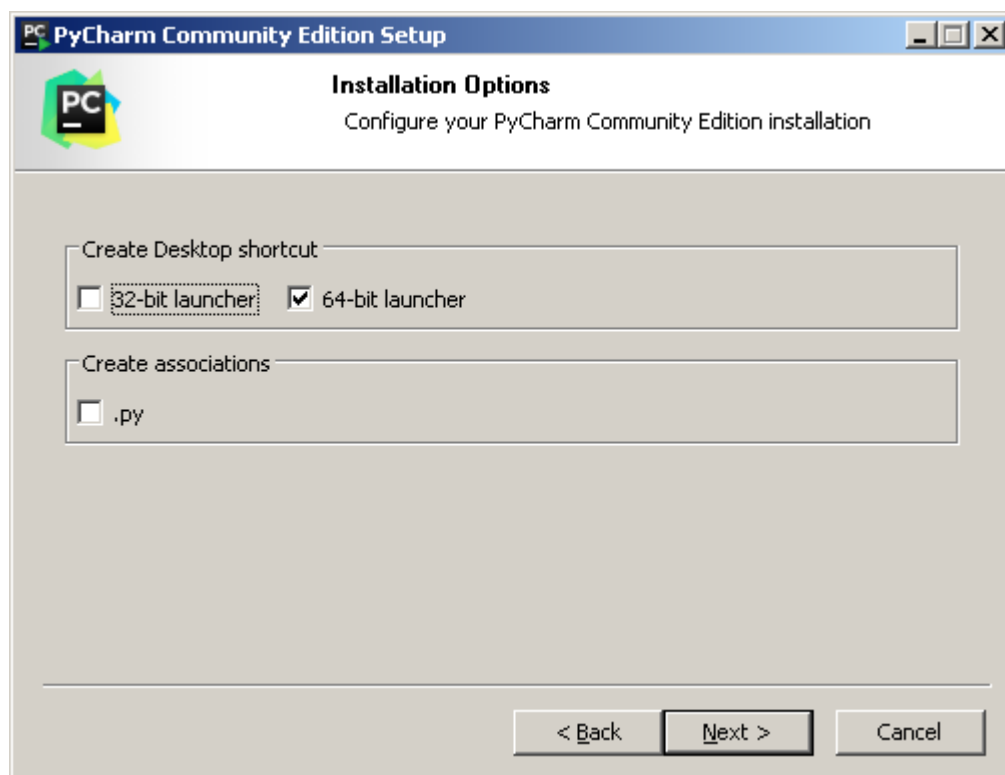
1. Запустите скачанный дистрибутив *PyCharm*.



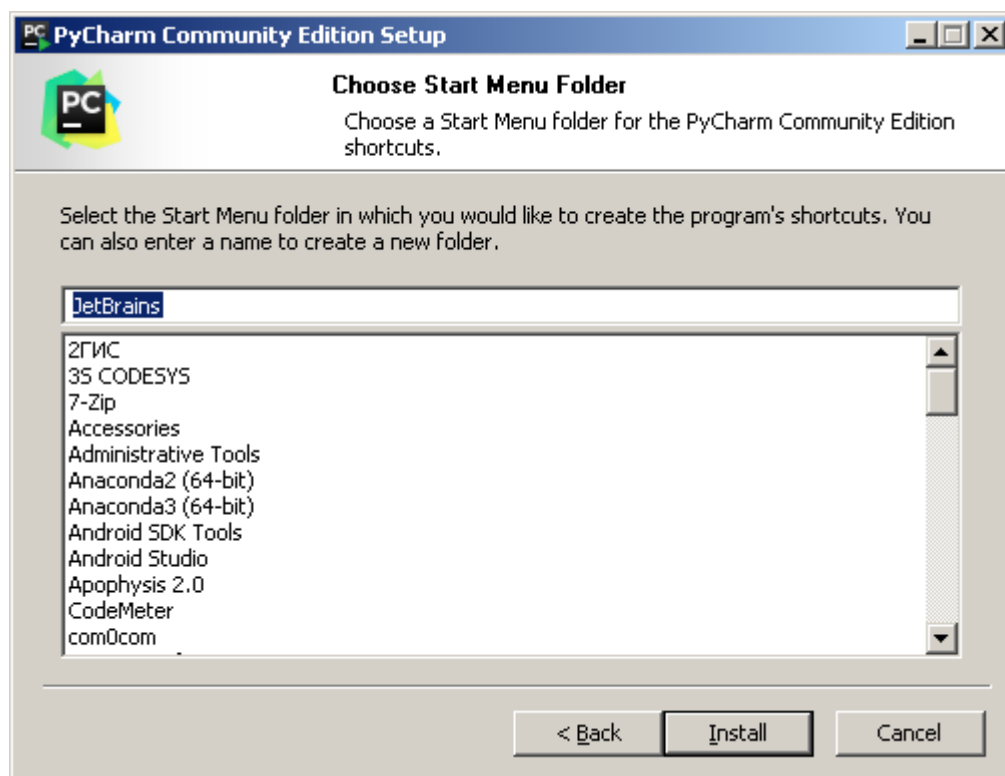
2. Выберите путь установки программы.



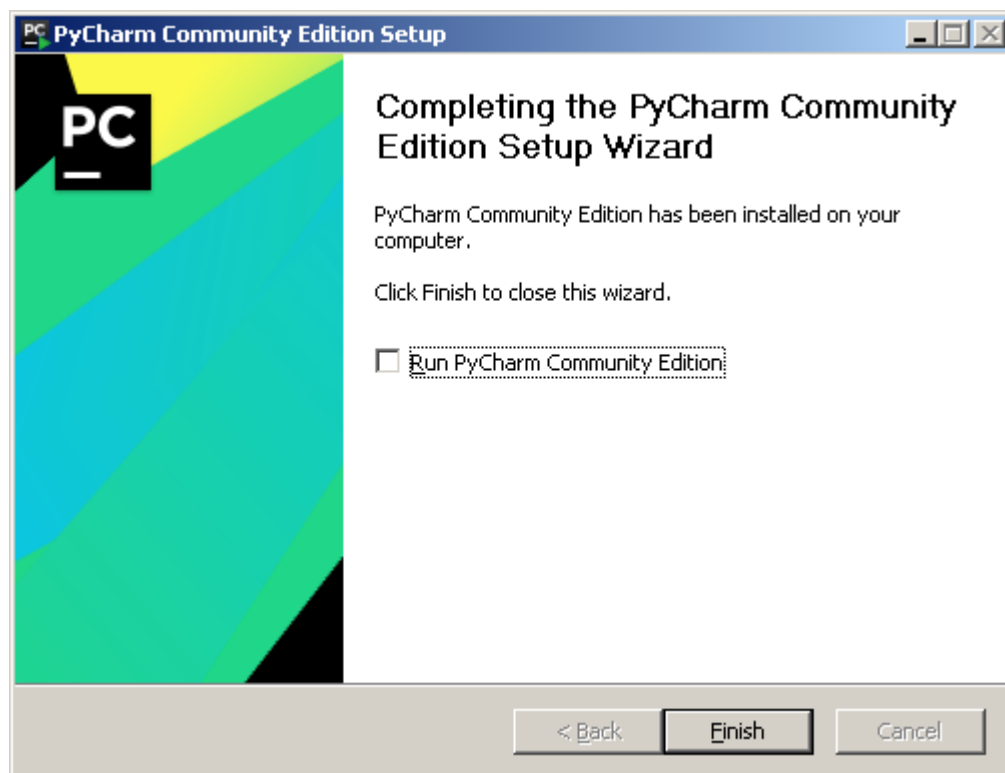
3. Укажите ярлыки, которые нужно создать на рабочем столе (запуск 32-х и 64-х разрядной версии *PyCharm*) и отметить опцию из блока *Create associations* если требуется связать файлы с расширением *.py* с *PyCharm*.



4. Выберите имя для папки в меню Пуск.



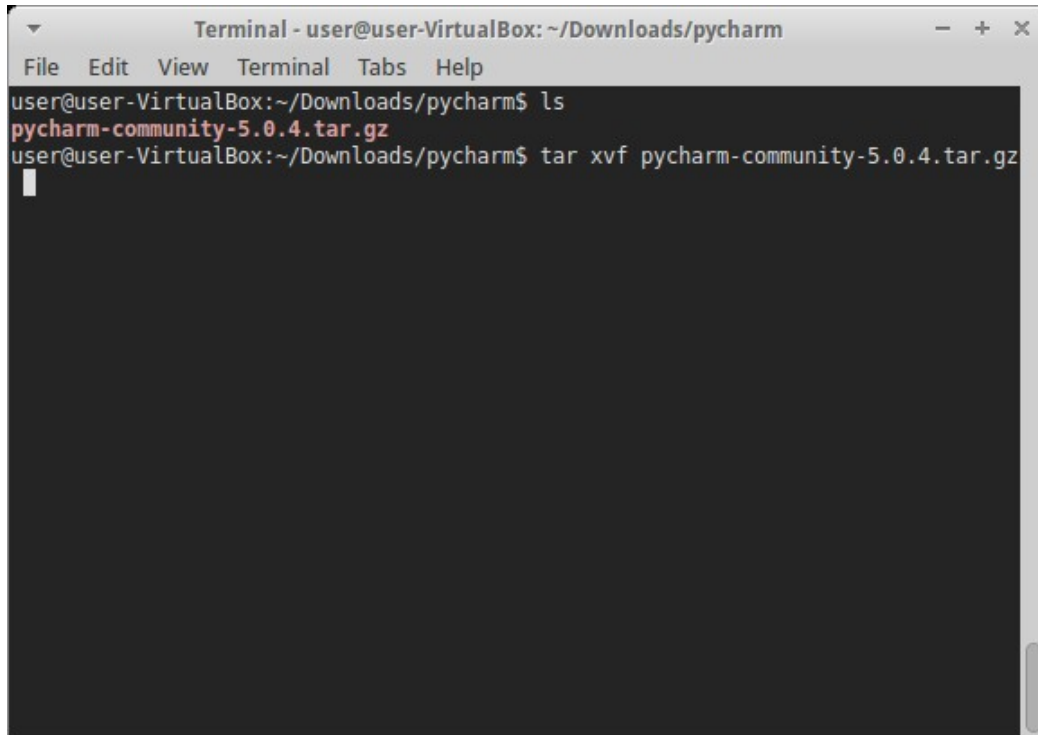
5. Далее *PyCharm* будет установлен на ваш компьютер.



1.4.2 Установка *PyCharm* в *Linux*

1. Скачайте с сайта дистрибутив на компьютер.
2. Распакуйте архивный файл, для этого можно воспользоваться командой:

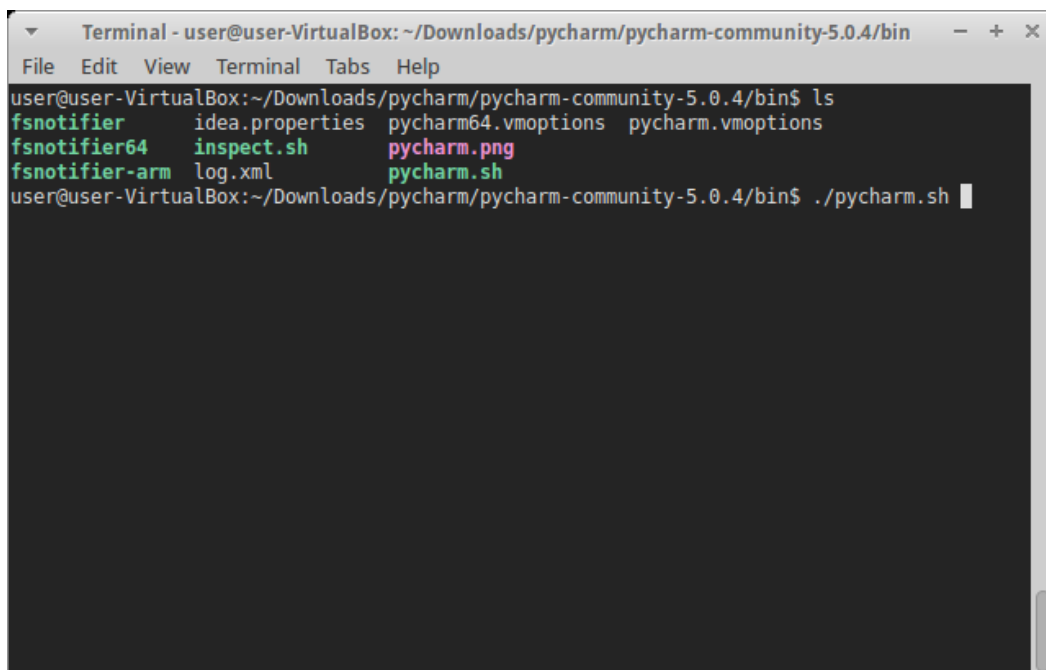
> `tar xvf имя_архива.tar.gz`



```
Terminal - user@user-VirtualBox: ~/Downloads/pycharm
File Edit View Terminal Tabs Help
user@user-VirtualBox:~/Downloads/pycharm$ ls
pycharm-community-5.0.4.tar.gz
user@user-VirtualBox:~/Downloads/pycharm$ tar xvf pycharm-community-5.0.4.tar.gz
```

Перейдите в каталог, который был создан после распаковки дистрибутива, найдите в нем подкаталог *bin* и зайдите в него. Запустите *pycharm.sh* командой:

> `./pycharm.sh`



```
Terminal - user@user-VirtualBox: ~/Downloads/pycharm/pycharm-community-5.0.4/bin
File Edit View Terminal Tabs Help
user@user-VirtualBox:~/Downloads/pycharm/pycharm-community-5.0.4/bin$ ls
fsnotifier        idea.properties  pycharm64.vmoptions  pycharm.vmoptions
fsnotifier64      inspect.sh        pycharm.png
fsnotifier-arm    log.xml           pycharm.sh
user@user-VirtualBox:~/Downloads/pycharm/pycharm-community-5.0.4/bin$ ./pycharm.sh
```

В результате должен запускаться *PyCharm*.

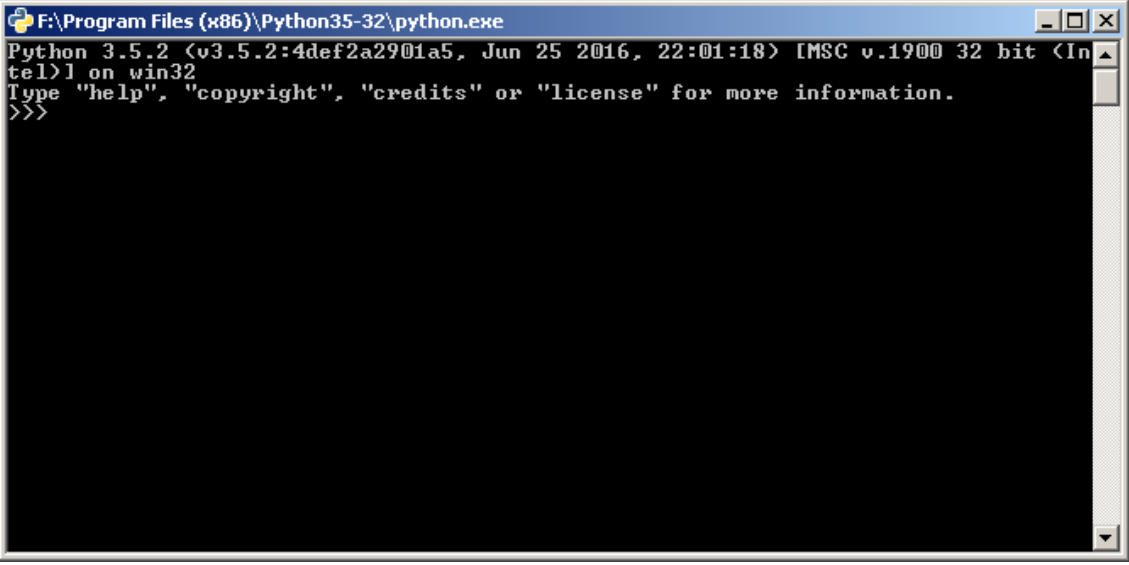
1.5 Проверка работоспособности

Теперь проверим работоспособность всего того, что мы установили.

1.5.1 Проверка интерпретатора *Python*

Для начала протестируем интерпретатор в командном режиме. Если вы работаете в *Windows*, то нажмите сочетание *Win+R* и в появившемся окне введите *python*. В *Linux* откройте окно терминала и в нем введите *python3* (или *python*).

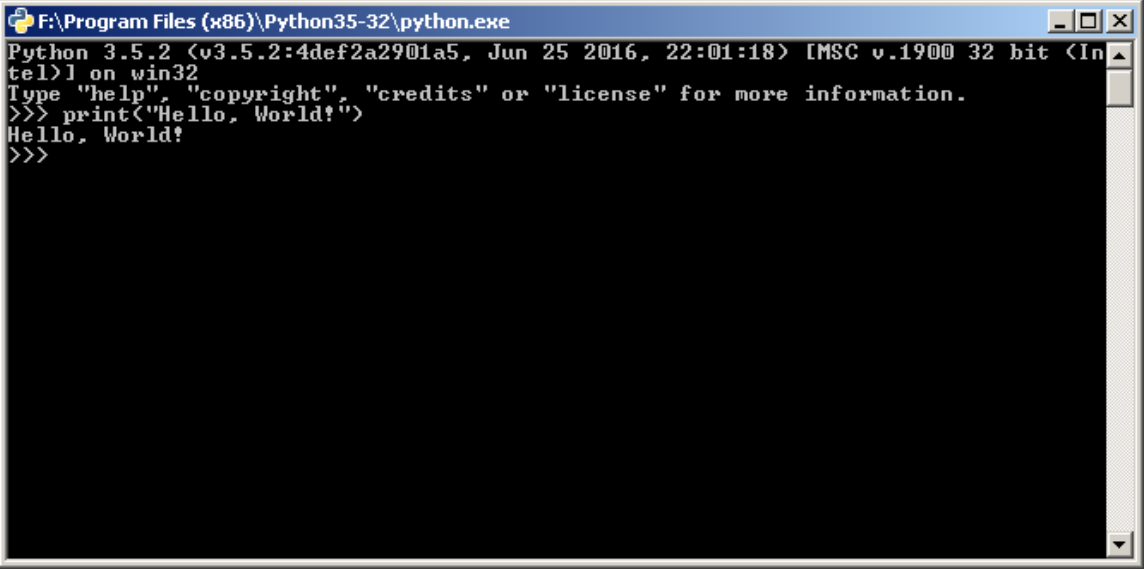
В результате *Python* запустится в командном режиме, выглядеть это будет примерно так (картинка приведена для *Windows*, в *Linux* результат будет аналогичным):



```
F:\Program Files (x86)\Python35-32\python.exe
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
tel> Type "help", "copyright", "credits" or "license" for more information.
tel>>>
```

В окне введите: *print("Hello, World!")*

Результат должен быть следующий:



```
F:\Program Files (x86)\Python35-32\python.exe
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
tel> Type "help", "copyright", "credits" or "license" for more information.
tel>>> print("Hello, World!")
Hello, World!
tel>>>
```

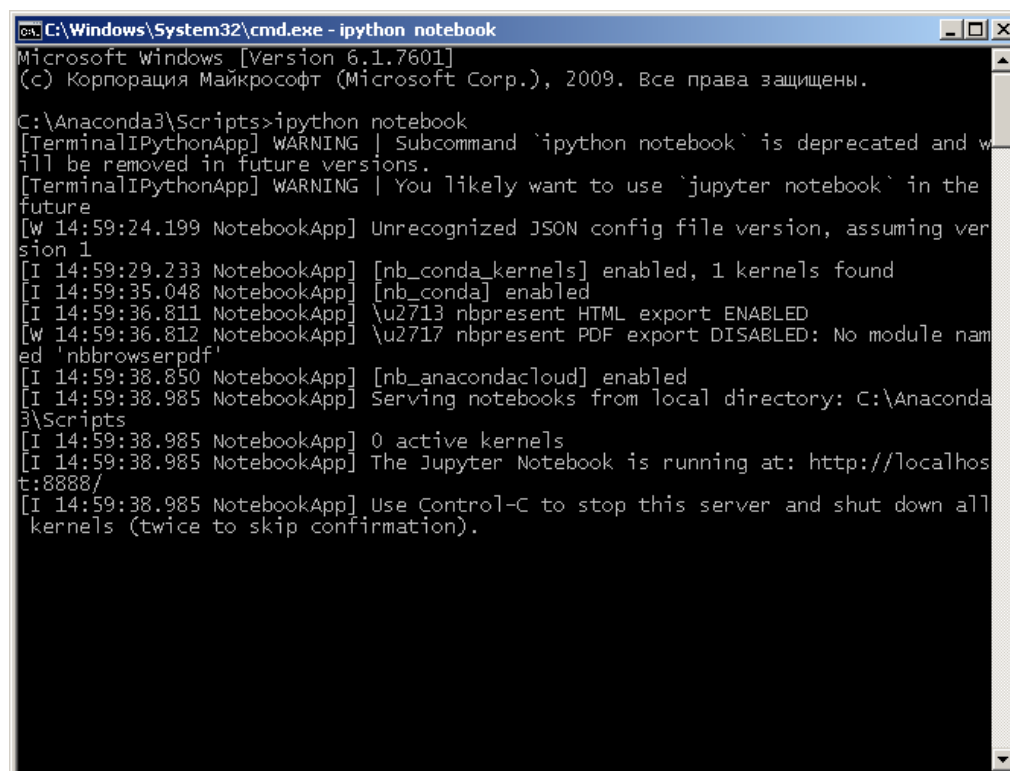
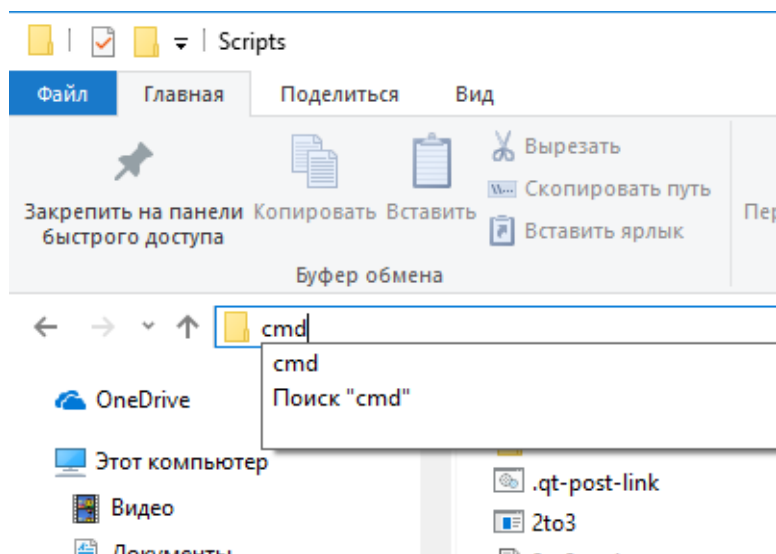
1.5.2 Проверка *Anaconda*

Здесь и далее будем считать, что пакет *Anaconda* установлен в *Windows*, в папку *C:\Anaconda3*, в *Linux*, вы его можно найти в каталоге, который выбрали при установке.

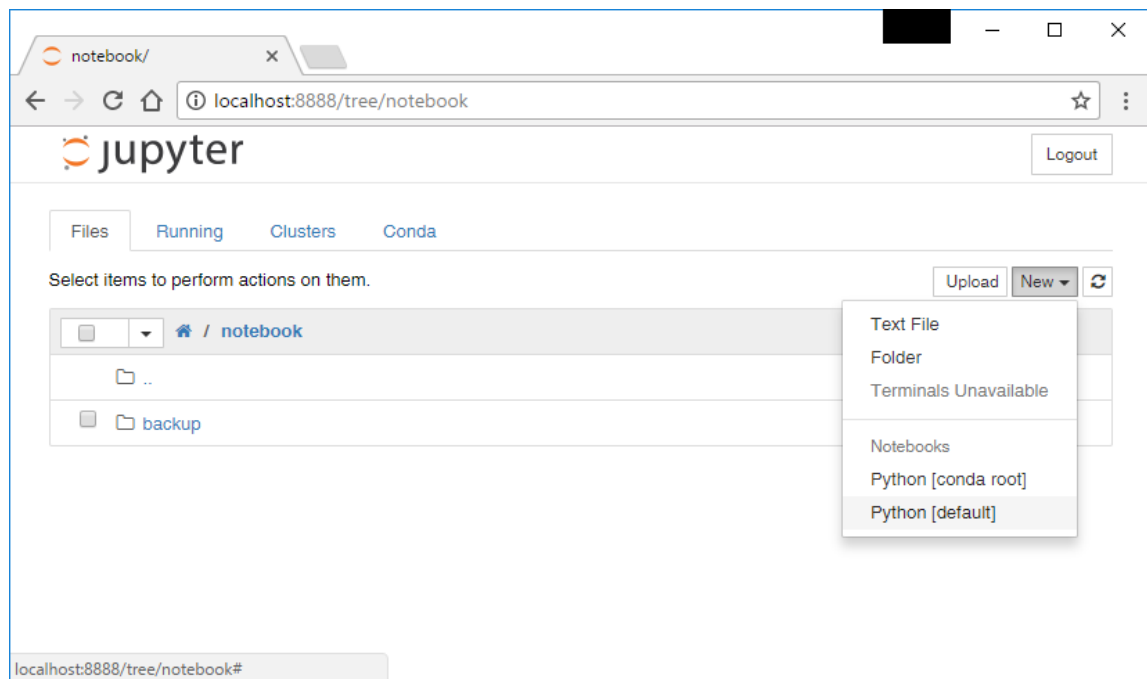
Перейдите в папку *Scripts* и введите в командной строке:

```
> ipython notebook
```

Если вы находитесь в *Windows* и открыли папку *C:\Anaconda3\Scripts* через проводник, то для запуска интерпретатора командной строки для этой папки в поле адреса введите *cmd*.



В результате запустится веб-сервер и среда разработки в браузере.

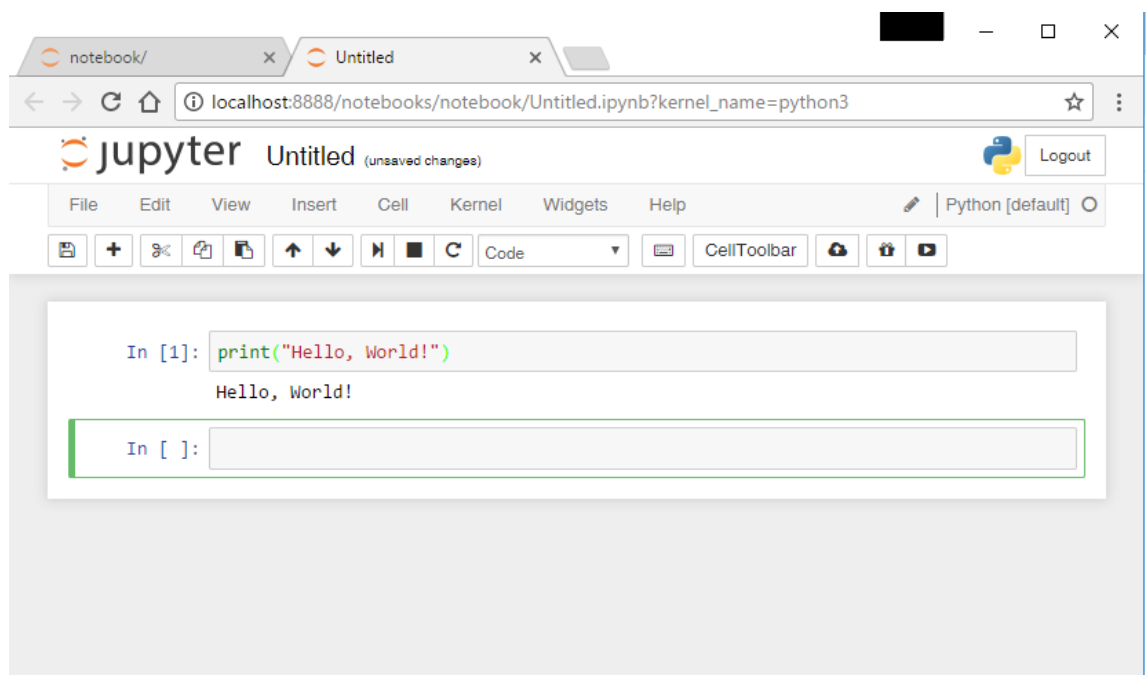


Создайте ноутбук для разработки, для этого нажмите на кнопку *New* (в правом углу окна) и в появившемся списке выберете *Python*.

В результате будет создана новая страница в браузере с ноутбуком. Введите в первой ячейке команду

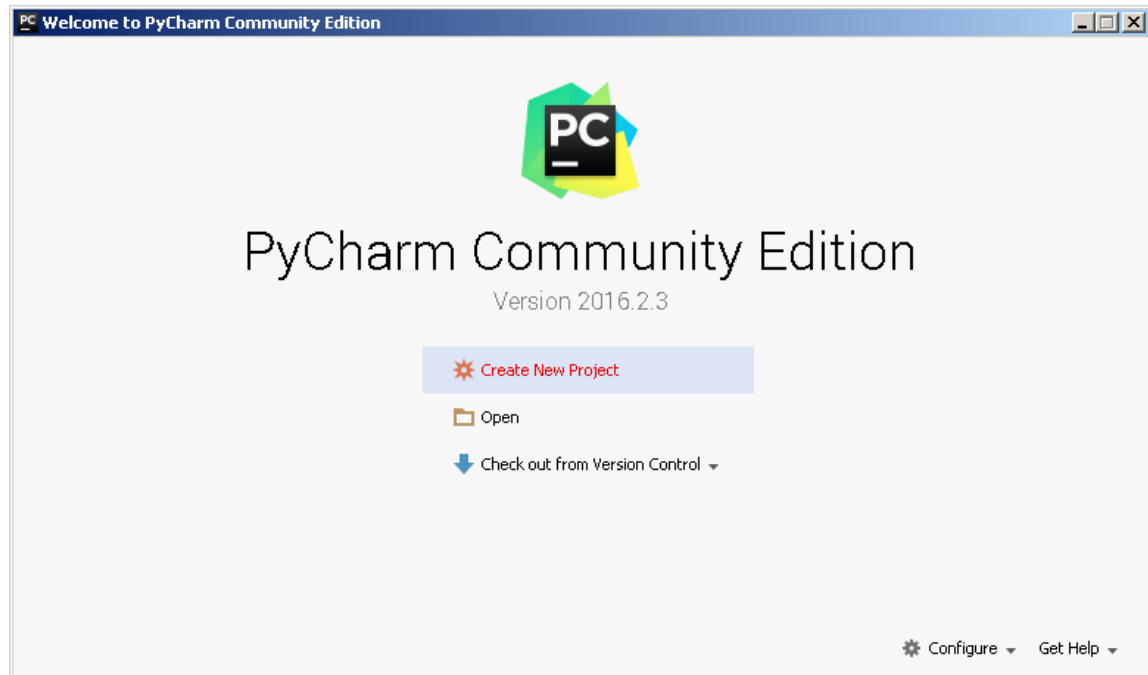
```
print("Hello, World!")
```

и нажмите *Alt+Enter* на клавиатуре. Ниже ячейки должна появиться соответствующая надпись.

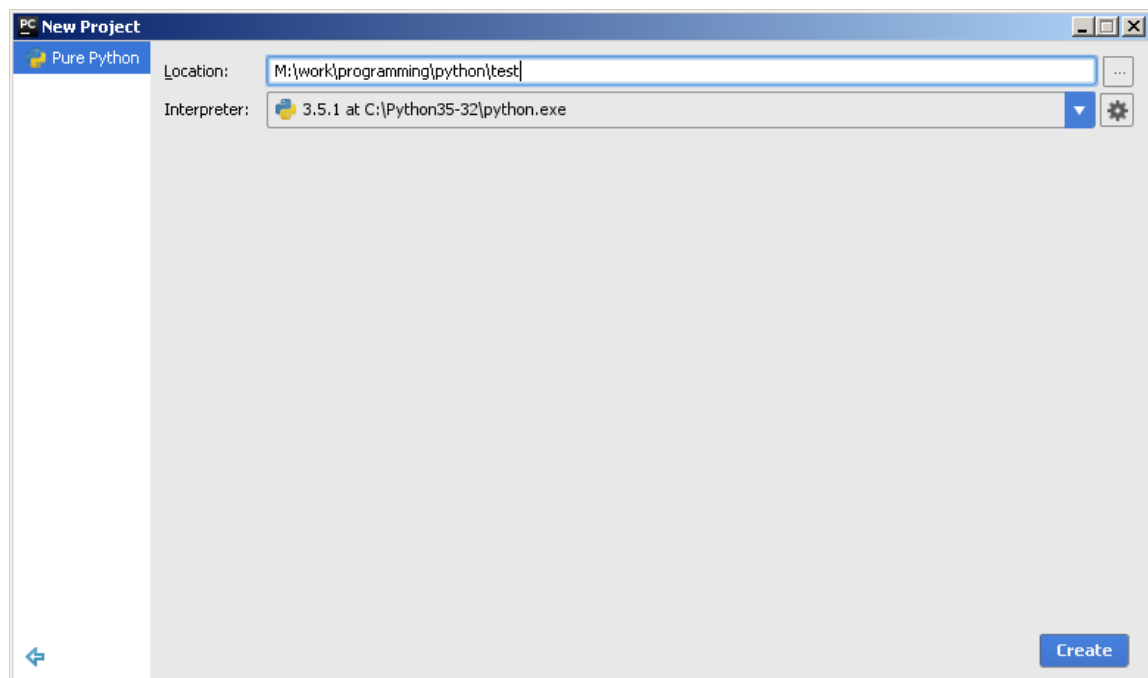


1.5.3 Проверка *PyCharm*

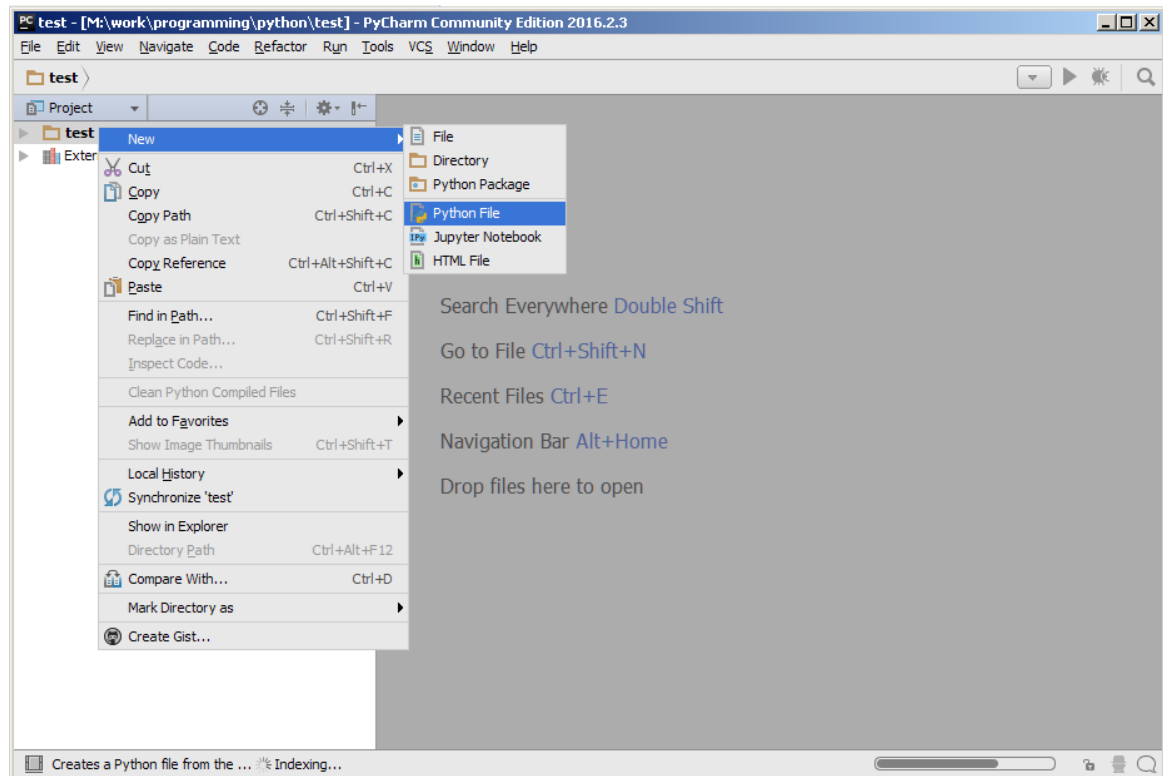
Запустите *PyCharm* и выберите *Create New Project* в появившемся окне.



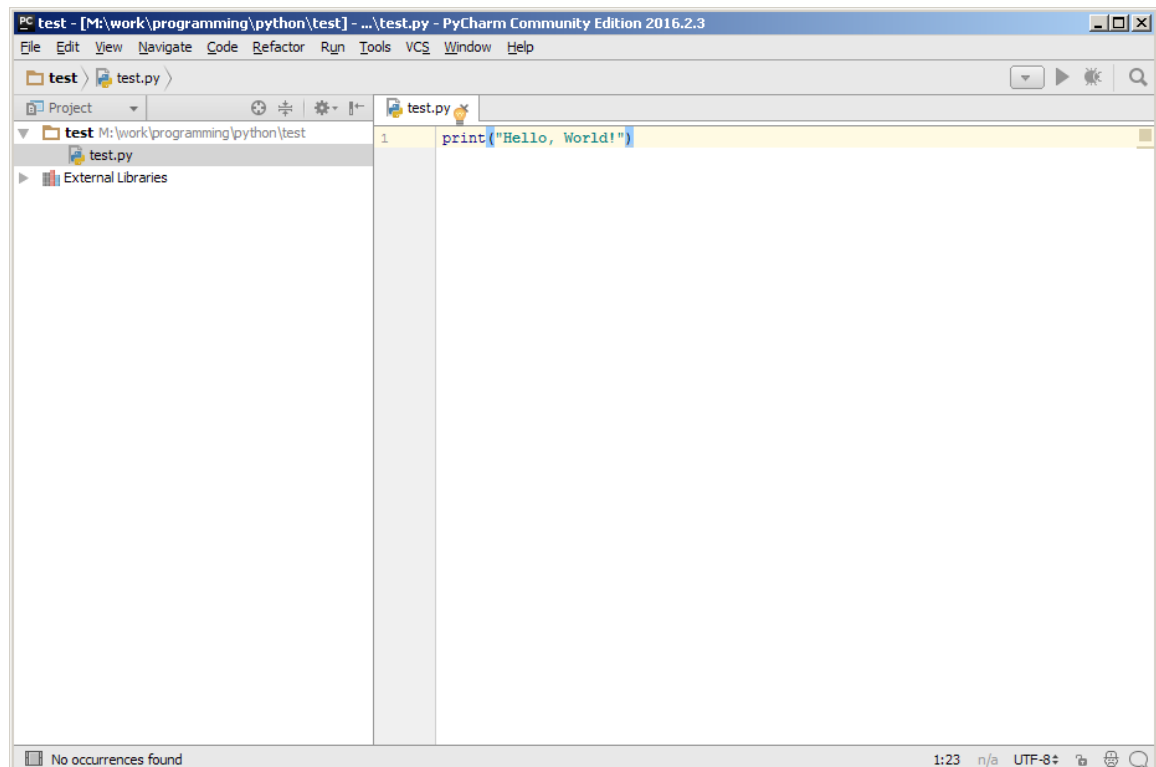
Укажите путь до проекта Python и интерпретатор, который будет использоваться для запуска и отладки.



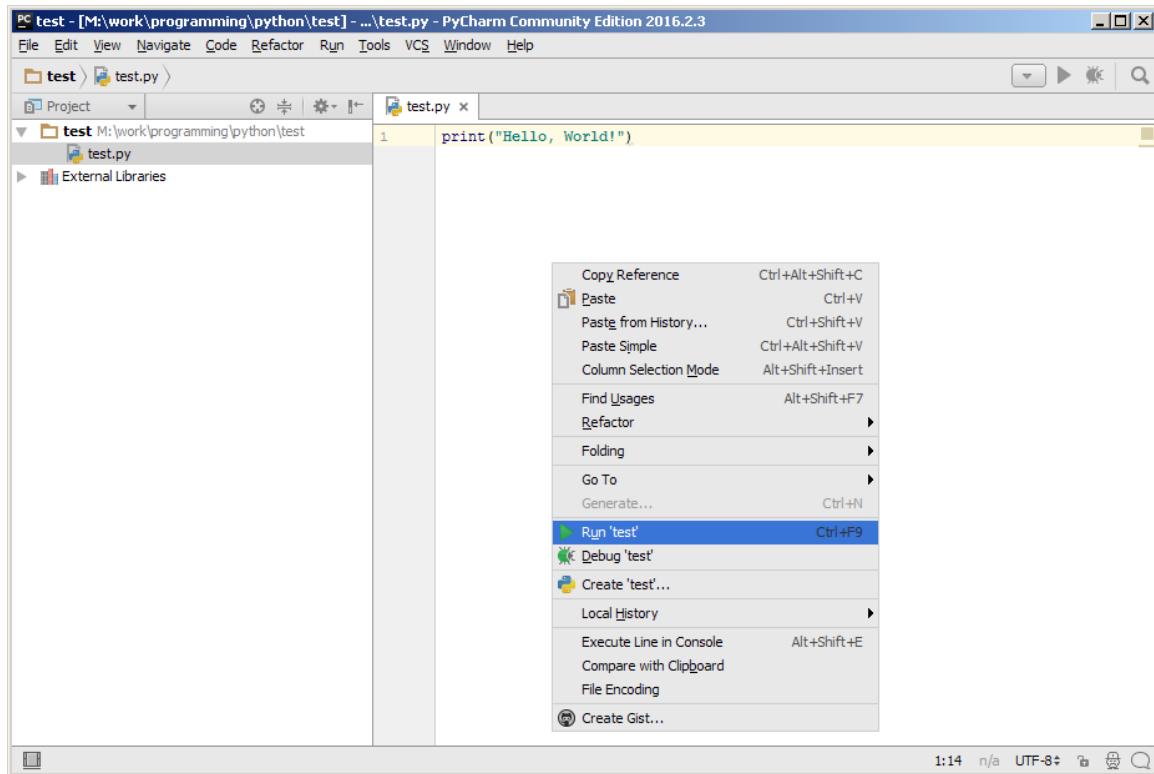
Добавьте *Python* файл в проект.



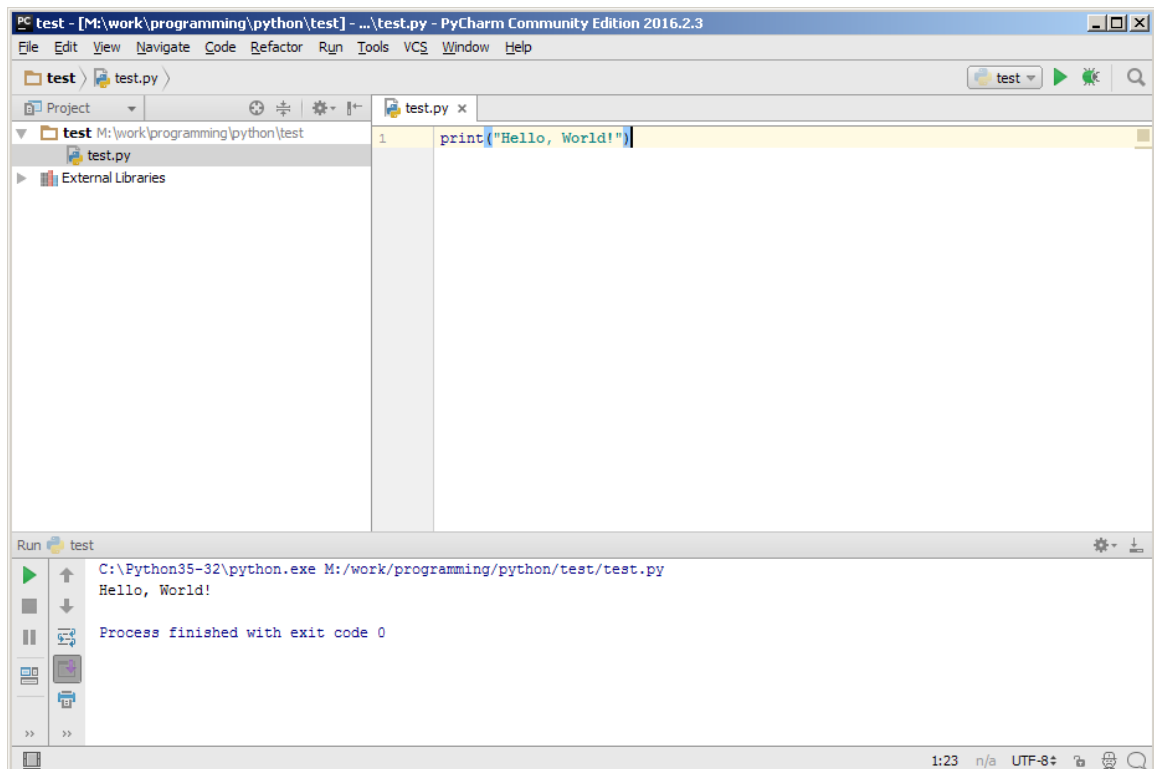
Введите код программы.



Запустите программу.



В результате должно открыться окно с выводом программы.



Урок 2. Запуск программ на *Python*

В этом уроке мы рассмотрим два основных подхода к работе с интерпретатором *Python* – это непосредственная интерпретация строк кода, вводимых с клавиатуры в интерактивном режиме и выполнение файлов с исходным кодом в пакетном режиме. Также коснемся некоторых особенностей работы с *Python* в *Linux* и *MS Windows*.

Язык *Python* – это интерпретируемый язык. Это означает, что помимо непосредственно самой программы, вам необходим специальный инструмент для её запуска. Напомню, что существуют компилируемые и интерпретируемые языки программирования. В первом случае, программа с языка высокого уровня переводится в машинный код для конкретной платформы. В дальнейшем, среди пользователей, она, как правило, распространяется в виде бинарного файла. Для запуска такой программы не нужны дополнительные программные средства (за исключением необходимых библиотек, но эти тонкости выходят за рамки нашего обсуждения). Самыми распространенными языками такого типа являются C++ и C. Программы на интерпретируемых языках, выполняются интерпретатором и распространяются в виде исходного кода.

Если вы еще не установили интерпретатор *Python*, то самое время это сделать. Подробно об этом написано в предыдущем уроке.

Python может работать в двух режимах:

- интерактивный;
- пакетный.

2.1 Интерактивный режим работы

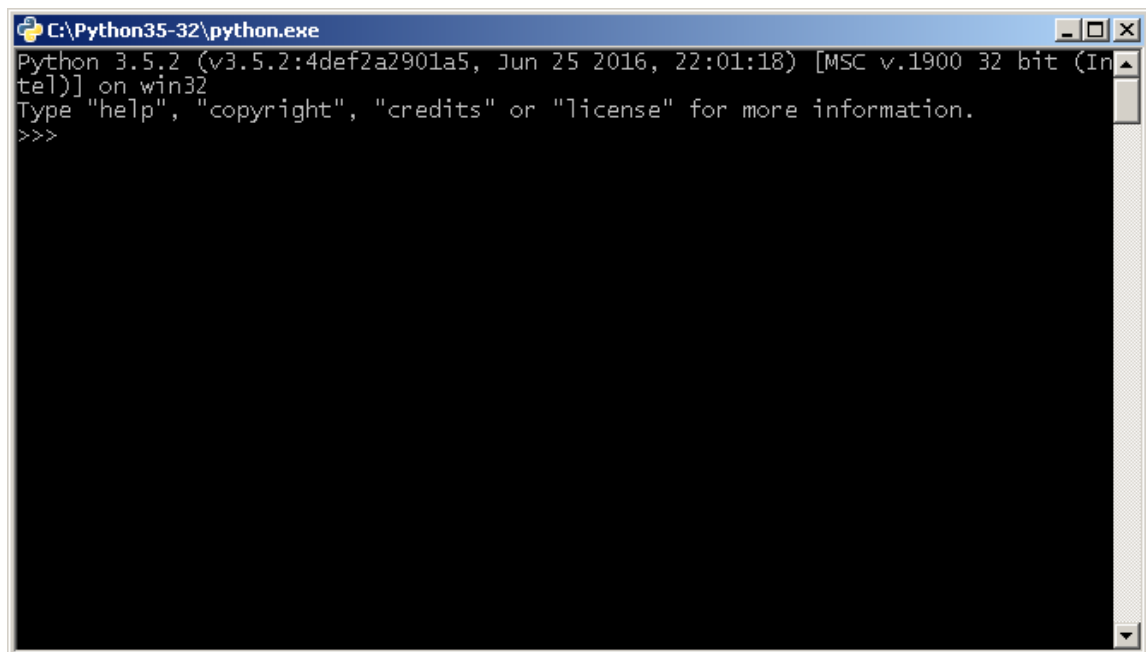
В интерактивный режим можно войти, набрав в командной строке

```
> python
```

или

```
> python3
```

В результате *Python* запустится в интерактивном режиме и будет ожидать ввод команд пользователя.



Если же у вас есть файл с исходным кодом на *Python*, и вы его хотите запустить, то для этого нужно в командной строке вызвать интерпретатор *Python* и в качестве аргумента передать ваш файл. Например, для файла с именем *test.py* процедура запуска будет выглядеть так:

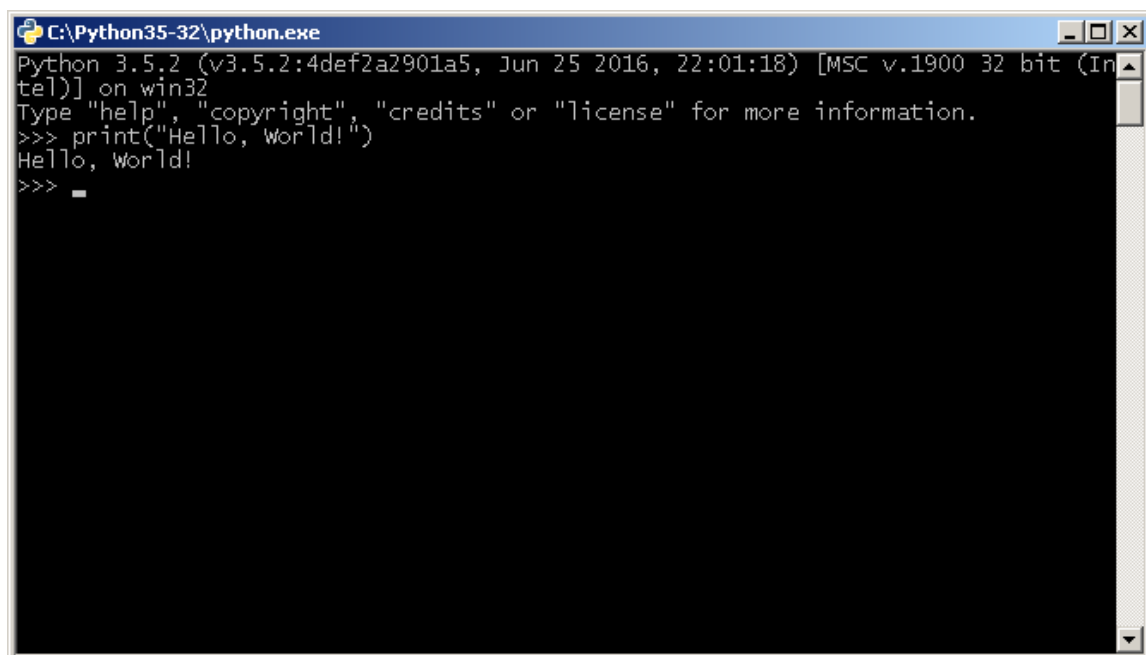
```
> python test.py
```

Откройте *Python* в интерактивном режиме и наберите в нем следующее:

```
print("Hello, World!")
```

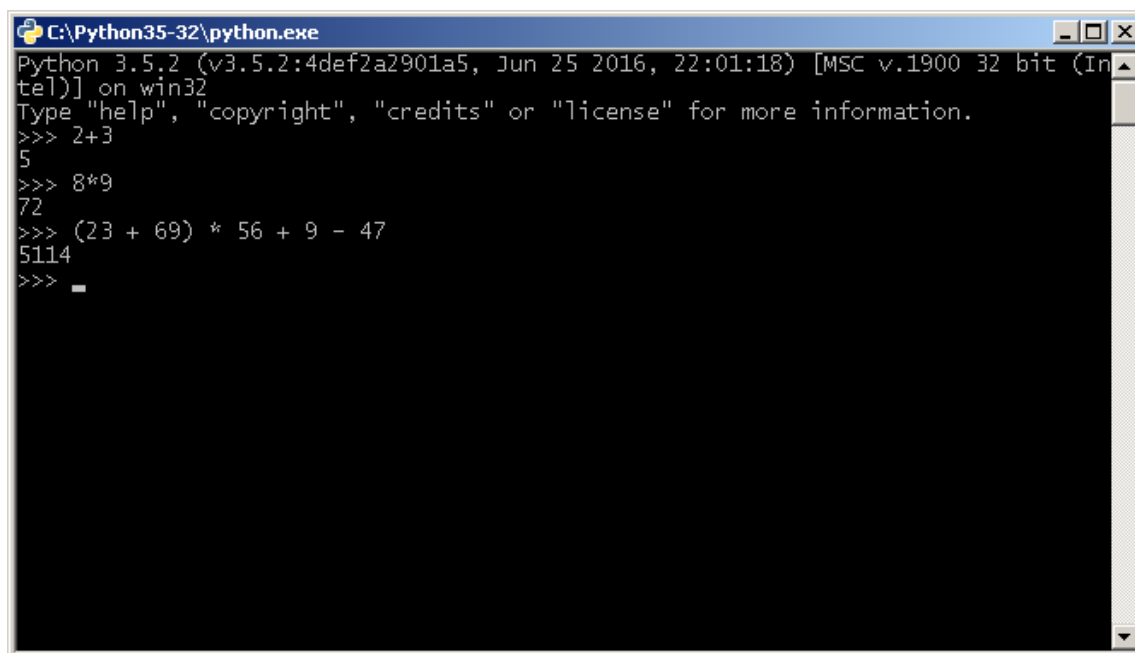
И нажмите *ENTER*.

В ответ на это интерпретатор выполнит данную строку и отобразит строкой ниже результат своей работы.



Python можно использовать как калькулятор для различных вычислений, а если дополнительно подключить необходимые математические библиотеки, то по своим возможностям он становится практически равным таким пакетам как *Matlab*, *Octave* и т.п.

Различные примеры вычислений приведены ниже. Более подробно об арифметических операциях будет рассказано в следующих уроках.



```
C:\Python35-32\python.exe
Python 3.5.2 (v3.5.2:4def2a2901a5, Jun 25 2016, 22:01:18) [MSC v.1900 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 2+3
5
>>> 8*9
72
>>> (23 + 69) * 56 + 9 - 47
5114
>>> _
```

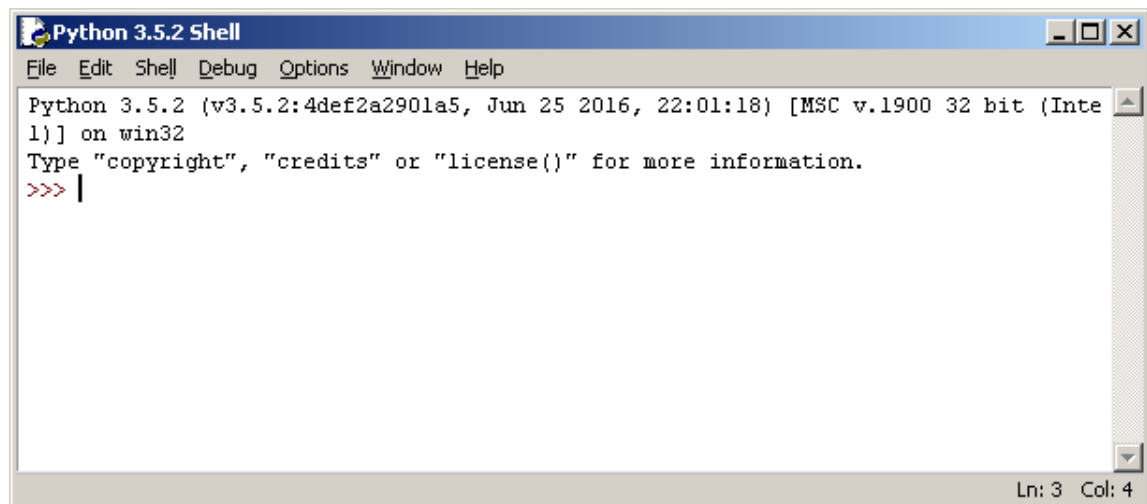
Для выхода из интерактивного режима, наберите команду

`exit()`

и нажмите *ENTER*.

В комплекте вместе с интерпретатором *Python* идет *IDLE* (интегрированная среда разработки). По своей сути она подобна интерпретатору, запущенному в интерактивном режиме с расширенным набором возможностей (подсветка синтаксиса, просмотр объектов, отладка и т.п.).

Для запуска *IDLE* в Windows необходимо перейти в папку *Python* в меню “Пуск” и найти там ярлык с именем “*IDLE (Python 3.5 XX-bit)*”.



В *Linux* оболочка *IDLE* по умолчанию отсутствует, поэтому ее предварительно нужно установить. Для этого, если у вас *Ubuntu*, введите в командной строке (для *Python 3.4*):

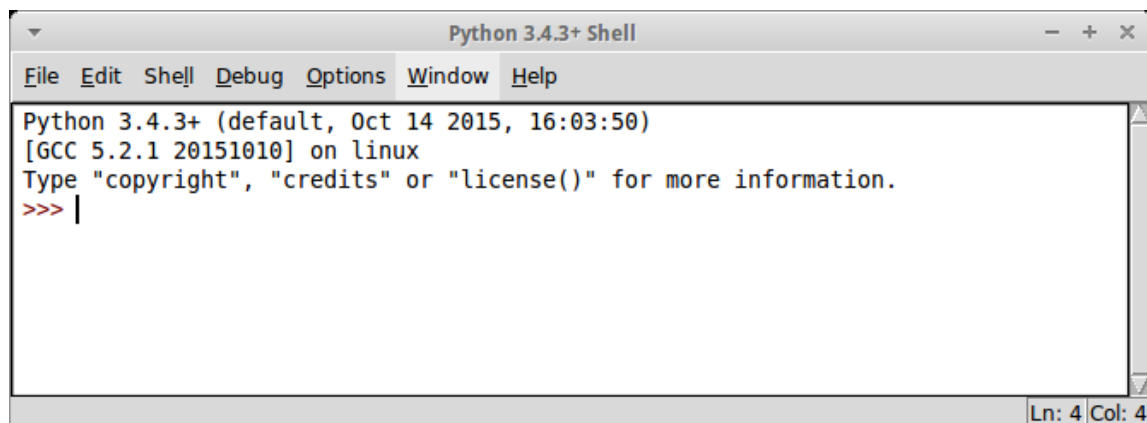
```
> sudo apt-get install idle-python3.4
```

В результате *IDLE* будет установлен на ваш компьютер.

Для запуска оболочки, введите:

```
> idle-python3.4
```

Ниже представлен внешний вид *IDLE* в ОС *Linux*.



2.2 Пакетный режим работы

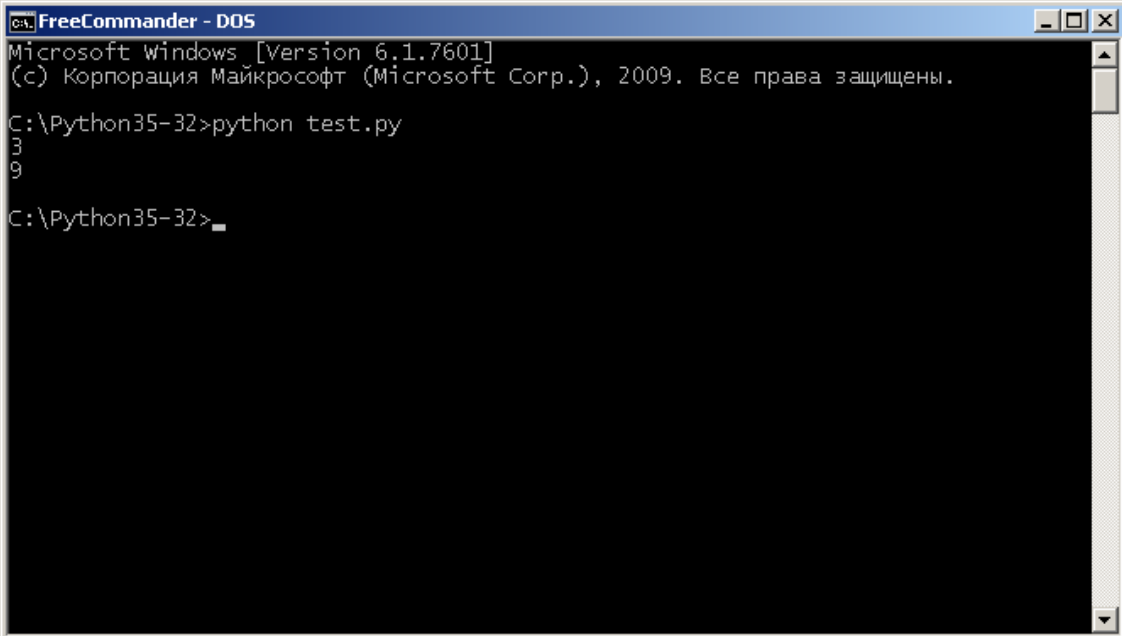
Теперь запустим *Python* в режиме интерпретации файлов с исходным кодом (пакетный режим). Создайте файл с именем *test.py*, откройте его с помощью любого текстового редактора и введите следующий код:

```
a = int(input())  
print(a**2)
```


Эта программа принимает целое число на вход и выводит его квадрат. Для запуска, наберите в командной строке

```
> python test.py
```

Пример работы программы приведен в окне ниже.



```
FreeCommander - DOS
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Python35-32>python test.py
3
9
C:\Python35-32>
```

Урок 3. Типы и модель данных

В данном уроке разберем как *Python* работает с переменными и определим, какие типы данных можно использовать в рамках этого языка. Подробно рассмотрим модель данных *Python*, а также механизмы создания и изменения значения переменных.

3.1 Кратко о типизации языков программирования

Если достаточно формально подходить к вопросу о типизации языка *Python*, то можно сказать, что он относится к языкам с неявной сильной динамической типизацией.

Неявная типизация означает, что при объявлении переменной вам не нужно указывать её тип, при явной – это делать необходимо. В качестве примера языков с явной типизацией можно привести *Java*, *C++*. Вот как будет выглядеть объявление целочисленной переменной в *Java* и *Python*.

Java:

```
int a = 1;
```

Python:

```
a = 1
```

Также языки бывают с динамической и статической типизацией. В первом случае тип переменной определяется непосредственно при выполнении программы, во втором – на этапе компиляции (о компиляции и интерпретации кратко рассказано в [уроке 2](#)). Как уже было сказано *Python* – это динамически типизированный язык, такие языки как *C*, *C#*, *Java* – статически типизированные.

Сильная типизация не позволяет производить операции в выражениях с данными различных типов, слабая – позволяет. В языках с сильной типизацией вы не можете складывать например строки и числа, нужно все приводить к одному типу. К первой группе можно отнести *Python*, *Java*, ко второй – *C* и *C++*.

3.2 Типы данных в *Python*

В *Python* типы данных можно разделить на встроенные в интерпретатор (*built-in*) и не встроенные, которые можно использовать при импортировании соответствующих модулей.

К основным встроенным типам относятся:

1. *None* (неопределенное значение переменной)
2. Логические переменные (*Boolean Type*)
3. Числа (*Numeric Type*)
 - a. *int* – целое число
 - b. *float* – число с плавающей точкой
 - c. *complex* – комплексное число
4. Списки (*Sequence Type*)
 - a. *list* – список
 - b. *tuple* – кортеж
 - c. *range* – диапазон
5. Строки (*Text Sequence Type*)
 - a. *str*
6. Бинарные списки (*Binary Sequence Types*)
 - a. *bytes* – байты
 - b. *bytearray* – массивы байт
 - c. *memoryview* – специальные объекты для доступа к внутренним данным объекта через *protocol buffer*
7. Множества (*Set Types*)
 - a. *set* – множество
 - b. *frozenset* – неизменяемое множество
8. Словари (*Mapping Types*)
 - a. *dict* – словарь

3.3 Модель данных

Рассмотрим как создаются объекты в памяти, их устройство, процесс объявления новых переменных и работу операции присваивания.

Для того, чтобы объявить и сразу инициализировать переменную необходимо написать её имя, потом поставить знак равенства и значение, с которым эта переменная будет создана.

Например строка:

```
b = 5
```

Объявляет переменную *b* и присваивает ей значение 5.

Целочисленное значение 5 в рамках языка *Python* по сути своей является объектом. Объект, в данном случае – это абстракция для представления данных, данные – это числа, списки, строки и т.п. При этом, под данными следует понимать как непосредственно сами объекты, так и отношения между ними (об этом чуть позже). Каждый объект имеет три атрибута – это идентификатор, значение и тип. Идентификатор – это уникальный признак объекта, позволяющий отличать объекты друг от друга, а значение – непосредственно информация, хранящаяся в памяти, которой управляет интерпретатор.

При инициализации переменной, на уровне интерпретатора, происходит следующее:

- создается целочисленный объект 5 (можно представить, что в этот момент создается ячейка и число 5 кладется в эту ячейку);
- данный объект имеет некоторый идентификатор, значение: 5, и тип: целое число;
- посредством оператора “=” создается ссылка между переменной *b* и целочисленным объектом 5 (переменная *b* ссылается на объект 5).

Имя переменной не должно совпадать с ключевыми словами интерпретатора *Python*. Список ключевых слов можно получить непосредственно в программе, для этого нужно подключить модуль *keyword* и воспользоваться командой *keyword.kwlist*:

```
>>> import keyword
>>> print("Python keywords: ", keyword.kwlist)
```

Проверить является ли идентификатор ключевым словом можно так:

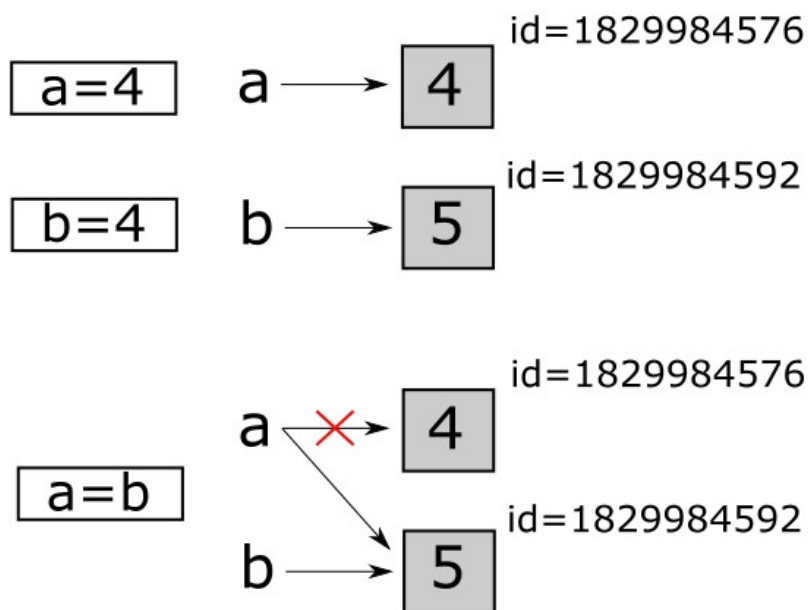
```
>>> keyword.iskeyword("try")
True
>>> keyword.iskeyword("b")
False
```

Для того, чтобы посмотреть на объект с каким идентификатором ссылается данная переменная, можно использовать функцию *id()*:

```
>>> a = 4
>>> b = 5
>>> id(a)
1829984576
```

```
>>> id(b)
1829984592
>>> a = b
>>> id(a)
1829984592
```

Как видно из примера, идентификатор – это некоторое целочисленное значение, посредством которого уникально адресуется объект. Изначально переменная *a* ссылается на объект 4 с идентификатором 1829984576, переменная *b* – на объект с *id* = 1829984592. После выполнения операции присваивания *a = b*, переменная *a* стала ссылаться на тот же объект, что и *b*.



Тип переменной можно определить с помощью функции *type()*. Пример использования приведен ниже:

```
>>> a = 10
>>> b = "hello"
>>> c = (1, 2)
>>> type(a)
<class 'int'>
>>> type(b)
<class 'str'>
>>> type(c)
<class 'tuple'>
```

3.4 Изменяемые и неизменяемые типы данных

В *Python* существуют изменяемые и неизменяемые типы.

К неизменяемым (*immutable*) типам относятся:

- целые числа (*int*);
- числа с плавающей точкой (*float*);
- комплексные числа (*complex*);
- логические переменные (*bool*);
- кортежи (*tuple*);
- строки (*str*);
- неизменяемые множества (*frozen set*).

К изменяемым (*mutable*) типам относятся

- списки (*list*);
- множества (*set*);
- словари (*dict*).

Как уже было сказано ранее, при создании переменной, вначале создается объект, который имеет уникальный идентификатор, тип и значение, после этого переменная может ссылаться на созданный объект.

Неизменяемость типа данных означает, что созданный объект больше не изменяется. Например, если мы объявим переменную $k = 15$, то будет создан объект со значением 15, типа *int* и идентификатором, который можно узнать с помощью функции *id()*:

```
>>> k = 15
>>> id(k)
1672501744
>>> type(k)
<class 'int'>
```

Объект с *id* = 1672501744 будет иметь значение 15 и изменить его уже нельзя.

Если тип данных изменяемый, то можно менять значение объекта.

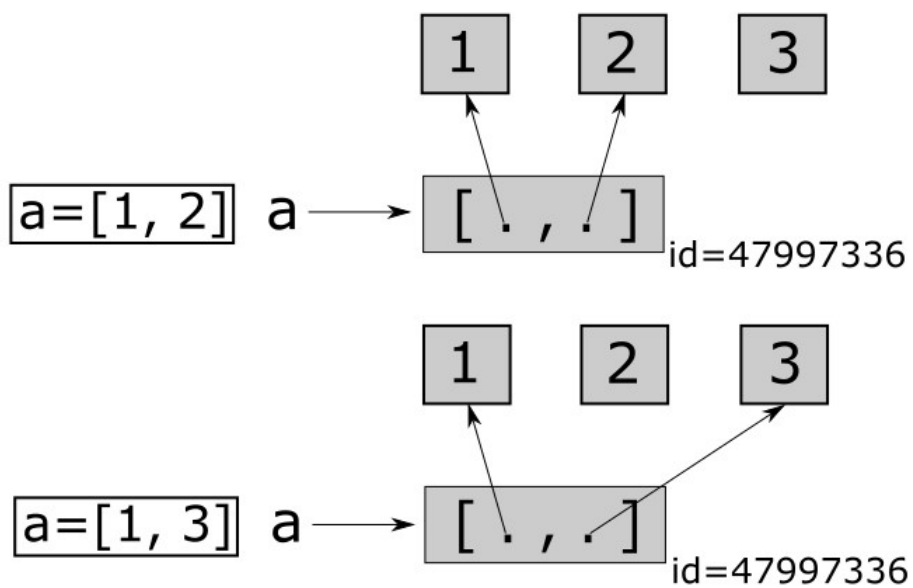
Например, создадим список [1, 2], а потом заменим второй элемент на 3:

```

>>> a = [1, 2]
>>> id(a)
47997336
>>> a[1] = 3
>>> a
[1, 3]
>>> id(a)
47997336

```

Как видно, объект на который ссылается переменная `a`, был изменен. Это можно проиллюстрировать следующим рисунком.



В рассмотренном случае, в качестве данных списка, выступают не объекты, а отношения между объектами. Т.е. в переменной `a` хранятся ссылки на объекты содержащие числа 1 и 3, а не непосредственно сами эти числа.

Урок 4. Арифметические операции

Язык *Python*, благодаря наличию огромного количества библиотек для решения разного рода вычислительных задач, является конкурентом таким пакетам как *Matlab* и *Octave*. Запущенный в интерактивном режиме, он, фактически, превращается в мощный калькулятор. В этом уроке речь пойдет об арифметических операциях, доступных в данном языке.

Как было сказано в предыдущем уроке, посвященному типам и модели данных *Python*, в этом языке существует три встроенных числовых типа данных:

- целые числа (*int*);
- вещественные числа (*float*);
- комплексные числа (*complex*).

Если в качестве операндов некоторого арифметического выражения используются только целые числа, то результат тоже будет целое число. Исключением является операция деления, результатом которой является вещественное число. При совместном использовании целочисленных и вещественных переменных, результат будет вещественным.

4.1 Арифметические операции с целыми и вещественными числами

Все эксперименты будем проводить в *Python*, запущенном в интерактивном режиме.

Сложение.

Складывать можно непосредственно сами числа...

```
>>> 3+2
```

```
5
```

либо переменные, но они должны предварительно быть проинициализированы:

```
>>> a = 3
```

```
>>> b = 2
```

```
>>> a + b
```

```
5
```


Результат операции сложения можно присвоить другой переменной...

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

либо ей же самой, в таком случае можно использовать полную или сокращенную запись, полная выглядит так:

```
>>> a = 3
>>> b = 2
>>> a = a + b
>>> print(a)
5
```

сокращенная так:

```
>>> a = 3
>>> b = 2
>>> a += b
>>> print(a)
5
```

Все перечисленные выше варианты использования операции сложения могут быть применены для всех нижеследующих операций.

Вычитание:

```
>>> 4-2
2
>>> a = 5
>>> b = 7
>>> a - b
-2
```

Умножение:

```
>>> 5 * 8
40

>>> a = 4
>>> a *= 10
>>> print(a)
40
```

Деление:

```
>>> 9 / 3
3.0

>>> a = 7
>>> b = 4
>>> a / b
1.75
```

Получение целой части от деления:

```
>>> 9 // 3
3

>>> a = 7
>>> b = 4
>>> a // b
1
```

Получение остатка от деления:

```
>>> 9 % 5
4

>>> a = 7
>>> b = 4
>>> a % b
3
```

Возведение в степень:

```
>>> 5**4
625

>>> a = 4
>>> b = 3
>>> a**b
64
```

4.2 Работа с комплексными числами

Для создания комплексного числа можно использовать функцию *complex(a, b)*, в которую, в качестве первого аргумента, передается действительная часть, в качестве второго – мнимая. Либо записать число в виде $a + bj$.

Рассмотрим несколько примеров.

Создание комплексного числа:

```
>>> z = 1 + 2j
>>> print(z)
(1+2j)
>>> x = complex(3, 2)
>>> print(x)
(3+2j)
```

Комплексные числа можно складывать, вычитать, умножать, делить и возводить в степень:

```
>>> x + z
(4+4j)
>>> x - z
(2+0j)
>>> x * z
(-1+8j)
>>> x / z
(1.4-0.8j)
>>> x ** z
(-1.1122722036363393-0.012635185355335208j)
>>> x ** 3
(-9+46j)
```

У комплексного числа можно извлечь действительную и мнимую части:

```
>>> x = 3 + 2j
>>> x.real
3.0
>>> x.imag
2.0
```

Для получения комплексно сопряженного числа необходимо использовать метод *conjugate()*:

```
>>> x.conjugate()  
(3-2j)
```

4.3 Битовые операции

В *Python* доступны битовые операции, их можно производить над целыми числами.

Побитовое И (*AND*):

```
>>> p = 9  
>>> q = 3  
>>> p & q  
1
```

Побитовое ИЛИ (*OR*):

```
>>> p | q  
11
```

Побитовое Исключающее ИЛИ (*XOR*):

```
>>> p ^ q  
10
```

Инверсия:

```
>>> ~p  
-10
```

Сдвиг вправо и влево:

```
>>> p << 1  
18  
>>> p >> 1  
4
```

4.4 Представление чисел в других системах счисления

В своей повседневной жизни мы используем десятичную систему исчисления, но при программировании, очень часто, приходится работать с шестнадцатеричной, двоичной и восьмеричной.

Представление числа в шестнадцатеричной системе:

```
>>> m = 124504
>>> hex(m)
'0x1e658'
```

Представление числа в восьмеричной системе:

```
>>> oct(m)
'0o363130'
```

Представление числа в двоичной системе:

```
>>> bin(m)
'0b11110011001011000'
```

4.5 Библиотека (модуль) *math*

В стандартную поставку *Python* входит библиотека *math*, в которой содержится большое количество часто используемых математических функций.

Для работы с данным модулем его предварительно нужно импортировать.

```
>>> import math
```

Рассмотрим наиболее часто используемые функции:

math.ceil(x)

Возвращает ближайшее целое число большее, чем *x*.

```
>>> math.ceil(3.2)
4
```

math.fabs(x)

Возвращает абсолютное значение числа.

```
>>> math.fabs(-7)
7.0
```

math.factorial(x)

Вычисляет факториал x .

```
>>> math.factorial(5)
120
```

math.floor(x)

Возвращает ближайшее целое число меньшее, чем x .

```
>>> math.floor(3.2)
3
```

math.exp(x)

Вычисляет e^x .

```
>>> math.exp(3)
20.08553692318766
```

math.log2(x)

Логарифм по основанию 2.

math.log10(x)

Логарифм по основанию 10.

math.log(x[, base])

По умолчанию вычисляет логарифм по основанию e , дополнительно можно указать основание логарифма.

```
>>> math.log2(8)
3.0
>>> math.log10(1000)
3.0
>>> math.log(5)
1.609437912434100
>>> math.log(4, 8)
0.6666666666666666
```

math.pow(x, y)

Вычисляет значение x в степени y .

```
>>> math.pow(3, 4)
```

```
81.0
```

math.sqrt(x)

Корень квадратный от x .

```
>>> math.sqrt(25)
```

```
5.0
```

Тригонометрические функции, их мы оставим без примера.

math.cos(x)

math.sin(x)

math.tan(x)

math.acos(x)

math.asin(x)

math.atan(x)

И напоследок пару констант:

math.pi

Число π .

math.e

Число e .

Помимо перечисленных, модуль *math* содержит ещё много различных функций, за более подробной информацией можете обратиться на официальный сайт (<https://docs.python.org/3/library/math.html>).

Урок 5. Условные операторы и циклы

В этом уроке рассмотрим оператор ветвления *if* и операторы цикла *while* и *for*. Основная цель – это дать общее представление об этих операторах и на простых примерах показать базовые принципы работы с ними.

5.1 Условный оператор ветвления *if*

Оператор ветвления *if* позволяет выполнить определенный набор инструкций в зависимости от некоторого условия. Возможны следующие варианты использования.

5.1.1 Конструкция *if*

Синтаксис оператора *if* выглядит так:

```
if выражение:  
    инструкция_1  
    инструкция_2  
    ...  
    инструкция_n
```

После оператора *if* записывается выражение. Если это выражение истинно, то выполняются инструкции, определяемые данным оператором. Выражение является истинным, если его результатом является число не равное нулю, непустой объект, либо логическое True. После выражения нужно поставить двоеточие ":".

ВАЖНО: блок кода, который необходимо выполнить, в случае истинности выражения, отделяется четырьмя пробелами слева! Примеры:

```
if 1:  
    print("hello 1")
```

Напечатает: *hello 1*

```
a = 3  
if a == 3:  
    print("hello 2")
```

Напечатает: *hello 2*


```
a = 3
if a > 1:
    print("hello 3")
```

Напечатает: *hello 3*

```
lst = [1, 2, 3]
if lst :
    print("hello 4")
```

Напечатает: *hello 4*

5.1.2 Конструкция *if – else*

Бывают случаи, когда необходимо предусмотреть альтернативный вариант выполнения программы. Т.е. при истинном условии нужно выполнить один набор инструкций, при ложном – другой. Для этого используется конструкция *if – else*:

```
if выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
else:
    инструкция_a
    инструкция_b
    ...
    инструкция_x
```

Примеры:

```
a = 3
if a > 2:
    print("H")
else:
    print("L")
```

Напечатает: H

```
a = 1
if a > 2:
    print("H")
else:
    print("L")
```

Напечатает: L

Условие такого вида можно записать в строчку, в таком случае оно будет представлять собой тернарное выражение:

```
a = 17
b = True if a > 10 else False
print(b)
```

В результате выполнения такого кода будет напечатано: *True*

5.1.3 Конструкция *if – elif – else*

Для реализации выбора из нескольких альтернатив можно использовать конструкцию *if – elif – else*:

```
if выражение_1:
    инструкции_(блок_1)
elif выражение_2:
    инструкции_(блок_2)
elif выражение_3:
    инструкции_(блок_3)
else:
    инструкции_(блок_4)
```

Пример:

```
a = int(input("введите число:"))
if a < 0:
    print("Neg")
elif a == 0:
    print("Zero")
else:
    print("Pos")
```

Если пользователь введет число меньше нуля, то будет напечатано “Neg”, равное нулю – “Zero”, большее нуля – “Pos”.

5.2 Оператор цикла *while*

Оператор цикла *while* выполняет указанный набор инструкций до тех пор, пока условие цикла истинно. Истинность условия определяется также как в операторе *if*. Синтаксис оператора *while* выглядит так:

```
while выражение:
    инструкция_1
    инструкция_2
    ...
    инструкция_n
```

Выполняемый набор инструкций называется телом цикла.

Пример:

```
a = 0
while a < 7:
    print("A")
    a += 1
```

Буква “A” будет выведена семь раз в столбик.

Пример бесконечного цикла:

```
a = 0
while a == 0:
    print("A")
```

5.3 Операторы *break* и *continue*

При работе с циклами используются операторы *break* и *continue*. Оператор *break* предназначен для досрочного прерывания работы цикла *while*:

```
a = 0
while a >= 0:
    if a == 7:
        break
    a += 1
    print("A")
```

В приведенном выше коде, выход из цикла произойдет при достижении переменной *a* значения 7. Если бы не было этого условия, то цикл выполнялся бы бесконечно.

Оператор *continue* запускает цикл заново, при этом код, расположенный после данного оператора, не выполняется.

Пример:

```
a = -1
while a < 10:
    a += 1
    if a >= 7:
        continue
    print("A")
```

При запуске данного кода символ “A” будет напечатан 7 раз, несмотря на то, что всего будет выполнено 11 проходов цикла.

5.4 Оператор цикла *for*

Оператор *for* выполняет указанный набор инструкций заданное количество раз, которое определяется количеством элементов в наборе.

Пример:

```
for i in range(5):
    print("Hello")
```

В результате “Hello” будет выведено пять раз.

Внутри тела цикла можно использовать операторы *break* и *continue*, принцип работы их точно такой же как и в операторе *while*.

Если у вас есть заданный список, и вы хотите выполнить над каждым элементом определенную операцию (возвести в квадрат и распечатать получившееся число), то с помощью *for* такая задача решается так:

```
lst = [1, 3, 5, 7, 9]
for i in lst:
    print(i**2)
```

Также можно пройти по всем буквам в строке:

```
word_str = "Hello, world!"  
for l in word_str:  
    print(l)
```

Строка *"Hello, world!"* будет напечатана в столбик.

На этом закончим краткий обзор операторов ветвления и цикла.

Урок 6. Работа с *IPython* и *Jupyter Notebook*

IPython представляет собой мощный инструмент для работы с языком *Python*. Базовые компоненты *IPython* – это интерактивная оболочка для с широким набором возможностей и ядро для *Jupyter*. *Jupyter notebook* является графической веб-оболочкой для *IPython*, которая расширяет идею консольного подхода к интерактивным вычислениям.

Основные отличительные особенности данной платформы – это комплексная интроспекция объектов, сохранение истории ввода на протяжении всех сеансов, кэширование выходных результатов, расширяемая система “магических” команд, логирование сессии, дополнительный командный синтаксис, подсветка кода, доступ к системной оболочке, стыковка с *pdb* отладчиком и *Python* профайлером.

IPython позволяет подключаться множеству клиентов к одному вычислительному ядру и, благодаря своей архитектуре, может работать в параллельном кластере.

В *Jupyter notebook* вы можете разрабатывать, документировать и выполнять приложения на языке *Python*, он состоит из двух компонентов: веб-приложение, запускаемое в браузере, и ноутбуки – файлы, в которых можно работать с исходным кодом программы, запускать его, вводить и выводить данные и т.п.

Веб приложение позволяет:

- редактировать *Python* код в браузере, с подсветкой синтаксиса, автоотступами и автодополнением;
- запускать код в браузере;
- отображать результаты вычислений с медиа представлением (схемы, графики);
- работать с языком разметки *Markdown* и *LaTeX*.

Ноутбуки – это файлы, в которых сохраняются исходный код, входные и выходные данные, полученные в рамках сессии. Фактически, он является записью вашей работы, но при этом позволяет заново выполнить код, присутствующий на нем. Ноутбуки можно экспортировать в форматы *PDF*, *HTML*.

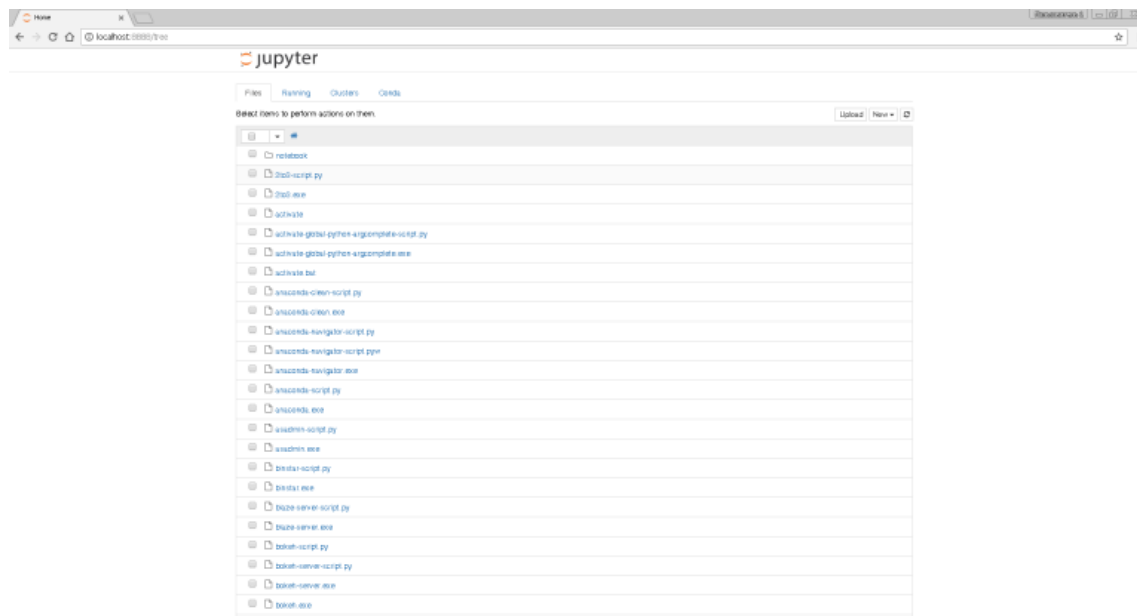
6.1 Установка и запуск

Jupyter Notebook входит в состав *Anaconda*. Описание процесса установки можно найти в первом уроке. Для запуска *Jupyter Notebook* перейдите в папку

Scripts (она находится внутри каталога, в котором установлена *Anaconda*) и в командной строке наберите:

```
> ipython notebook
```

В результате будет запущена оболочка в браузере.



6.2 Примеры работы

Будем следовать правилу: лучше один раз увидеть! Рассмотрим несколько примеров, выполнив которые, вы сразу поймете принцип работы с *Jupyter notebook*.

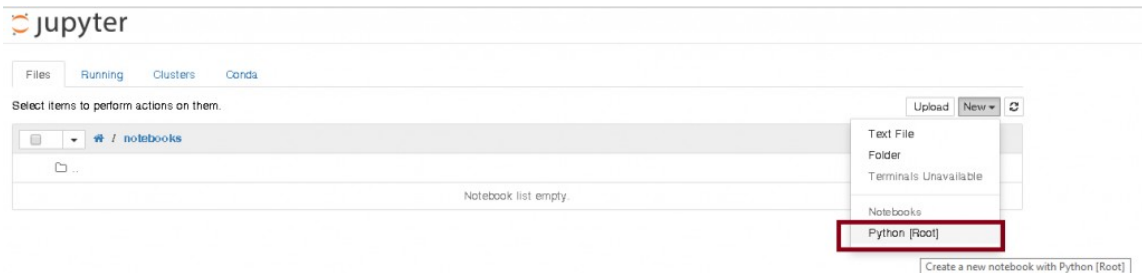
Запустите *Jupyter notebook* и создайте папку для наших примеров, для этого нажмите на *New* в правой части экрана и выберите в выпадающем списке *Folder*.



По умолчанию папке присваивается имя *“Untitled folder”*, переименуем ее в *“notebooks”*: поставьте галочку напротив имени папки и нажмите на кнопку *“Rename”*.



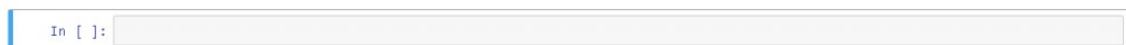
Зайдите в эту папку и создайте в ней ноутбук, воспользовавшись той же кнопкой *New*, только на этот раз нужно выбрать “*Python [Root]*”.



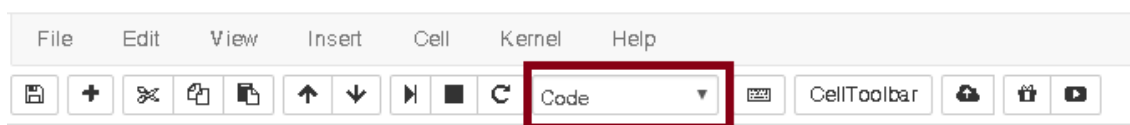
В результате будет создан ноутбук.



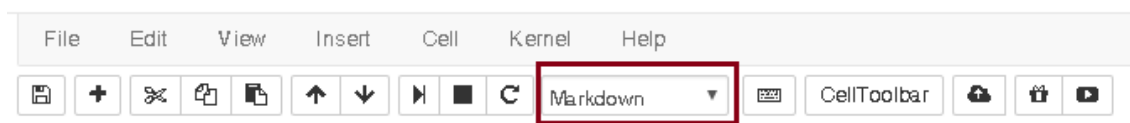
Код на языке *Python* или текст в нотации *Markdown* нужно вводить в ячейки:



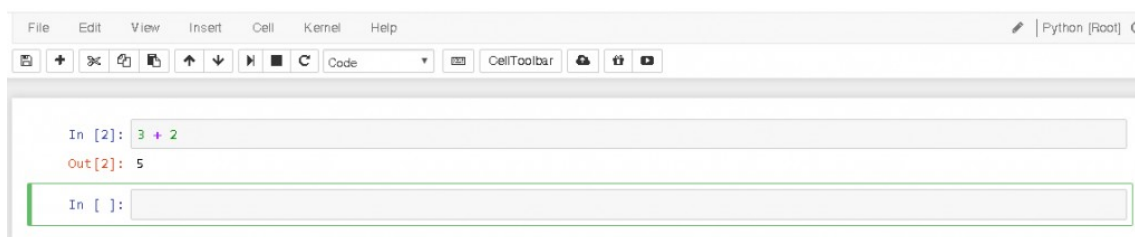
Если это код *Python*, то на панели инструментов нужно выставить свойство “*Code*”.



Если это *Markdown* текст – выставить “*Markdown*”.



Для начал решим простую арифметическую задачу: выставите свойство “Code”, введите в ячейке “2 + 3” без кавычек и нажмите *Ctrl+Enter* или *Shift+Enter*, в первом случае введенный вами код будет выполнен интерпретатором *Python*, во втором – будет выполнен код и создана новая ячейка, которая расположится уровнем ниже так, как показано на рисунке.



Если у вас получилось это сделать, выполните еще несколько примеров.

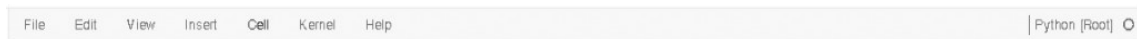


6.3 Основные элементы интерфейса *Jupyter notebook*

У каждого ноутбука есть имя, оно отображается в верхней части экрана. Для изменения имени нажмите на его текущее имя и введите новое.



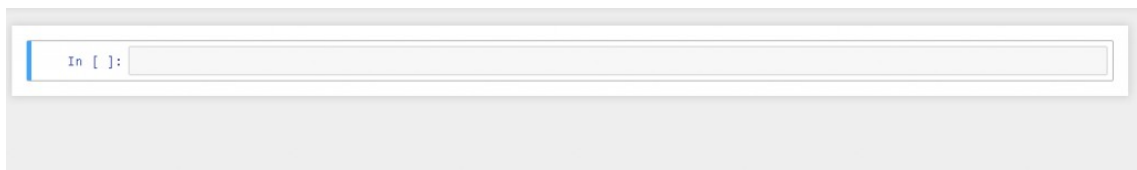
Из элементов интерфейса можно выделить, панель меню:



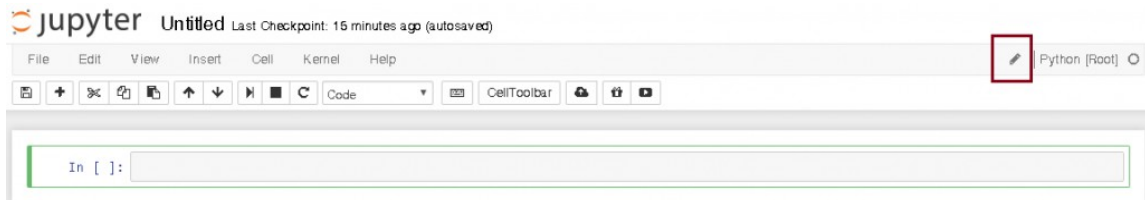
панель инструментов:



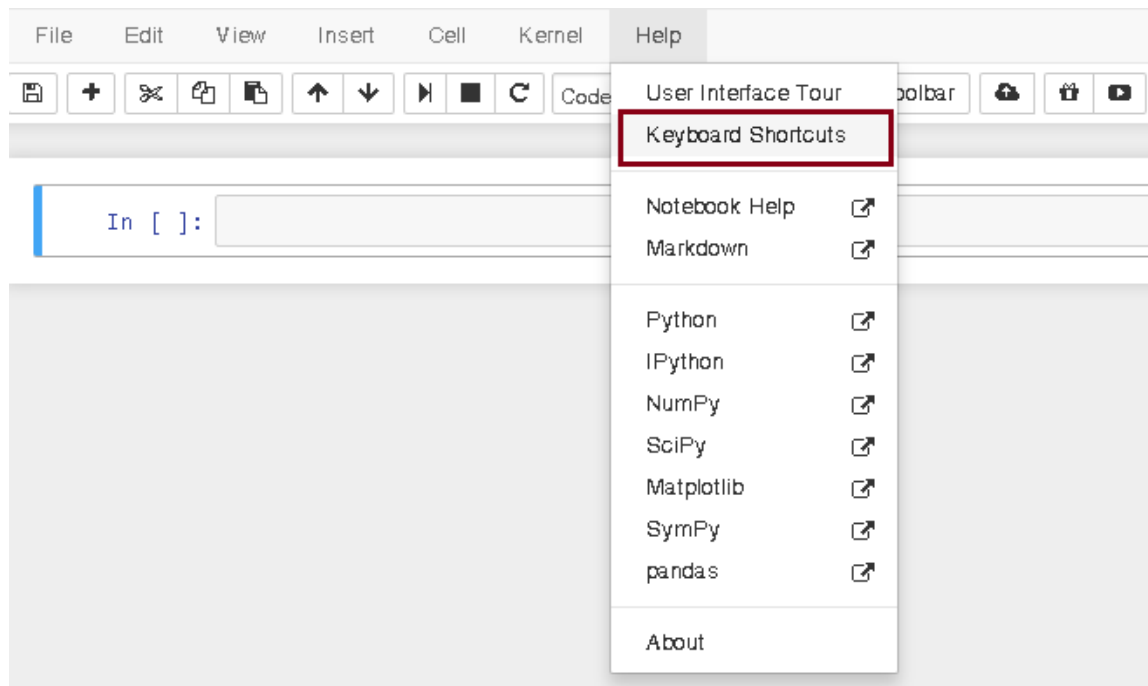
и рабочее поле с ячейками:



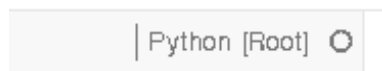
Ноутбук может находиться в одном из двух режимов – это режим правки (*Edit mode*) и командный режим (*Command mode*). Текущий режим отображается на панели меню в правой части, в режиме правки появляется изображение карандаша, отсутствие этой иконки значит, что ноутбук находится в командном режиме.



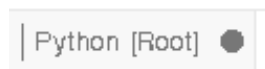
Для открытия справки по сочетаниям клавиш нажмите “*Help->Keyboard Shortcuts*”



В самой правой части панели меню находится индикатор загруженности ядра *Python*. Если ядро находится в режиме ожидания, то индикатор представляет собой окружность.



Если оно выполняет какую-то задачу, то изображение измениться на закрашенный круг.



6.4 Запуск и прерывание выполнения кода

Если ваша программа зависла, то можно прервать ее выполнение выбрав на панели меню пункт *Kernel -> Interrupt*.

Для добавления новой ячейки используйте *Insert->Insert Cell Above* и *Insert->Insert Cell Below*.

Для запуска ячейки используете команды из меню *Cell*, либо следующие сочетания клавиш:

Ctrl+Enter – выполнить содержимое ячейки.

Shift+Enter – выполнить содержимое ячейки и перейти на ячейку ниже.

Alt+Enter – выполнить содержимое ячейки и вставить новую ячейку ниже.

6.5 Как сделать ноутбук доступным для других людей?

Существует несколько способов поделиться своим ноутбуком с другими людьми, причем так, чтобы им было удобно с ним работать:

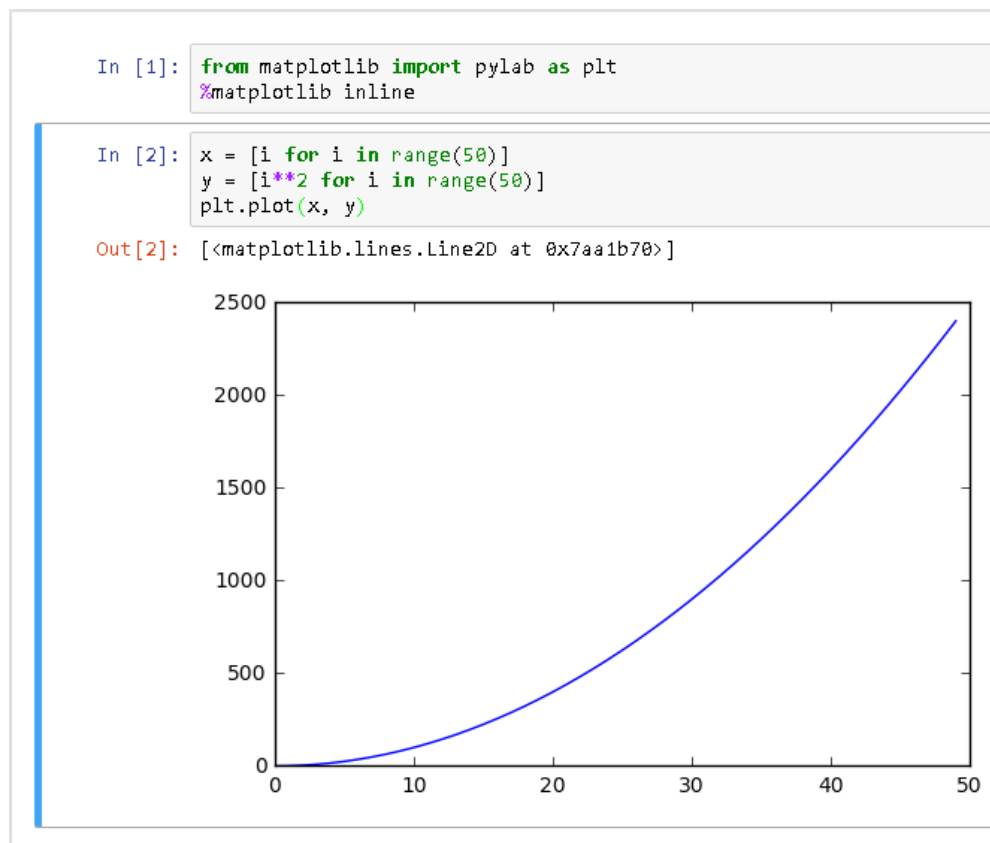
- передать непосредственно файл ноутбука, имеющий расширение “*.ipynb*”, при этом открыть его можно только с помощью *Jupyter Notebook*;
- сконвертировать ноутбук в *html*;
- использовать <https://gist.github.com/>;
- использовать <http://nbviewer.jupyter.org/>.

6.6 Вывод изображений в ноутбуке

Печать изображений может пригодиться в том случае, если вы используете библиотеку *matplotlib* для построения графиков. По умолчанию, графики не выводятся в рабочее поле ноутбука. Для того, чтобы графики отображались, необходимо ввести и выполнить следующую команду:

```
%matplotlib inline
```

Пример вывода графика представлен на рисунке ниже.



6.7 Магия

Важной частью функционала *Jupyter Notebook* является поддержка магии. Под магией в *IPython* понимаются дополнительные команды, выполняемые в рамках оболочки, которые облегчают процесс разработки и расширяют ваши возможности.

Список доступных магических команд можно получить с помощью команды **`%lsmagic`**

```
In [1]: %lsmagic
Out[1]: Available line magics:
%alias %alias_magic %autocall %automagic %autosave %bookmark %cd %clear %cls %colors %config %connect_info %copy %ddir %debug %dhist %dirs %doctest_mode %echo %ed %edit %env %gui %hist %history %killbgscripts %ldir %les %load %load_ext %loadpy %logoff %logon %logstart %logstate %logstop %ls %lsmagic %macro %magic %matplotlib %mkdir %more %notebook %page %pastebin %pdb %pdef %pdoc %pfile %pinfo %pinfo2 %ppod %pprint %precision %profile %prun %psource %pushd %pwd %pycat %pylab %qtconsole %quickref %recall %rehashx %reload_ext %ren %rep %rerun %reset %reset_selective %rmdir %run %save %sc %set_env %store %sx %system %tb %time %timeit %unalias %unload_ext %who %who_ls %whos %xdel %xmode

Available cell magics:
%%! %%HTML %%SVG %%bash %%capture %%cmd %%debug %%file %%html %%javascript %%js %%latex %%perl %%prun %%pypy %%python %%python2 %%python3 %%ruby %%script %%sh %%svg %%sx %%system %%time %%timeit %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Для работы с переменными окружения используется команда **`%env`**.

```
In [3]: %env TEST = 5
env: TEST=5
```

Запуск *Python* кода из “*.py*” файлов, а также из других ноутбуков – файлов с расширением “*.ipynb*”, осуществляется с помощью команды **`%run`**.

```
In [5]: %run ./test.py

Hello
Hello
Hello
Hello
Hello
```

Для измерения времени работы кода используйте **`%%time`** и **`%timeit`**.

`%%time` позволяет получить информацию о времени работы кода в рамках одной ячейки.

```
In [2]: %%time
import time
for i in range(50):
    time.sleep(0.1)

Wall time: 5.45 s
```

`%timeit` запускает переданный ей код 100000 раз (по умолчанию) и выводит информацию среднем значении трех наиболее быстрых прогонах.

```
In [1]: %timeit x = [(i**10) for i in range(10)]

100000 loops, best of 3: 5.75 µs per loop
```

Урок 7. Работа со списками (*list*)

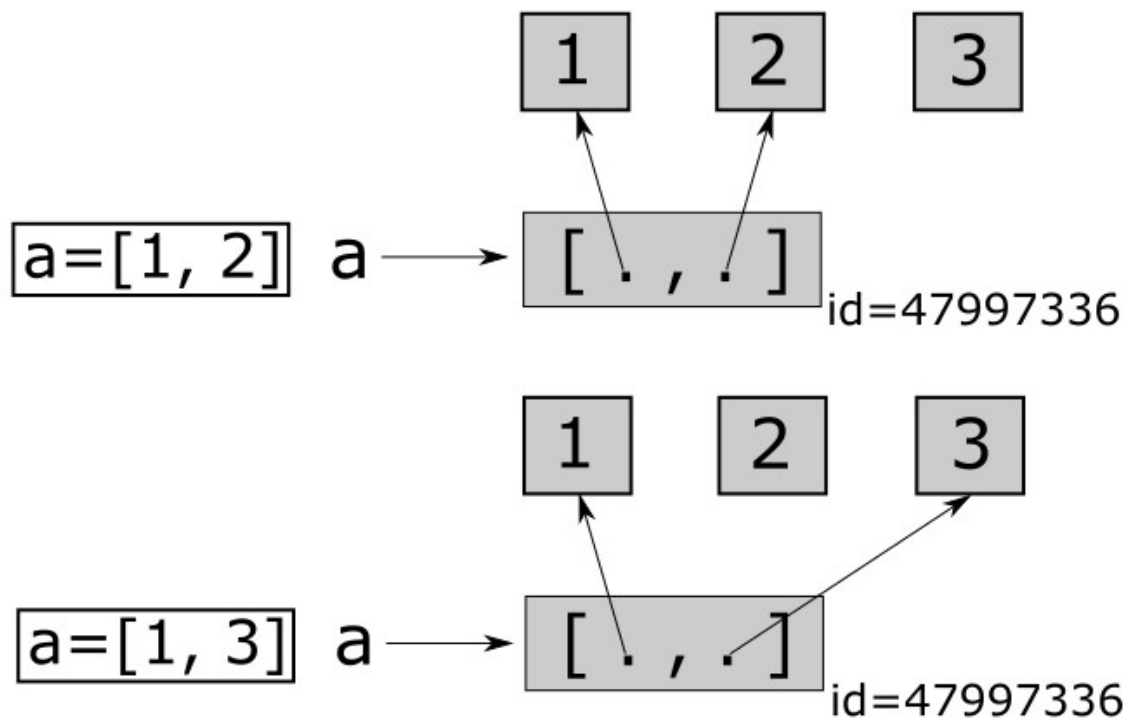
Одна из ключевых особенностей *Python*, благодаря которой он является таким популярным – это простота. Особенно подкупает простота работы с различными структурами данных – списками, кортежами, словарями и множествами. Данный урок, посвящен спискам.

7.1 Что такое список (*list*) в *Python*?

Список (*list*) – это структура данных для хранения объектов различных типов. Если вы использовали другие языки программирования, то вам должно быть знакомо понятие массива. Так вот, список очень похож на массив, только, как было уже сказано выше, в нем можно хранить объекты различных типов. Размер списка не статичен, его можно изменять. Список по своей природе является изменяемым типом данных. Про типы данных можно подробно прочитать в третьем уроке. Переменная, определяемая как список, содержит ссылку на структуру в памяти, которая в свою очередь хранит ссылки на какие-либо другие объекты или структуры.

7.2 Как списки хранятся в памяти?

Как уже было сказано выше, список является изменяемым типом данных. При его создании, в памяти резервируется область, которую можно условно назвать некоторым “контейнером”, в котором хранятся ссылки на другие элементы данных в памяти. В отличие от таких типов данных как число или строка, содержимое “контейнера” списка можно менять. Для того, чтобы лучше визуально представлять себе этот процесс взгляните на картинку ниже. Изначально был создан список содержащий ссылки на объекты 1 и 2, после операции `a[1] = 3`, вторая ссылка в списке стала указывать на объект 3.



Более подробно эти вопросы обсуждались в [уроке 3 \(Типы и модель данных\)](#).

7.3 Создание, изменение, удаление списков и работа с его элементами

Создать список можно одним из следующих способов:

```
>>> a = []  
>>> type(a)  
<class 'list'>  
>>> b = list()  
>>> type(b)  
<class 'list'>
```

Также можно создать список с заранее заданным набором данных:

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


Если у вас уже есть список и вы хотите создать его копию, то можно воспользоваться следующим способом:

```
>>> a = [1, 3, 5, 7]
>>> b = a[:]
>>> print(a)
[1, 3, 5, 7]
>>> print(b)
[1, 3, 5, 7]
```

или сделать это так:

```
>>> a = [1, 3, 5, 7]
>>> b = list(a)
>>> print(a)
[1, 3, 5, 7]
>>> print(b)
[1, 3, 5, 7]
```

В случае, если вы выполните простое присвоение списков друг другу, то переменной *b* будет присвоена ссылка на тот же элемент данных в памяти, на который ссылается *a*, а не копия списка *a*. Т.е. если вы будете изменять список *a*, то и *b* тоже будет меняться:

```
>>> a = [1, 3, 5, 7]
>>> b = a
>>> print(a)
[1, 3, 5, 7]
>>> print(b)
[1, 3, 5, 7]
>>> a[1] = 10
>>> print(a)
[1, 10, 5, 7]
>>> print(b)
[1, 10, 5, 7]
```

Добавление элемента в список осуществляется с помощью метода *append()*:

```
>>> a = []
>>> a.append(3)
>>> a.append("hello")
>>> print(a)
[3, 'hello']
```

Для удаления элемента из списка, в случае, если вы знаете его значение, используйте метод *remove(x)*, при этом будет удалена первая ссылка на данный элемент:

```
>>> b = [2, 3, 5]
>>> print(b)
[2, 3, 5]
>>> b.remove(3)
>>> print(b)
[2, 5]
```

Если необходимо удалить элемент по его индексу, воспользуйтесь командой *del имя_списка[индекс]*:

```
>>> c = [3, 5, 1, 9, 6]
>>> print(c)
[3, 5, 1, 9, 6]
>>> del c[2]
>>> print(c)
[3, 5, 9, 6]
```

Изменить значение элемента списка, зная его индекс, можно обратившись напрямую к нему:

```
>>> d = [2, 4, 9]
>>> print(d)
[2, 4, 9]
>>> d[1] = 17
>>> print(d)
[2, 17, 9]
```

Очистить список можно просто заново его проинициализировав, так как будто вы его вновь создаете. Для получения доступа к элементу списка укажите индекс этого элемента в квадратных скобках:

```
>>> a = [3, 5, 7, 10, 3, 2, 6, 0]
>>> a[2]
7
```

Можно использовать отрицательные индексы, в таком случае счет будет идти с конца, например для доступа к последнему элементу списка можно использовать вот такую команду:

```
>>> a[-1]
0
```

Для получения из списка некоторого подсписка в определенном диапазоне индексов, укажите начальный и конечный индекс в квадратных скобках, разделив их двоеточием:

```
>>> a[1:4]
[5, 7, 10]
```

7.4 Методы списков

list.append(x)

Добавляет элемент в конец списка. Ту же операцию можно сделать так $a[\text{len}(a):] = [x]$:

```
>>> a = [1, 2]
>>> a.append(3)
>>> print(a)
[1, 2, 3]
```

list.extend(L)

Расширяет существующий список за счет добавления всех элементов из списка L . Эквивалентно команде $a[\text{len}(a):] = L$:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a.extend(b)
>>> print(a)
[1, 2, 3, 4]
```

list.insert(i, x)

Вставить элемент x в позицию i . Первый аргумент – индекс элемента после которого будет вставлен элемент x :

```
>>> a = [1, 2]
>>> a.insert(0, 5)
>>> print(a)
[5, 1, 2]
>>> a.insert(len(a), 9)
>>> print(a)
[5, 1, 2, 9]
```

list.remove(x)

Удаляет первое вхождение элемента x из списка:

```
>>> a = [1, 2, 3]
>>> a.remove(1)
>>> print(a)
[2, 3]
```

list.pop([i])

Удаляет элемент из позиции i и возвращает его. Если использовать метод без аргумента, то будет удален последний элемент из списка:

```
>>> a = [1, 2, 3, 4, 5]
>>> print(a.pop(2))
3
>>> print(a.pop())
5
>>> print(a)
[1, 2, 4]
```

list.clear()

Удаляет все элементы из списка. Эквивалентно *del a[:]*:

```
>>> a = [1, 2, 3, 4, 5]
>>> print(a)
[1, 2, 3, 4, 5]
>>> a.clear()
>>> print(a)
[]
```

list.index(x[, start[, end]])

Возвращает индекс элемента:

```
>>> a = [1, 2, 3, 4, 5]
>>> a.index(4)
3
```

list.count(x)

Возвращает количество вхождений элемента x в список:

```
>>> a=[1, 2, 2, 3, 3]
>>> print(a.count(2))
2
```

list.sort(key=None, reverse=False)

Сортирует элементы в списке по возрастанию. Для сортировки в обратном порядке используйте флаг *reverse=True*. Дополнительные возможности открывает параметр *key*, за более подробной информацией обратитесь к документации:

```
>>> a = [1, 4, 2, 8, 1]
>>> a.sort()
>>> print(a)
[1, 1, 2, 4, 8]
```

list.reverse()

Изменяет порядок расположения элементов в списке на обратный:

```
>>> a = [1, 3, 5, 7]
>>> a.reverse()
>>> print(a)
[7, 5, 3, 1]
```

list.copy()

Возвращает копию списка. Эквивалентно `a[:]`:

```
>>> a = [1, 7, 9]
>>> b = a.copy()
>>> print(a)
[1, 7, 9]
>>> print(b)
[1, 7, 9]
>>> b[0] = 8
>>> print(a)
[1, 7, 9]
>>> print(b)
[8, 7, 9]
```

7.5 List Comprehensions

List Comprehensions чаще всего на русский язык переводят как “абстракция списков” или “списковое включение”, является частью синтаксиса языка, которая предоставляет простой способ построения списков. Проще всего работу *list comprehensions* показать на примере. Допустим вам необходимо создать список целых чисел от 0 до n , где n предварительно задается. Классический способ решения данной задачи выглядел бы так:

```
>>> n = int(input())
7
>>> a=[]
>>> for i in range(n):
        a.append(i)
>>> print(a)
[0, 1, 2, 3, 4, 5, 6]
```

Использование *list comprehensions* позволяет сделать это значительно проще:

```
>>> n = int(input())
7
>>> a = [i for i in range(n)]
>>> print(a)
[0, 1, 2, 3, 4, 5, 6]
```

или вообще вот так, в случае если вам не нужно больше использовать *n*:

```
>>> a = [i for i in range(int(input()))]
7
>>> print(a)
[0, 1, 2, 3, 4, 5, 6]
```

7.6 List Comprehensions как обработчик списков

В языке *Python* есть две мощные функции для работы с коллекциями: *map* и *filter*. Они позволяют использовать функциональный стиль программирования, не прибегая к помощи циклов, для работы с такими типами как *list*, *tuple*, *set*, *dict* и т.п. Списковое включение позволяет обойтись без этих функций. Приведем несколько примеров для того, чтобы понять о чем идет речь.

Пример с заменой функции *map*.

Пусть у нас есть список и нужно получить на базе него новый, который содержит элементы первого, возведенные в квадрат. Решим эту задачу с использованием циклов:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = []

>>> for i in a:
    b.append(i ** 2)

>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [1, 4, 9, 16, 25, 36, 49]
```

Та же задача, решенная с использованием *map*, будет выглядеть так:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = list(map(lambda x: x**2, a))

>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [1, 4, 9, 16, 25, 36, 49]
```

В данном случае применена *lambda*-функция, о том, что это такое и как ее использовать можете прочитать в «10.4 *Lambda*-функция».

Через списковое включение эта задача будет решена так:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = [i**2 for i in a]
>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [1, 4, 9, 16, 25, 36, 49]
```

Пример с заменой функции *filter*.

Построим на базе существующего списка новый, состоящий только из четных чисел:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = []
>>> for i in a:
    if i%2 == 0:
        b.append(i)
>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [2, 4, 6]
```

Решим эту задачу с использованием *filter*:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = list(filter(lambda x: x % 2 == 0, a))
>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [2, 4, 6]
```

Решение через списковое включение:

```
>>> a = [1, 2, 3, 4, 5, 6, 7]
>>> b = [i for i in a if i % 2 == 0]
>>> print('a = {}\nb = {}'.format(a, b))
a = [1, 2, 3, 4, 5, 6, 7]
b = [2, 4, 6]
```


7.7 Слайсы / Срезы

Слайсы (срезы) являются очень мощной составляющей *Python*, которая позволяет быстро и лаконично решать задачи выборки элементов из списка. Выше уже был пример использования слайсов, здесь разберем более подробно работу с ними.

Создадим список для экспериментов:

```
>>> a = [i for i in range(10)]
```

Слайс задается тройкой чисел, разделенных запятой: *start:stop:step*. *Start* – позиция с которой нужно начать выборку, *stop* – конечная позиция, *step* – шаг. При этом необходимо помнить, что выборка не включает элемент, определяемый *stop*.

Рассмотрим примеры:

```
>>> # Получить копию списка
>>> a[:]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> # Получить первые пять элементов списка
>>> a[0:5]
[0, 1, 2, 3, 4]
>>> # Получить элементы с 3-го по 7-ой
>>> a[2:7]
[2, 3, 4, 5, 6]
>>> # Взять из списка элементы с шагом 2
>>> a[::2]
[0, 2, 4, 6, 8]
>>> # Взять из списка элементы со 2-го по 8-ой с шагом 2
>>> a[1:8:2]
```

Слайсы можно сконструировать заранее, а потом уже использовать по мере необходимости. Это возможно сделать, в виду того, что слайс – это объект класса *slice*. Ниже приведен пример, демонстрирующий эту функциональность:

```
>>> s = slice(0, 5, 1)
>>> a[s]
[0, 1, 2, 3, 4]
>>> s = slice(1, 8, 2)
>>> a[s]
[1, 3, 5, 7]
```

7.8 “List Comprehensions”... в генераторном режиме

Есть ещё один способ создания списков, который похож на списковое включение, но результатом работы является не объект класса *list*, а генератор. Подробно про генераторы написано в “Уроке 15. Итераторы и генераторы”.

Предварительно импортируем модуль `sys`, он нам понадобится:

```
>>> import sys
```

Создадим список, используя списковое включение :

```
>>> a = [i for i in range(10)]
```

проверим тип переменной `a`:

```
>>> type(a)
<class 'list'>
```

и посмотрим сколько она занимает памяти в байтах:

```
>>> sys.getsizeof(a)
192
```

Для создания объекта-генератора, используется синтаксис такой же как и для спискового включения, только вместо квадратных скобок используются круглые:

```
>>> b = (i for i in range(10))
>>> type(b)
<class 'generator'>
>>> sys.getsizeof(b)
120
```

Обратите внимание, что тип этого объекта *generator*, и в памяти он занимает места меньше, чем список, это объясняется тем, что в первом случае в памяти хранится весь набор чисел от 0 до 9, а во втором функция, которая будет нам генерировать числа от 0 до 9. Для наших примеров разница в размере не существенна, рассмотрим вариант с 10000 элементами:

```
>>> c = [i for i in range(10000)]
>>> sys.getsizeof(c)
87624
```

```
>>> d = (i for i in range(10000))
>>> sys.getsizeof(d)
120
```

Сейчас уже разница существенна, как вы уже поняли, размер генератора в данном случае не будет зависеть от количества чисел, которые он должен создать.

Если вы решаете задачу обхода списка, то принципиальной разницы между списком и генератором не будет:

```
>>> for val in a:
    print(val, end=' ')
0 1 2 3 4 5 6 7 8 9
>>> for val in b:
    print(val, end=' ')
0 1 2 3 4 5 6 7 8 9
```

Но с генератором нельзя работать также как со списком: нельзя обратиться к элементу по индексу и т.п.

Урок 8. Кортежи (*tuple*)

Данный урок посвящен кортежам (*tuple*) в *Python*. Основное внимание уделено вопросу использования кортежей, рассмотрены способы создания и основные приемы работы с ними. Также затронем тему преобразования кортежа в список и обратно.

8.1 Что такое кортеж (*tuple*) в *Python*?

Кортеж (*tuple*) – это неизменяемая структура данных, которая по своему подобию очень похожа на список. Как вы наверное знаете, а если нет, то, пожалуйста, ознакомьтесь с седьмым уроком, список – это изменяемый тип данных. Т.е. если у нас есть список `a = [1, 2, 3]` и мы хотим заменить второй элемент с 2 на 15, то мы можем это сделать, напрямую обратившись к элементу списка:

```
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
>>> a[1] = 15
>>> print(a)
[1, 15, 3]
```

С кортежем мы не можем производить такие операции, т.к. элементы его изменять нельзя:

```
>>> b = (1, 2, 3)
>>> print(b)
(1, 2, 3)
>>> b[1] = 15
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    b[1] = 15
TypeError: 'tuple' object does not support item assignment
```

8.2 Зачем нужны кортежи в *Python*?

Существует несколько причин, по которым стоит использовать кортежи вместо списков. Одна из них – это обезопасить данные от случайного изменения.

Если мы получили откуда-то массив данных, и у нас есть желание поработать с ним, но при этом непосредственно менять данные мы не собираемся, тогда, это как раз тот случай, когда кортежи придутся как нельзя кстати. Используя их в данной задаче, мы дополнительно получаем сразу несколько бонусов – во-первых, это экономия места. Дело в том, что кортежи в памяти занимают меньший объем по сравнению со списками:

```
>>> lst = [10, 20, 30]
>>> tpl = (10, 20, 30)
>>> print(lst.__sizeof__())
32
>>> print(tpl.__sizeof__())
24
```

Во-вторых – прирост производительности, который связан с тем, что кортежи работают быстрее, чем списки (т.е. на операции перебора элементов и т.п. будет тратиться меньше времени). Важно также отметить, что кортежи можно использовать в качестве ключа у словаря.

8.3 Создание, удаление кортежей и работа с его элементами

8.3.1 Создание кортежей

Для создания пустого кортежа можно воспользоваться одной из следующих команд:

```
>>> a = ()
>>> print(type(a))
<class 'tuple'>
>>> b = tuple()
>>> print(type(b))
<class 'tuple'>
```

Кортеж с заданным содержанием создается также как список, только вместо квадратных скобок используются круглые:

```
>>> a = (1, 2, 3, 4, 5)
>>> print(type(a))
<class 'tuple'>
>>> print(a)
(1, 2, 3, 4, 5)
```

При желании можно воспользоваться функцией *tuple()*:

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

8.3.2 Доступ к элементам кортежа

Доступ к элементам кортежа осуществляется также как к элементам списка – через указание индекса. Но, как уже было сказано – изменять элементы кортежа нельзя!

```
>>> a = (1, 2, 3, 4, 5)
>>> print(a[0])
1
>>> print(a[1:3])
(2, 3)
>>> a[1] = 3
Traceback (most recent call last):
  File "<pyshell#24>", line 1, in <module>
    a[1] = 3
TypeError: 'tuple' object does not support item assignment
```

8.3.3 Удаление кортежей

Удалить отдельные элементы из кортежа невозможно:

```
>>> a = (1, 2, 3, 4, 5)
>>> del a[0]
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in <module>
    del a[0]
TypeError: 'tuple' object doesn't support item deletion
```

Но можно удалить кортеж целиком:

```
>>> del a
>>> print(a)
Traceback (most recent call last):
  File "<pyshell#28>", line 1, in <module>
    print(a)
NameError: name 'a' is not defined
```

8.3.4 Преобразование кортежа в список и обратно

На базе кортежа можно создать список, верно и обратное утверждение. Для превращения списка в кортеж достаточно передать его в качестве аргумента функции *tuple()*:

```
>>> lst = [1, 2, 3, 4, 5]
>>> print(type(lst))
<class 'list'>
>>> print(lst)
[1, 2, 3, 4, 5]
>>> tpl = tuple(lst)
>>> print(type(tpl))
<class 'tuple'>
>>> print(tpl)
(1, 2, 3, 4, 5)
```

Обратная операция также является корректной:

```
>>> tpl = (2, 4, 6, 8, 10)
>>> print(type(tpl))
<class 'tuple'>
>>> print(tpl)
(2, 4, 6, 8, 10)
>>> lst = list(tpl)
>>> print(type(lst))
<class 'list'>
>>> print(lst)
[2, 4, 6, 8, 10]
```

Урок 9. Словари (*dict*)

9.1 Что такое словарь (*dict*) в Python?

Словарь (*dict*) представляет собой структуру данных (которая еще называется ассоциативный массив), предназначенную для хранения произвольных объектов с доступом по ключу. Данные в словаре хранятся в формате ключ – значение. Если вспомнить такую структуру как список, то доступ к его элементам осуществляется по индексу, который представляет собой целое положительное число, причем мы сами, непосредственно, не участвуем в его создании (индекса). В словаре аналогом индекса является ключ, при этом ответственность за его формирование ложится на программиста.

9.2 Создание, изменение, удаление словарей и работа с его элементами

9.2.1 Создание словаря

Пустой словарь можно создать, используя функцию *dict()*, либо просто указав пустые фигурные скобки:

```
>>> d1 = dict()
>>> print(type(d1))
<class 'dict'>
>>> d2 = {}
>>> print(type(d2))
<class 'dict'>
```

Если необходимо создать словарь с заранее подготовленным набором данных, то можно использовать один из перечисленных выше подходов, но с перечислением групп ключ-значение:

```
>>> d1 = dict(Ivan="manager", Mark="worker")
>>> print(d1)
{'Mark': 'worker', 'Ivan': 'manager'}
>>> d2 = {"A1": "123", "A2": "456"}
>>> print(d2)
{'A2': '456', 'A1': '123'}
```


9.2.2 Добавление и удаление элемента

Чтобы добавить элемент в словарь нужно указать новый ключ и значение:

```
>>> d1 = {"Russia": "Moscow", "USA": "Washington"}
>>> d1["China"] = "Beijing"
>>> print(d1)
{'Russia': 'Moscow', 'USA': 'Washington', 'China': 'Beijing'}
```

Для удаления элемента из словаря можно воспользоваться командой *del*:

```
>>> d2 = {"A1": "123", "A2": "456"}
>>> del d2["A1"]
>>> print(d2)
{'A2': '456'}
```

9.2.3 Работа со словарем

Проверка наличия ключа в словаре производится с помощью оператора *in*:

```
>>> d2 = {"A1": "123", "A2": "456"}
>>> "A1" in d2
True
>>> "A3" in d2
False
```

Доступ к элементу словаря, осуществляется так же как доступ к элементу списка, только в качестве индекса указывается ключ:

```
>>> d1 = {"Russia": "Moscow", "USA": "Washington"}
>>> d1["Russia"]
'Moscow'
```

9.3 Методы словарей

У словарей доступен следующий набор методов.

clear()

Удаляет все элементы словаря:

```
>>> d2 = {"A1": "123", "A2": "456"}
>>> print(d2)
{'A2': '456', 'A1': '123'}
>>> d2.clear()
>>> print(d2)
{}
```

copy()

Создает новую копию словаря:

```
>>> d2 = {"A1": "123", "A2": "456"}
>>> d3 = d2.copy()
>>> print(d3)
{'A1': '123', 'A2': '456'}
>>> d3["A1"] = "789"
>>> print(d2)
{'A2': '456', 'A1': '123'}
>>> print(d3)
{'A1': '789', 'A2': '456'}
```

fromkeys(seq[, value])

Создает новый словарь с ключами из *seq* и значениями из *value*. По умолчанию *value* присваивается значение *None*.

get(key)

Возвращает значение из словаря по ключу *key*:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.get("A1")
'123'
```

items()

Возвращает элементы словаря (ключ, значение) в отформатированном виде:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.items()
dict_items([('A2', '456'), ('A1', '123')])
```

keys()

Возвращает ключи словаря:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.keys()
dict_keys(['A2', 'A1'])
```

pop(key[, default])

Если ключ *key* есть в словаре, то данный элемент удаляется из словаря и возвращается значение по этому ключу, иначе будет возвращено значение *default*. Если *default* не указан и запрашиваемый ключ отсутствует в словаре, то будет вызвано исключение *KeyError*:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.pop("A1")
'123'
>>> print(d)
{'A2': '456'}
```

popitem()

Удаляет и возвращает пару (ключ, значение) из словаря. Если словарь пуст, то будет вызвано исключение *KeyError*:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.popitem()
('A2', '456')
>>> print(d)
{'A1': '123'}
```

setdefault(key[, default])

Если ключ *key* есть в словаре, то возвращается значение по ключу. Если такого ключа нет, то в словарь вставляется элемент с ключом *key* и значением *default*, если *default* не определен, то по умолчанию присваивается *None*:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.setdefault("A3", "777")
'777'
>>> print(d)
{'A2': '456', 'A3': '777', 'A1': '123'}
>>> d.setdefault("A1")
'123'
>>> print(d)
{'A2': '456', 'A3': '777', 'A1': '123'}
```

update([other])

Обновляет словарь парами (*key/value*) из *other*, если ключи уже существуют, то обновляет их значения:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.update({"A1": "333", "A3": "789"})
>>> print(d)
{'A2': '456', 'A3': '789', 'A1': '333'}
```

values()

Возвращает значения элементов словаря:

```
>>> d = {"A1": "123", "A2": "456"}
>>> d.values()
dict_values(['456', '123'])
```

Урок 10. Функции в *Python*

Урок посвящен созданию функций в *Python* и работе с ними (передача аргументов, возврат значения и т.п.). Также рассмотрены *lambda*-функций, их особенности и использование.

10.1 Что такое функция в *Python*?

По своей сути функции в *Python* практически ничем не отличаются от функций из других языков программирования. Функцией называют именованный фрагмент программного кода, к которому можно обратиться из другого места вашей программы (но есть *lambda*-функции, у которых нет имени, о них будет рассказано в конце урока). Как правило, функции создаются для работы с данными, которые передаются ей в качестве аргументов, также функция может формировать некоторое возвращаемое значение.

10.2 Создание функций

Для создания функции используется ключевое слово *def*, после которого указывается имя и список аргументов в круглых скобках. Тело функции выделяется также как тело условия (или цикла): четырьмя пробелами. Таким образом самая простая функция, которая ничего не делает, будет выглядеть так:

```
def fun() :  
    pass
```

Возврат значения функцией осуществляется с помощью ключевого слова *return*, после которого указывается возвращаемое значение. Пример функции возвращающей единицу представлен ниже:

```
>>> def fun() :  
    return 1  
  
>>> fun()  
1
```

10.3 Работа с функциями

Во многих случаях функции используют для обработки данных. Эти данные могут быть глобальными, либо передаваться в функцию через аргументы. Список

аргументов определяется на этапе реализации и указывается в круглых скобках после имени функции. Например операцию сложения двух аргументов можно реализовать вот так:

```
>>> def summa(a, b):  
    return a + b  
  
>>> summa(3, 4)  
7
```

Рассмотрим еще два примера использования функции: вычисление числа Фибоначчи с использованием рекурсии и вычисление факториала с использованием цикла.

Вычисление числа Фибоначчи:

```
>>> def fibb(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    elif n == 2:  
        return 1  
    else:  
        return fibb(n-1) + fibb(n-2)  
  
>>> print(fibb(10))  
55
```

Вычисление факториала:

```
>>> def factorial(n):  
    prod = 1  
    for i in range(1, n+1):  
        prod *= i  
    return prod  
  
>>> print(factorial(5))  
120
```

Функцию можно присвоить переменной и использовать ее, если необходимо сократить имя. В качестве примера можно привести вариант использования функции вычисления факториала из пакета *math*:

```
>>> import math
>>> f = math.factorial
>>> print(f(5))
120
```

10.4 *Lambda*-функции

Lambda-функция – это безымянная функция с произвольным числом аргументов и вычисляющая одно выражение. Тело такой функции не может содержать более одной инструкции (или выражения). Данную функцию можно использовать в рамках каких-либо конвейерных вычислений (например внутри *filter()*, *map()* и *reduce()*) либо самостоятельно, в тех местах, где требуется произвести какое-либо вычисление, которые удобно “завернуть” в функцию:

```
>>> (lambda x: x**2)(5)
25
```

Lambda-функцию можно присвоить какой-либо переменной и в дальнейшем использовать ее в качестве имени функции:

```
>>> sqrt = lambda x: x**0.5
>>> sqrt(25)
5.0
```

Списки можно обрабатывать *lambda*-функциями внутри таких функций как *map()*, *filter()*, *reduce()*. Функция *map* принимает два аргумента, первый – это функция, которая будет применена к каждому элементу списка, а второй – это список, который нужно обработать:

```
>>> l = [1, 2, 3, 4, 5, 6, 7]
>>> list(map(lambda x: x**3, l))
[1, 8, 27, 64, 125, 216, 343]
```

Урок 11. Работа с исключениями

Данный урок посвящен исключениям и работе с ними. Основное внимание уделено понятию исключения в языках программирования, обработке исключений в Python, их генерации и созданию пользовательских исключений.

11.1 Исключения в языках программирования

Исключениями (*exceptions*) в языках программирования называют проблемы, возникающие в ходе выполнения программы, которые допускают возможность дальнейшей ее работы в рамках основного алгоритма. Типичным примером исключения является деление на ноль, невозможность считать данные из файла (устройства), отсутствие доступной памяти, доступ к закрытой области памяти и т.п. Для обработки таких ситуаций в языках программирования, как правило, предусматривается специальный механизм, который называется обработка исключений (*exception handling*).

Исключения разделяют на синхронные и асинхронные. Синхронные исключения могут возникнуть только в определенных местах программы. Например, если у вас есть код, который открывает файл и считывает из него данные, то исключение типа “ошибка чтения данных” может произойти только в указанном куске кода. Асинхронные исключения могут возникнуть в любой момент работы программы, они, как правило, связаны с какими-либо аппаратными проблемами, либо приходом данных. В качестве примера можно привести сигнал отключения питания.

В языках программирования чаще всего предусматривается специальный механизм обработки исключений. Обработка может быть с возвратом, когда после обработки исключения выполнение программы продолжается с того места, где оно возникло. И обработка без возврата, в этом случае, при возникновении исключения, осуществляется переход в специальный, заранее подготовленный, блок кода.

Различают структурную и неструктурную обработку исключений. Неструктурная обработка предполагает регистрацию функции обработчика для каждого исключения, соответственно данная функция будет вызвана при возникновении конкретного исключения. Для структурной обработки язык

программирования должен поддерживать специальные синтаксические конструкции, которые позволяют выделить код, который необходимо контролировать и код, который нужно выполнить при возникновении исключительной ситуации.

В *Python* выделяют два различных вида ошибок: синтаксические ошибки и исключения.

11.2 Синтаксические ошибки в *Python*

Синтаксические ошибки возникают в случае если программа написана с нарушениями требований *Python* к синтаксису. Определяются они в процессе парсинга программы. Ниже представлен пример с ошибочным написанием функции *print*:

```
>>> for i in range(10):
    prin("hello!")
Traceback (most recent call last):
  File "<pyshell#2>", line 2, in <module>
    prin("hello!")
NameError: name 'prin' is not defined
```

11.3 Исключения в *Python*

Второй вид ошибок – это исключения. Они возникают в случае если синтаксически программа корректна, но в процессе выполнения возникает ошибка (деление на ноль и т.п.). Более подробно про понятие исключения написано выше, в разделе “исключения в языках программирования”.

Пример исключения *ZeroDivisionError*, которое возникает при делении на 0:

```
>>> a = 10
>>> b = 0
>>> c = a / b
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    c = a / b
ZeroDivisionError: division by zero
```

В *Python* исключения являются определенным типом данных, через который пользователь (программист) получает информацию об ошибке. Если в коде программы исключение не обрабатывается, то приложение останавливается и в консоли печатается подробное описание произошедшей ошибки с указанием места в программе, где она произошла и тип этой ошибки.

11.4 Иерархия исключений в *Python*

Существует довольно большое количество встроенных типов исключений в языке *Python*, все они составляют определенную иерархию, которая выглядит так, как показано ниже.

```
BaseException
+- SystemExit
+- KeyboardInterrupt
+- GeneratorExit
+- Exception
    +- StopIteration
    +- StopAsyncIteration
    +- ArithmeticError
    |     +- FloatingPointError
    |     +- OverflowError
    |     +- ZeroDivisionError
    +- AssertionError
    +- AttributeError
    +- BufferError
    +- EOFError
    +- ImportError
    |     +- ModuleNotFoundError
    +- LookupError
    |     +- IndexError
    |     +- KeyError
    +- MemoryError
    +- NameError
    |     +- UnboundLocalError
    +- OSError
    |     +- BlockingIOError
    |     +- ChildProcessError
    |     +- ConnectionError
    |     |     +- BrokenPipeError
    |     |     +- ConnectionAbortedError
    |     |     +- ConnectionRefusedError
    |     |     +- ConnectionResetError
```

```

|     +- FileNotFoundError
|     +- InterruptedError
|     +- IsADirectoryError
|     +- NotADirectoryError
|     +- PermissionError
|     +- ProcessLookupError
|     +- TimeoutError
+- ReferenceError
+- RuntimeError
|     +- NotImplementedError
|     +- RecursionError
+- SyntaxError
|     +- IndentationError
|         +- TabError
+- SystemError
+- TypeError
+- ValueError
|     +- UnicodeError
|         +- UnicodeDecodeError
|         +- UnicodeEncodeError
|         +- UnicodeTranslateError
+- Warning
    +- DeprecationWarning
    +- PendingDeprecationWarning
    +- RuntimeWarning
    +- SyntaxWarning
    +- UserWarning
    +- FutureWarning
    +- ImportWarning
    +- UnicodeWarning
    +- BytesWarning
    +- ResourceWarning

```

Как видно из приведенной выше схемы, все исключения являются подклассом исключения *BaseException*. Более подробно об иерархии исключений и их описании можете прочитать в официальной документации <https://docs.python.org/3/library/exceptions.html>.

11.5 Обработка исключений в *Python*

Обработка исключений нужна для того, чтобы приложение не завершалось аварийно каждый раз, когда возникает исключение. Для этого блок кода, в котором

возможно появление исключительной ситуации необходимо поместить во внутрь синтаксической конструкции *try...except*:

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except Exception as e:
    print("Error! " + str(e))
print("stop")
```

В приведенной выше программе возможных два вида исключений – это *ValueError*, возникающее в случае, если на запрос программы “введите число”, вы введете строку, и *ZeroDivisionError* – если вы введете в качестве числа 0.

Вывод программы при вводе нулевого числа будет таким:

```
start
input number: 0
Error!
stop
```

Если бы инструкций *try...except* не было, то при выбросе любого из исключений программа аварийно завершится:

```
print("start")
val = int(input("input number: "))
tmp = 10 / val
print(tmp)
print("stop")
```

Если ввести 0 на запрос приведенной выше программы, произойдет ее остановка с распечаткой сообщения об исключении:

```
start
input number: 0
Traceback (most recent call last):
  File "F:/work/programming/python/devpractice/tmp.py", line 3, in <module>
    tmp = 10 / val
ZeroDivisionError: division by zero
```

Обратите внимание, надпись *stop* уже не печатается в конце вывода программы.

Согласно документу по языку *Python*, описывающему ошибки и исключения, оператор *try* работает [следующим образом](#):

- Вначале выполняется код, находящийся между операторами *try* и *except*.
- Если в ходе его выполнения исключения не произошло, то код в блоке *except* пропускается, а код в блоке *try* выполняется весь до конца.
- Если исключение происходит, то выполнение в рамках блока *try* прерывается и выполняется код в блоке *except*. При этом для оператора *except* можно указать, какие исключения можно обрабатывать в нем. При возникновении исключения, ищется именно тот блок *except*, который может обработать данное исключение.
- Если среди *except* блоков нет подходящего для обработки исключения, то оно передается наружу из блока *try*. В случае, если обработчик исключения так и не будет найден, то исключение будет необработанным (*unhandled exception*) и программа аварийно остановится.

Для указания набора исключений, который должен обрабатывать данный блок *except* их необходимо перечислить в скобках (круглых) через запятую после оператора *except*.

Если бы мы в нашей программе хотели обрабатывать только *ValueError* и *ZeroDivisionError*, то программа выглядела бы так:

```
print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except(ValueError, ZeroDivisionError):
    print("Error!")
print("stop")
```

Или так, если хотим обрабатывать *ValueError*, *ZeroDivisionError* по отдельности, и, при этом, сохранить работоспособность при возникновении исключений отличных от вышеперечисленных:

```

print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except ValueError:
    print("ValueError!")
except ZeroDivisionError:
    print("ZeroDivisionError!")
except:
    print("Error!")
print("stop")

```

Существует возможность передать подробную информацию о произошедшем исключении в код внутри блока *except*:

```

print("start")
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except ValueError as ve:
    print("ValueError! {0}".format(ve))
except ZeroDivisionError as zde:
    print("ZeroDivisionError! {0}".format(zde))
except Exception as ex:
    print("Error! {0}".format(ex))
print("stop")

```

11.6 Использование *finally* в обработке исключений

Для выполнения определенного программного кода при выходе из блока *try/except*, используйте оператор *finally*:

```
try:
    val = int(input("input number: "))
    tmp = 10 / val
    print(tmp)
except:
    print("Exception")
finally:
    print("Finally code")
```

Не зависимо от того, возникнет или нет во время выполнения кода в блоке *try* исключение, код в блоке *finally* все равно будет выполнен.

Если необходимо выполнить какой-то программный код, в случае если в процессе выполнения блока *try* не возникло исключений, то можно использовать оператор *else*:

```
try:
    f = open("tmp.txt", "r")
    for line in f:
        print(line)
    f.close()
except Exception as e:
    print(e)
else:
    print("File was readed")
```

11.7 Генерация исключений в *Python*

Для принудительной генерации исключения используется инструкция *raise*. Самый простой пример работы с *raise* может выглядеть так:

```
try:
    raise Exception("Some exception")
except Exception as e:
    print("Exception exception " + str(e))
```

Таким образом, можно “вручную” вызывать исключения при необходимости.

11.8 Пользовательские исключения (*User-defined Exceptions*) в Python

В Python можно создавать собственные исключения. Такая практика позволяет увеличить гибкость процесса обработки ошибок в рамках той предметной области, для которой написана ваша программа.

Для реализации собственного типа исключения необходимо создать класс, являющийся наследником от одного из классов исключений:

```
class NegValException(Exception):  
    pass  
  
try:  
    val = int(input("input positive number: "))  
    if val < 0:  
        raise NegValException("Neg val: " + str(val))  
    print(val + 10)  
except NegValException as e:  
    print(e)
```


Урок 12. Ввод-вывод данных. Работа с файлами

В уроке рассмотрены основные способы ввода и вывода данных в *Python* с использованием консоли и работа с файлами: открытие, закрытие, чтение и запись.

12.1 Вывод данных в консоль

Один из самых распространенных способов вывести данные в *Python* – это напечатать их в консоли. Если вы находитесь на этапе изучения языка, такой способ является основным для того, чтобы быстро просмотреть результат своей работы. Для вывода данных в консоль используется функция *print*.

Рассмотрим основные способы использования данной функции:

```
>>> print("Hello")
Hello
>>> print("Hello, " + "world!")
Hello, world!
>>> print("Age: " + str(23))
Age: 23
```

По умолчанию, для разделения элементов в функции *print* используется пробел:

```
>>> print("A", "B", "C")
A B C
```

Для замены разделителя необходимо использовать параметр *sep* функции *print*:

```
print("A", "B", "C", sep="#")
A#B#C
```

В качестве конечного элемента выводимой строки, используется символ перевода строки:

```
>>> for i in range(3):
    print("i: " + str(i))

i: 0
i: 1
i: 2
```

Для его замены используется параметр *end*:

```
>>> for i in range(3):
        print("[i: " + str(i) + "]", end=" -- ")
[i: 0] -- [i: 1] -- [i: 2] --
```

12.2 Ввод данных с клавиатуры

Для считывания вводимых с клавиатуры данных используется функция *input()*:

```
>>> input()
test
'test'
```

Для сохранения данных в переменной используется следующий синтаксис:

```
>>> a = input()
hello
>>> print(a)
hello
```

Если считывается с клавиатуры целое число, то строку, получаемую с помощью функции *input()*, можно передать сразу в функцию *int()*:

```
>>> val = int(input())
123
>>> print(val)
123
>>> type(val)
<class 'int'>
```

Для вывода строки-приглашения, используйте ее в качестве аргумента функции *input()*:

```
>>> tv = int(input("input number: "))
input number: 334
>>> print(tv)
334
```

Преобразование строки в список осуществляется с помощью метода `split()`, по умолчанию, в качестве разделителя, используется пробел:

```
>>> l = input().split()
1 2 3 4 5 6 7
>>> print(l)
['1', '2', '3', '4', '5', '6', '7']
```

Разделитель можно заменить, указав его в качестве аргумента метода `split()`:

```
>>> nl = input().split("-")
1-2-3-4-5-6-7
>>> print(nl)
['1', '2', '3', '4', '5', '6', '7']
```

Для считывания списка чисел с одновременным приведением их к типу `int` можно воспользоваться вот такой конструкцией:

```
>>> nums = map(int, input().split())
1 2 3 4 5 6 7
>>> print(list(nums))
[1, 2, 3, 4, 5, 6, 7]
```

12.3 Работа с файлами

12.3.1 Открытие и закрытие файла

Для открытия файла используется функция `open()`, которая возвращает файловый объект. Наиболее часто используемый вид данной функции выглядит так `open(имя_файла, режим_доступа)`.

Для указания режима доступа используются следующие символы:

- 'r' – открыть файл для чтения;
- 'w' – открыть файл для записи;
- 'x' – открыть файл с целью создания, если файл существует, то вызов функции `open` завершится с ошибкой;
- 'a' – открыть файл для записи, при этом новые данные будут добавлены в конец файла, без удаления существующих;
- 'b' – бинарный режим;
- 't' – текстовый режим;
- '+' – открывает файл для обновления.

По умолчанию файл открывается на чтение в текстовом режиме.

У файлового объекта есть следующие атрибуты:

file.closed – возвращает *true* если файл закрыт и *false* в противном случае;

file.mode – возвращает режим доступа к файлу, при этом файл должен быть открыт;

file.name – имя файла.

```
>>> f = open("test.txt", "r")
>>> print("file.closed: " + str(f.closed))
file.closed: False
>>> print("file.mode: " + f.mode)
file.mode: r
>>> print("file.name: " + f.name)
file.name: test.txt
```

Для закрытия файла используется метод `close()`.

12.3.2 Чтение данных из файла

Чтение данных из файла осуществляется с помощью методов *read(размер)* и *readline()*.

Метод *read(размер)* считывает из файла определенное количество символов, переданное в качестве аргумента. Если использовать этот метод без аргументов, то будет считан весь файл:

```
>>> f = open("test.txt", "r")
>>> f.read()
'1 2 3 4 5\nWork with file\n'
>>> f.close()
```

В качестве аргумента метода можно передать количество символом, которое нужно считать:

```
>>> f = open("test.txt", "r")
>>> f.read(5)
'1 2 3'
>>> f.close()
```

Метод `readline()` позволяет считать строку из открытого файла:

```
>>> f = open("test.txt", "r")
>>> f.readline()
'1 2 3 4 5\n'
>>> f.close()
```

Построчное считывание можно организовать с помощью оператора `for`:

```
>>> f = open("test.txt", "r")
>>> for line in f:
...     print(line)
...
1 2 3 4 5
Work with file
>>> f.close()
```

12.3.3 Запись данных в файл

Для записи данных файл используется метод `write(строка)`, при успешной записи он вернет количество записанных символов:

```
>>> f = open("test.txt", "a")
>>> f.write("Test string")
11
>>> f.close()
```

12.3.4 Дополнительные методы для работы с файлами

Метод `tell()` возвращает текущую позицию “условного курсора” в файле. Например, если вы считали пять символов, то “курсор” будет установлен в позицию 5:

```
>>> f = open("test.txt", "r")
>>> f.read(5)
'1 2 3'
>>> f.tell()
5
>>> f.close()
```

Метод `seek(позиция)` выставляет позицию в файле:

```
>>> f = open("test.txt", "r")
>>> f.tell()
0
>>> f.seek(8)
8
>>> f.read(1)
'5'
>>> f.tell()
9
>>> f.close()
```

Хорошей практикой при работе с файлами является применение оператора *with*. При его использовании нет необходимости закрывать файл, при завершении работы с ним, эта операция будет выполнена автоматически:

```
>>> with open("test.txt", "r") as f:
...     for line in f:
...         print(line)
...
1 2 3 4 5
Work with file
Test string
>>> f.closed
True
```

Урок 13. Модули и пакеты

Модули и пакеты значительно упрощают работу программиста. Классы, объекты, функции и константы, которыми приходится часто пользоваться можно упаковать в модуль, и, в дальнейшем, загружать его в свои программы при необходимости. Пакеты позволяют формировать пространства имен для работы с модулями.

13.1 Модули в *Python*

13.1.1 Что такое модуль в *Python*?

Под модулем в *Python* понимается файл с расширением *.py*. Модули предназначены для того, чтобы в них хранить часто используемые функции, классы, константы и т.п. Можно условно разделить модули и программы: программы предназначены для непосредственного запуска, а модули для импортирования их в другие программы. Стоит заметить, что модули могут быть написаны не только на языке *Python*, но и на других языках (например *C*).

13.1.2 Как импортировать модули в *Python*?

Самый простой способ импортировать модуль в *Python* это воспользоваться конструкцией:

import имя_модуля

Импорт и использование модуля *math*, который содержит математические функции, будет выглядеть вот так:

```
>>> import math
>>> math.factorial(5)
120
```

За один раз можно импортировать сразу несколько модулей, для этого их нужно перечислить через запятую после слова *import*:

import имя_модуля1, имя_модуля2

```
>>> import math, datetime
>>> math.cos(math.pi/4)
0.707106781186547
```

```
>>> datetime.date(2017, 3, 21)
datetime.date(2017, 3, 21)
```

Если вы хотите задать псевдоним для модуля в вашей программе, можно воспользоваться вот таким синтаксисом:

import имя_модуля as новое_имя

```
>>> import math as m
>>> m.sin(m.pi/3)
0.866025403784438
```

Используя любой из вышеперечисленных подходов, при вызове функции из импортированного модуля, вам всегда придется указывать имя модуля (или псевдоним). Для того, чтобы этого избежать делайте импорт через конструкцию *from ... import...*

from имя_модуля import имя_объекта

```
>>> from math import cos
>>> cos(3.14)
0.999998731727539
```

При этом импортируется только конкретный объект (в нашем примере: функция cos), остальные функции недоступны, даже если при их вызове указать имя модуля:

```
>>> from math import cos
>>> cos(3.14)
-0.999998731727539
>>> sin(3.14)
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    sin(3.14)
NameError: name 'sin' is not defined
>>> math.sin(3.14)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    math.sin(3.14)
NameError: name 'math' is not defined
```


Для импортирования нескольких функций из модуля, можно перечислить их имена через запятую:

from имя_модуля import имя_объекта1, имя_объекта2

```
>>> from math import cos, sin, pi
>>> cos(pi/3)
0.5000000000000000
>>> sin(pi/3)
0.866025403784438
```

Импортируемому объекту можно задать псевдоним:

from имя_модуля import имя_объекта as псевдоним_объекта

```
>>> from math import factorial as f
>>> f(4)
24
```

Если необходимо импортировать все функции, классы и т.п. из модуля, то воспользуйтесь следующей формой оператора *from ... import ...*

from имя_модуля import *

```
>>> from math import *
>>> cos(pi/2)
6.123233995736766e-17
>>> sin(pi/4)
0.707106781186547
>>> factorial(6)
720
```

13.2 Пакеты в *Python*

13.2.1 Что такое пакет в *Python*?

Пакет в *Python* – это каталог, включающий в себя другие каталоги и модули, но при этом дополнительно содержащий файл `__init__.py`. Пакеты используются для формирования пространства имен, что позволяет работать с модулями через указание уровня вложенности (через точку).

Для импортирования пакетов используется тот же синтаксис, что и для работы с модулями.

13.2.2 Использование пакетов в *Python*

Рассмотрим следующую структуру пакета:

```
fincalc  
|-- __init__.py  
|-- simper.py  
|-- compper.py  
|-- annuity.py
```

Пакет *fincalc* содержит в себе модули для работы с простыми процентами (*simper.py*), сложными процентами (*compper.py*) и аннуитетами (*annuity.py*).

Для использования функции из модуля работы с простыми процентами, можно использовать один из следующих вариантов:

```
import fincalc.simper  
fv = fincalc.simper.fv(pv, i, n)
```

```
import fincalc.simper as sp  
fv = sp.fv(pv, i, n)
```

```
from fincalc import simper  
fv = simper.fv(pv, i, n)
```

Файл *__init__.py* может быть пустым или может содержать переменную *__all__*, хранящую список модулей, который импортируется при загрузке через конструкцию

from имя_пакета import *

Например для нашего случая содержимое *__init__.py* может быть вот таким:

```
__all__ = ["simper", "compper", "annuity"]
```

Урок 14. Классы и объекты

Данный урок посвящен объектно-ориентированному программированию в *Python*. Разобраны такие темы как создание объектов и классов, работа с конструктором, наследование и полиморфизм в *Python*.

14.1 Основные понятия объектно-ориентированного программирования

Объектно-ориентированное программирование (ООП) является методологией разработки программного обеспечения, в основе которой лежит понятие класса и объекта, при этом сама программа создается как некоторая совокупность объектов, которые взаимодействуют друг с другом и с внешним миром. Каждый объект является экземпляром некоторого класса. Классы образуют иерархии.

Выделяют три основных “столпа” ООП- это инкапсуляция, наследование и полиморфизм.

Инкапсуляция

Под инкапсуляцией понимается сокрытие реализации, данных и т.п. от внешней стороны. Например, можно определить класс “холодильник”, который будет содержать следующие данные: производитель, объем, количество камер хранения, потребляемая мощность и т.п., и методы: открыть/закрыть холодильник, включить/выключить, но при этом реализация того, как происходит непосредственно включение и выключение пользователю вашего класса не доступна, что позволяет ее менять без опасения, что это может отразиться на использующей класс «холодильник» программе. При этом класс становится новым типом данных в рамках разрабатываемой программы. Можно создавать переменные этого нового типа, такие переменные называются объектами.

Наследование

Под наследованием понимается возможность создания нового класса на базе существующего. Наследование предполагает наличие отношения “является” между классом наследником и классом родителем. При этом класс потомок будет содержать те же атрибуты и методы, что и базовый класс, но при этом его можно (и нужно) расширять через добавление новых методов и атрибутов.

Примером базового класса, демонстрирующего наследование, можно определить класс “автомобиль”, имеющий атрибуты: масса, мощность двигателя, объем топливного бака и методы: завести и заглушить. У такого класса может быть потомок – “грузовой автомобиль”, он будет содержать те же атрибуты и методы, что и класс “автомобиль”, и дополнительные свойства: количество осей, мощность компрессора и т.п.

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.

14.2 Классы в *Python*

14.2.1 Создание классов и объектов

Создание класса в *Python* начинается с инструкции *class*. Вот так будет выглядеть минимальный класс:

```
class C:  
    pass
```

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

Для того чтобы создать объект класса необходимо воспользоваться следующим синтаксисом:

```
имя_объекта = имя_класса()
```

14.2.2 Статические и динамические атрибуты класса

Как уже было сказано выше, класс может содержать атрибуты и методы. Атрибут может быть статическим и динамическим (уровня объекта класса). Суть в том, что для работы со статическим атрибутом, вам не нужно создавать экземпляры класса, а для работы с динамическим – нужно.

Пример:

```
class Rectangle:
    default_color = "green"
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

В представленном выше классе, атрибут *default_color* – это статический атрибут, и доступ к нему, как было сказано выше, можно получить не создавая объект класса *Rectangle*:

```
>>> Rectangle.default_color
'green'
```

width и *height* – это динамические атрибуты, при их создании было использовано ключевое слово *self*. Пока просто примите это как должное, более подробно про *self* будет рассказано ниже. Для доступа к *width* и *height* предварительно нужно создать объект класса *Rectangle*:

```
>>> rect = Rectangle(10, 20)
>>> rect.width
10
>>> rect.height
20
```

Если обратиться через класс, то получим ошибку:

```
>>> Rectangle.width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Rectangle' has no attribute 'width'
```

При этом, если вы обратитесь к статическому атрибуту через экземпляр класса, то все будет ОК, до тех пор, пока вы не попытаетесь его поменять.

Проверим ещё раз значение атрибута *default_color*:

```
>>> Rectangle.default_color
'green'
```

Присвоим ему новое значение:

```
>>> Rectangle.default_color = "red"
>>> Rectangle.default_color
'red'
```

Создадим два объекта класса *Rectangle* и проверим, что *default_color* у них совпадает:

```
>>> r1 = Rectangle(1, 2)
>>> r2 = Rectangle(10, 20)
>>> r1.default_color
'red'
>>> r2.default_color
'red'
```

Если поменять значение *default_color* через имя класса *Rectangle*, то все будет ожидаемо: у объектов *r1* и *r2* это значение изменится, но если поменять его через экземпляр класса, то у экземпляра будет создан атрибут с таким же именем как статический, а доступ к последнему будет потерян.

Меняем *default_color* через *r1*:

```
>>> r1.default_color = "blue"
>>> r1.default_color
'blue'
```

При этом у *r2* остается значение статического атрибута:

```
>>> r2.default_color
'red'
>>> Rectangle.default_color
'red'
```

Следует сказать, что напрямую работать с атрибутами – не очень хорошая идея, лучше для этого использовать свойства.

14.2.3 Методы класса

Добавим к нашему классу метод. Метод – это функция, находящаяся внутри класса и выполняющая определенную работу.

Методы бывают статическими, классовыми (среднее между статическими и обычными) и уровня класса (будем их называть просто словом метод). Статический метод создается с декоратором `@staticmethod`, классовой – с декоратором `@classmethod`, первым аргументом в него передается `cls`, обычный метод создается без специального декоратора, ему первым аргументом передается `self`:

```
class MyClass:
    @staticmethod
    def ex_static_method():
        print("static method")

    @classmethod
    def ex_class_method(cls):
        print("class method")

    def ex_method(self):
        print("method")
```

Статический и классовой метод можно вызвать, не создавая экземпляра класса, для вызова `ex_method()` нужен объект:

```
>>> MyClass.ex_static_method()
static method
>>> MyClass.ex_class_method()
class method
>>> MyClass.ex_method()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: ex_method() missing 1 required positional argument: 'self'

>>> m = MyClass()
>>> m.ex_method()
method
```

14.2.4 Конструктор класса и инициализация экземпляра класса

В *Python* разделяют конструктор класса и метод для инициализации экземпляра класса. Конструктор класса - это метод `__new__(cls, *args, **kwargs)` для инициализации экземпляра класса используется метод `__init__(self)`. При этом, как вы могли заметить `__new__` – это классовый метод, а `__init__` таким не является. Метод `__new__` редко переопределяется, чаще используется реализация базового класса *object* (см. раздел Наследование), `__init__` же наоборот является очень удобным способом задать параметры объекта при его создании.

Создадим реализацию класса *Rectangle* с измененным конструктором и инициализатором, через который задается ширина и высота прямоугольника:

```
class Rectangle:
    def __new__(cls, *args, **kwargs):
        print("Hello from __new__")
        return super().__new__(cls)

    def __init__(self, width, height):
        print("Hello from __init__")
        self.width = width
        self.height = height
```

```
>>> rect = Rectangle(10, 20)
Hello from __new__
Hello from __init__
```

```
>>> rect.width
10
>>> rect.height
20
```

14.2.5 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:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height
```

В приведенной реализации метод *area* получает доступ к атрибутам *width* и *height* для расчета площади. Если бы в качестве первого параметра не было указано *self*, то при попытке вызвать *area* программа была бы остановлена с ошибкой.

14.2.6 Уровни доступа атрибута и метода

Если вы знакомы с языками программирования *Java*, *C#*, *C++* то, наверное, уже задались вопросом: “а как управлять уровнем доступа?”. В перечисленных языках вы можете явно указать для переменной, что доступ к ней снаружи класса запрещен, это делается с помощью ключевых слов (*private*, *protected* и т. д.). В *Python* таких возможностей нет, и любой может обратиться к атрибутам и методам вашего класса, если возникнет такая необходимость. Это существенный недостаток этого языка, т.к. нарушается один из ключевых принципов ООП – инкапсуляция. Хорошим тоном считается, что для чтения/изменения какого-то атрибута должны использоваться специальные методы, которые называются *getter/setter*, их можно реализовать, но ничего не мешает изменить атрибут напрямую. При этом есть соглашение, что метод или атрибут, который начинается с нижнего подчеркивания, является скрытым, и снаружи класса трогать его не нужно (хотя сделать это можно).

Внесем соответствующие изменения в класс *Rectangle*:

```
class Rectangle:
    def __init__(self, width, height):
        self._width = width
        self._height = height
```

```

def get_width(self):
    return self._width

def set_width(self, w):
    self._width = w

def get_height(self):
    return self._height

def set_height(self, h):
    self._height = h

def area(self):
    return self._width * self._height

```

В приведенном примере для доступа к `_width` и `_height` используются специальные методы, но ничего не мешает вам обратиться к ним (атрибутам) напрямую:

```

>>> rect = Rectangle(10, 20)
>>> rect.get_width()
10
>>> rect._width
10

```

Если же атрибут или метод начинается с двух подчеркиваний, то тут напрямую вы к нему уже не обратитесь (простым способом). Модифицируем наш класс *Rectangle*:

```

class Rectangle:
    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    def get_width(self):
        return self.__width

    def set_width(self, w):
        self.__width = w

```

```

def get_height(self):
    return self.__height

def set_height(self, h):
    self.__height = h

def area(self):
    return self.__width * self.__height

```

Попытка обратиться к `__width` напрямую вызовет ошибку, нужно работать только через `get_width()`:

```

>>> rect = Rectangle(10, 20)
>>> rect.__width
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Rectangle' object has no attribute '__width'

>>> rect.get_width()
10

```

Но на самом деле это сделать можно, просто этот атрибут теперь для внешнего использования носит название: `__Rectangle__width`:

```

>>> rect.__Rectangle__width
10
>>> rect.__Rectangle__width = 20
>>> rect.get_width()
20

```

14.2.7 Свойства

Свойством называется такой метод класса, работа с которым подобна работе с атрибутом. Для объявления метода свойством необходимо использовать декоратор `@property`.

Важным преимуществом работы через свойства является то, что вы можете осуществлять проверку входных значений, перед тем как присвоить их атрибутам.

Сделаем реализацию класса *Rectangle* с использованием свойств:

```
class Rectangle:
    def __init__(self, width, height):
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self, h):
        if h > 0:
            self.__height = h
        else:
            raise ValueError

    def area(self):
        return self.__width * self.__height
```

Теперь работать с *width* и *height* можно так, как будто они являются атрибутами:

```
>>> rect = Rectangle(10, 20)
>>> rect.width
10
>>> rect.height
20
```

Можно не только читать, но и задавать новые значения свойствам:

```
>>> rect.width = 50
>>> rect.width
50
>>> rect.height = 70
>>> rect.height
70
```

Если вы обратили внимание: в *setter*'ах этих свойств осуществляется проверка входных значений, если значение меньше нуля, то будет выброшено исключение *ValueError*:

```
>>> rect.width = -10
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "test.py", line 28, in width
    raise ValueError
ValueError
```

14.3 Наследование

В организации наследования участвуют как минимум два класса: класс родитель и класс потомок. При этом возможно множественное наследование, в этом случае у класса потомка может быть несколько родителей. Не все языки программирования поддерживают множественное наследование, но в *Python* его можно использовать. По умолчанию все классы в *Python* являются наследниками от *object*, явно этот факт указывать не нужно.

Синтаксически создание класса с указанием его родителя выглядит так:

***class* имя_класса(имя_родителя1, [имя_родителя2,..., имя_родителя_n])**

Переработаем наш пример так, чтобы в нем присутствовало наследование:

```

class Figure:
    def __init__(self, color):
        self.__color = color

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, c):
        self.__color = c


class Rectangle(Figure):
    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width

    @width.setter
    def width(self, w):
        if w > 0:
            self.__width = w
        else:
            raise ValueError

    @property
    def height(self):
        return self.__height

    @height.setter
    def height(self, h):
        if h > 0:
            self.__height = h
        else:
            raise ValueError

```

```
def area(self):  
    return self.__width * self.__height
```

Родительским классом является *Figure*, который при инициализации принимает цвет фигуры и предоставляет его через свойства. *Rectangle* – класс наследник от *Figure*. Обратите внимание на его метод `__init__`: в нем первым делом вызывается конструктор (хотя это не совсем верно, но будем говорить так) его родительского класса: `super().__init__(color)`

`super` – это ключевое слово, которое используется для обращения к родительскому классу.

Теперь у объекта класса *Rectangle* помимо уже знакомых свойств *width* и *height* появилось свойство *color*:

```
>>> rect = Rectangle(10, 20, "green")  
>>> rect.width  
10  
>>> rect.height  
20  
>>> rect.color  
'green'  
>>> rect.color = "red"  
>>> rect.color  
'red'
```

14.4 Полиморфизм

Как уже было сказано во введении в рамках ООП полиморфизм, как правило, используется с позиции переопределения методов базового класса в классе наследнике. Проще всего это рассмотреть на примере. Добавим в наш базовый класс метод *info()*, который печатает сводную информацию по объекту класса *Figure* и переопределим этот метод в классе *Rectangle*, добавим в него дополнительные данные:

```
class Figure:
    def __init__(self, color):
        self.__color = color

    @property
    def color(self):
        return self.__color

    @color.setter
    def color(self, c):
        self.__color = c

    def info(self):
        print("Figure")
        print("Color: " + self.__color)

class Rectangle(Figure):
    def __init__(self, width, height, color):
        super().__init__(color)
        self.__width = width
        self.__height = height

    @property
    def width(self):
        return self.__width
```



```

@width.setter
def width(self, w):
    if w > 0:
        self.__width = w
    else:
        raise ValueError

@property
def height(self):
    return self.__height

@height.setter
def height(self, h):
    if h > 0:
        self.__height = h
    else:
        raise ValueError

def info(self):
    print("Rectangle")
    print("Color: " + self.color)
    print("Width: " + str(self.width))
    print("Height: " + str(self.height))
    print("Area: " + str(self.area()))

def area(self):
    return self.__width * self.__height

```

Посмотрим, как это работает:

```

>>> fig = Figure("orange")
>>> fig.info()
Figure
Color: orange
>>> rect = Rectangle(10, 20, "green")

```

```
>>> rect.info()  
Rectangle  
Color: green  
Width: 10  
Height: 20  
Area: 200
```

Таким образом, класс наследник может расширять функционал класса родителя.

Урок 15. Итераторы и генераторы

Генераторы и итераторы представляют собой инструменты, которые, как правило, используются для поточной обработки данных. В этом уроке рассмотрим концепцию итераторов в *Python*, научимся создавать свои итераторы и разберемся как работать с генераторами.

15.1 Итераторы в языке *Python*

Во многих современных языках программирования используют такие сущности как итераторы. Основное их назначение – это упрощение навигации по элементам объекта, который, как правило, представляет собой некоторую коллекцию (список, словарь и т.п.). Язык *Python*, в этом случае, не исключение и в нем тоже есть поддержка итераторов. Итератор представляет собой объект перечислитель, который для данного объекта выдает следующий элемент, либо бросает исключение, если элементов больше нет.

Основное место использования итераторов – это цикл *for*. Если вы перебираете элементы в некотором списке или символы в строке с помощью цикла *for*, то, фактически, это означает, что при каждой итерации цикла происходит обращение к итератору, содержащемуся в строке/списке, с требованием выдать следующий элемент, если элементов в объекте больше нет, то итератор генерирует исключение, обрабатываемое в рамках цикла *for* незаметно для пользователя.

Приведем несколько примеров, которые помогут лучше понять эту концепцию. Для начала выведем элементы произвольного списка на экран:

```
>>> num_list = [1, 2, 3, 4, 5]
>>> for i in num_list:
    print(i)
1
2
3
4
5
```

Как уже было сказано, объекты, элементы которых можно перебирать в цикле *for*, содержат в себе объект итератор, для того, чтобы его получить необходимо использовать функцию *iter()*, а для извлечения следующего элемента из итератора – функцию *next()*:

```
>>> itr = iter(num_list)
>>> print(next(itr))
1
>>> print(next(itr))
2
>>> print(next(itr))
3
>>> print(next(itr))
4
>>> print(next(itr))
5
>>> print(next(itr))
Traceback (most recent call last):
  File "<pysHELL#12>", line 1, in <module>
    print(next(itr))
StopIteration
```

Как видно из приведенного выше примера вызов функции *next(itr)* каждый раз возвращает следующий элемент из списка, а когда эти элементы заканчиваются, генерируется исключение *StopIteration*.

15.2 Создание собственных итераторов

Если нужно обойти элементы внутри объекта вашего собственного класса, необходимо построить свой итератор. Создадим класс, объект которого будет итератором, выдающим определенное количество единиц, которое пользователь задает при создании объекта. Такой класс будет содержать конструктор, принимающий на вход количество единиц и метод *__next__()*, без него экземпляры данного класса не будут итераторами.

```

class SimpleIterator:
    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

s_iter1 = SimpleIterator(3)
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))
print(next(s_iter1))

```

В нашем примере при четвертом вызове функции *next()* будет выброшено исключение *StopIteration*. Если мы хотим, чтобы с данным объектом можно было работать в цикле *for*, то в класс *SimpleIterator* нужно добавить метод *__iter__()*, который возвращает итератор, в данном случае этот метод должен возвращать *self*:

```

class SimpleIterator:
    def __iter__(self):
        return self

    def __init__(self, limit):
        self.limit = limit
        self.counter = 0

    def __next__(self):
        if self.counter < self.limit:
            self.counter += 1
            return 1
        else:
            raise StopIteration

```

```
s_iter2 = SimpleIterator(5)
for i in s_iter2:
    print(i)
```

15.3 Генераторы

Генераторы позволяют значительно упростить работу по конструированию итераторов. В предыдущих примерах, для построения итератора и работы с ним, мы создавали отдельный класс. Генератор – это функция, которая будучи вызванной в функции *next()* возвращает следующий объект согласно алгоритму ее работы. Вместо ключевого слова *return* в генераторе используется *yield*. Проще всего работу генератор посмотреть на примере. Напишем функцию, которая генерирует необходимое нам количество единиц:

```
def simple_generator(val):
    while val > 0:
        val -= 1
        yield 1

gen_iter = simple_generator(5)
print(next(gen_iter))
print(next(gen_iter))
print(next(gen_iter))
print(next(gen_iter))
print(next(gen_iter))
print(next(gen_iter))
```

Данная функция будет работать точно также, как класс *SimpleIterator* из предыдущего примера.

Ключевым моментом для понимания работы генераторов является то, при вызове *yield* функция не прекращает свою работу, а “замораживается” до очередной итерации, запускаемой функцией *next()*. Если вы в своем генераторе, где-то используете ключевое слово *return*, то дойдя до этого места будет выброшено исключение *StopIteration*, а если после ключевого слова *return* поместить какую-либо информацию, то она будет добавлена к описанию *StopIteration*.

Урок 16. Установка пакетов в *Python*

16.1 Где взять отсутствующий пакет?

Необходимость в установке дополнительного пакета возникнет очень быстро, если вы решите поработать над задачей, за рамками базового функционала, который предоставляет *Python*. Например: работа с *web*, обработка изображений, криптография и т.п. В этом случае, необходимо узнать, какой пакет содержит функционал, который вам необходим, найти его, скачать, разместить в нужном каталоге и начать использовать. Все эти действия можно сделать вручную, но этот процесс поддается автоматизации. К тому же скачивать пакеты с неизвестных сайтов может быть довольно опасно.

К счастью для нас, в рамках экосистемы *Python*, все эти задачи решены. Существует так называемый *Python Package Index (PyPI)* – это репозиторий, открытый для всех *Python* разработчиков, в нем вы можете найти пакеты для решения практически любых задач. Там также есть возможность выкладывать свои пакеты. Для скачивания и установки используется специальная утилита, которая называется *pip*.

16.2 Менеджер пакетов в *Python* – *pip*

Pip – это консольная утилита (без графического интерфейса). После того, как вы ее скачаете и установите, она пропишется в *PATH* и будет доступна для использования.

Эту утилиту можно запускать как самостоятельно:

```
> pip <аргументы>
```

так и через интерпретатор *Python*:

```
> python -m pip <аргументы>
```

Ключ *-m* означает, что мы хотим запустить модуль (в данном случае *pip*). Более подробно о том, как использовать *pip*, вы сможете прочитать ниже.

16.3 Установка *pip*

При развертывании современной версии *Python* (начиная с *Python 2.7.9* и *Python 3.4*), *pip* устанавливается автоматически. Но если, по какой-то причине, *pip* не установлен на вашем ПК, то сделать это можно вручную. Существует несколько способов.

Универсальный способ

Будем считать, что *Python* у вас уже установлен, теперь необходимо установить *pip*. Для того, чтобы это сделать, скачайте скрипт *get-pip.py*:

```
> curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py
```

и выполните его:

```
> python get-pip.py
```

При этом, вместе с *pip* будут установлены *setuptools* и *wheels*. *Setuptools* – это набор инструментов для построения пакетов *Python*. *Wheels* – это формат дистрибутива для пакета *Python*. Обсуждение этих составляющих выходит за рамки урока, поэтому мы не будем на них останавливаться.

Способ для Linux

Если вы используете *Linux*, то для установки *pip* можно воспользоваться имеющимся в вашем дистрибутиве пакетным менеджером. Ниже будут перечислены команды для ряда *Linux* систем, запускающие установку *pip* (будем рассматривать только *Python 3*, т.к. *Python 2* уже морально устарел, а его поддержка и развитие будут прекращены после 2020 года).

Fedora

Fedora 21:

```
> sudo yum install python3 python3-wheel
```

Fedora 22:

```
> sudo dnf install python3 python3-wheel
```

openSUSE

```
> sudo zypper install python3-pip python3-setuptools python3-wheel
```


Debian/Ubuntu

```
> sudo apt install python3-venv python3-pip
```

Arch Linux

```
> sudo pacman -S python-pip
```

16.4 Обновление *pip*

Если вы работаете с *Linux*, то для обновления *pip* запустите следующую команду:

```
> pip install -U pip
```

Для *Windows* команда будет следующей:

```
> python -m pip install -U pip
```

16.5 Использование *pip*

Далее рассмотрим основные варианты использования *pip*: установка пакетов, удаление и обновление пакетов.

Установка пакета

Pip позволяет установить самую последнюю версию пакета, конкретную версию или воспользоваться логическим выражением, через которое можно определить, что вам, например, нужна версия не ниже указанной. Также есть поддержка установки пакетов из репозитория. Рассмотрим, как использовать эти варианты.

Установка последней версии пакета:

```
> pip install ProjectName
```

Установка определенной версии:

```
> pip install ProjectName==3.2
```

Установка пакета с версией не ниже 3.1:

```
> pip install ProjectName>=3.1
```

Установка Python пакета из git репозитория:

```
> pip install -e git+https://gitrepo.com/ProjectName.git
```

Установка из альтернативного индекса:

```
> pip install --index-url http://pypi.org/ ProjectName
```

Установка пакета из локальной директории:

```
> pip install ./dist/ProjectName.tar.gz
```

Удаление пакета

Для того, чтобы удалить пакет воспользуйтесь командой

```
> pip uninstall ProjectName
```

Обновление пакетов

Для обновления пакета используйте ключ *–upgrade*:

```
> pip install --upgrade ProjectName
```

Просмотр установленных пакетов

Для вывода списка всех установленных пакетов применяется команда *pip list*:

```
> pip list
```

Если вы хотите получить более подробную информацию о конкретном пакете, то используйте аргумент *show*:

```
> pip show ProjectName
```

Поиск пакета в репозитории

Если вы не знаете точное название пакета, или хотите посмотреть на пакеты, содержащие конкретное слово, то вы можете это сделать, используя аргумент *search*:

```
> pip search "test"
```

Урок 17. Виртуальные окружения

17.1 Что такое виртуальное окружение и зачем оно нужно?

При разработке *Python*-приложений или использовании решений на *Python*, созданных другими разработчиками, может возникнуть ряд проблем, связанных с использованием библиотек различных версий. Рассмотрим их более подробно.

Во-первых: различные приложения могут использовать одну и ту же библиотеку, но при этом требуемые версии могут отличаться.

Во-вторых: может возникнуть необходимость в том, чтобы запретить вносить изменения в приложение на уровне библиотек, т.е. вы установили приложение и хотите, чтобы оно работало независимо от того обновляются у вас библиотеки или нет. Как вы понимаете, если оно будет использовать библиотеки из глобального хранилища (*/usr/lib/pythonXX/site-packages*), то, со временем, могут возникнуть проблемы.

В-третьих: у вас просто может не быть доступа к каталогу */usr/lib/pythonXX/site-packages*.

Для решения данных вопросов используется подход, основанный на построении виртуальных окружений – своего рода песочниц, в рамках которых запускается приложение со своими библиотеками, обновление и изменение которых не затронет другие приложения, использующие те же библиотеки.

17.2 ПО позволяющее создавать виртуальное окружение в *Python*

Программное обеспечение, которое позволяет создавать виртуальные окружения в *Python* можно разделить на те, что входят в стандартную библиотеку *Python* и не входят в нее. Сделаем краткий обзор доступных инструментов.

Начнем с инструментов, которые входят в *PyPI*. *PyPI* – это *Python Package Index* (*PyPI*) – [репозиторий пакетов](#) *Python*, доступный для любого разработчика и пользователя *Python*.

virtualenv

Это, наверное, одни из самых популярных инструментов, позволяющих создавать виртуальные окружения. Он прост в установке и использовании. В сети довольно много руководств по *virtualenv*, самые интересные, на наш взгляд, будут собраны в конце урока в разделе “Полезные ссылки”. В общем, этот инструмент нужно обязательно освоить, как минимум, потому что описание развертывания и использования многих систем, созданных с использованием *Python*, включает в себя процесс создания виртуального окружения с помощью *virtualenv*.

pyenv

Инструмент для изоляции версий *Python*. Чаще всего применяется, когда на одной машине вам нужно иметь несколько версий интерпретатора для тестирования на них разрабатываемого вами ПО.

virtualenvwrapper

Virtualenvwrapper – это обертка для *virtualenv*, позволяющая хранить все изолированные окружения в одном месте, создавать их, копировать и удалять. Предоставляет удобный способ переключения между окружениями и возможность расширять функционал за счет *plug-in*’ов.

Существуют ещё инструменты и *plug-in*’ы, выполняющие работу по изоляции частей системы *Python*, но мы их не будем рассматривать.

Инструменты, входящие в стандартную библиотеку *Python*.

venv

Этот модуль появился в *Python 3* не может быть использован для решения задачи изоляции в *Python 2*. По своему функционалу очень похож на *virtualenv*. Если вы работаете с третьим *Python*, то можете смело его использовать.

17.3 virtualenv

Будем рассматривать работу с *virtualenv* в рамках операционной системы *Linux*. Для *Windows* все будет очень похоже, за исключением моментов, связанных со спецификой этой ОС: названия и расположение каталогов, запуск скриптов оболочки и т.п.

17.3.1 Установка virtualenv

Virtualenv можно установить с использованием менеджера *pip*, либо скачать исходные коды проекта и установить приложение вручную.

Установка с использованием *pip*.

Для установки *virtualenv* откройте консоль и введите следующую команду:

```
> pip install virtualenv
```

Установка из исходного кода проекта.

В этом случае, вам нужно будет выполнить чуть большее количество действий.

Введите в консоли следующий набор команд:

```
> curl -O https://pypi.python.org/packages/source/v/virtualenv/virtualenv-  
X.X.tar.gz  
> tar xvfz virtualenv-X.X.tar.gz  
> cd virtualenv-X.X  
> [sudo] python setup.py install
```

X.X – это версия приложения, ее вам нужно знать заранее.

Если использовать ключевое слово *sudo*, инструмент будет установлен глобально, в противном случае – локально.

Мы рекомендуем вам использовать *pip* для установки *virtualenv*.

17.3.2 Создание виртуального окружения

Виртуальное окружение создается следующей командой:

```
> virtualenv PRG1
```

PRG1 в данном случае – это имя окружения.

После выполнения данной команды, в текущем каталоге будет создан новый каталог с именем *PRG1*. Разберем более подробно его содержимое.

PRG1/bin/ – содержит скрипты для активации/деактивации окружения, интерпретатор Python, используемый в рамках данного окружения, менеджер *pip* и ещё несколько инструментов, обеспечивающих работу с пакетами Python. В Windows, это каталог *PRG1\Scripts*.

PRG1/include/ и *PRG1/lib/* – каталоги, содержащие библиотечные файлы окружения. Новые пакеты будут установлены в каталог *PRG1/lib/pythonX.X/site-packages/*.

17.3.3 Активация виртуального окружения

Для активации виртуального окружения воспользуйтесь командой (для *Linux*):

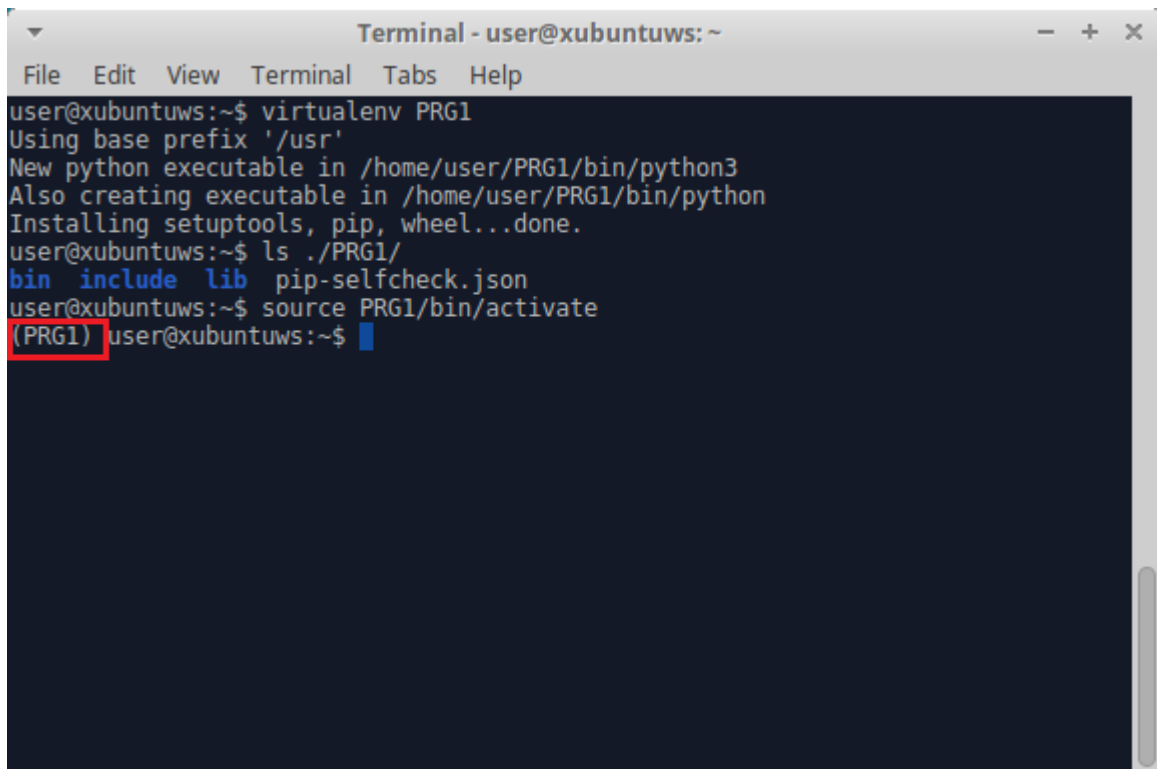
```
> source PRG1/bin/activate
```

для *Windows* команда будет выглядеть так:

```
> PRG1\Scripts\activate.bat
```

Команда *source* выполняет *bash*-скрипт без запуска второго *bash*-процесса.

Если команда выполнена успешно, то вы увидите, что перед приглашением в командной строке появилась дополнительная надпись, совпадающая с именем виртуального окружения.

A screenshot of a terminal window titled "Terminal - user@xubuntuws: ~". The terminal shows the following commands and output:
1. `user@xubuntuws:~$ virtualenv PRG1`
Output: `Using base prefix '/usr'`
`New python executable in /home/user/PRG1/bin/python3`
`Also creating executable in /home/user/PRG1/bin/python`
`Installing setuptools, pip, wheel...done.`
2. `user@xubuntuws:~$ ls ./PRG1/`
Output: `bin include lib pip-selfcheck.json`
3. `user@xubuntuws:~$ source PRG1/bin/activate`
Output: `(PRG1) user@xubuntuws:~$`
The prompt `(PRG1)` is highlighted with a red box, indicating the successful activation of the virtual environment.

При этом в переменную окружения *PATH*, в самое начало, будет добавлен путь до директории *bin*, созданного каталога *PRG1*.

Если вы создадите виртуальное окружение с ключем *--system-site-packages*:

```
> virtualenv --system-site-packages PRG1
```

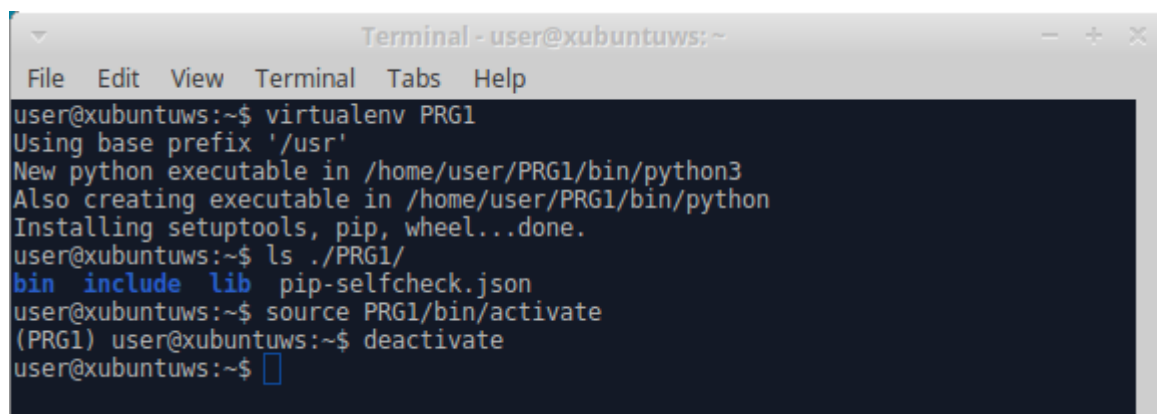
то в рамках окружения *PRG1* вы будите иметь доступ к глобальному хранилищу пакетов:

- в *Linux*: `/usr/lib/pythonX.X/site-packages`
- в *Windows*: `\PythonXX\Lib\site-packages`

17.3.4 Деактивация виртуального окружения

Для деактивации виртуального окружения (выхода из него), введите команду *deactivate* для *Linux* или *deactivate.bat*, если вы работаете в *Windows*.

```
> deactivate
```



```
Terminal - user@xubuntuws:~
File Edit View Terminal Tabs Help
user@xubuntuws:~$ virtualenv PRG1
Using base prefix '/usr'
New python executable in /home/user/PRG1/bin/python3
Also creating executable in /home/user/PRG1/bin/python
Installing setuptools, pip, wheel...done.
user@xubuntuws:~$ ls ./PRG1/
bin include lib pip-selfcheck.json
user@xubuntuws:~$ source PRG1/bin/activate
(PRG1) user@xubuntuws:~$ deactivate
user@xubuntuws:~$
```

17.4 venv

Устанавливать *venv* не нужно, т.к. он входит в стандартную библиотеку *Python*. Т.е. если вы установили себе *Python*, то *venv* у вас уже есть. Помните, что *venv* работает только в *Python 3*!

17.4.1 Создание виртуального окружения

Для создания виртуального окружения с именем *PRG2* с помощью *venv* выполните следующую команду:

```
> python -m venv PRG2
```

В результате будет создан каталог *PRG2* со структурой похожей на ту, что была описана для *virtualenv*. Функциональное назначение каталогов тоже самое.

17.4.2 Активация виртуального окружения

Активация виртуального окружения в *Linux* выполняется командой:

```
> source PRG2/bin/activate
```

в *Windows*:

```
> PRG2\Scripts\activate.bat
```

17.4.3 Деактивация виртуального окружения

Деактивация выполняется командой *deactivate* (работает как в *Windows*, так и в Linux)

```
> deactivate
```

17.5 Полезные ссылки

Ниже приведен список полезных ссылок, для того, чтобы более глубоко изучить тему создания виртуальных окружений в *Python*.

Официальная документация

[Документация по *virtualenv*](#)

[Документация по *virtualenvwrapper*](#)

[Документация по *venv*](#)

Статьи

[*Python*. Строим виртуальное окружение с помощью *virtualenv*](#)

[Памятка по *virtualenv* и изолированным проектам на *Python*](#)

Урок 18. Аннотация типов в *Python*

18.1 Зачем нужны аннотации?

Для начала ответим на вопрос: зачем нужны аннотации в *Python*? Если кратко, то ответ будет таким: для того чтобы повысить информативность исходного кода, и иметь возможность с помощью специальных инструментов производить его анализ. Одной из наиболее востребованных, в этом смысле, тем является контроль типов переменных. Несмотря на то, что *Python* – это язык с динамической типизацией, иногда возникает необходимость в контроле типов.

Согласно *PEP 3107* могут быть следующие варианты использования аннотаций:

- проверка типов;
- расширение функционала *IDE* в части предоставления информации об ожидаемых типах аргументов и типе возвращаемого значения у функций;
- перегрузка функций и работа с дженериками;
- взаимодействие с другими языками;
- использование в предикатных логических функциях;
- маппинг запросов в базах данных;
- маршалинг параметров в *RPC* (удаленный вызов процедур).

18.2 Контроль типов в *Python*

Рассмотрим небольшую демонстрацию того, как решался вопрос контроля типов в *Python* без использования аннотаций. Один из возможных вариантов (наверное самый логичный) решения данной задачи – это использование комментариев, составленных определенным образом.

Выглядит это так:

```
name = "John" # type: str
```

Мы создали переменную с именем *name* и предполагаем, что ее тип – *str*. Естественно, для самого интерпретатора *Python* это не имеет значения, мы, без труда можем продолжить нашу мини программу таким образом:

```
name = "John" # type: str
print(name)
```

```
name = 10
print(name)
```

И это будет корректно с точки зрения *Python*. Но если необходимо проконтролировать, что переменной *name* будут присваиваться значения только строкового типа, мы должны: во-первых указать в комментарии о нашем намерении – это мы сделали, во-вторых использовать специальный инструмент, который выполнит соответствующую проверку. Таким инструментом является *mypy*.

Установить его можно с помощью *pip*:

```
python -m pip install mypy
```

Если мы сохраним приведенный выше код в файле *type_tester.py* и выполним следующую команду:

```
python -m mypy test_type.py
```

Получим такое сообщение:

```
test_type.py:3: error: Incompatible types in assignment
(expression has type "int", variable has type "str")
```

Оно говорит о том, что обнаружено несоответствие типов в операции присваивание: переменная имеет тип *"str"*, а ей присвоено значение типа *"int"*.

18.3 Обзор *PEP*'ов регламентирующий работу с аннотациями

Начнем наше введение в тему аннотаций в *Python* с краткого обзора четырех ключевых документов:

[PEP 3107 — Function Annotations](#)

[PEP 484 — Type Hints](#)

[PEP 526 — Syntax for Variable Annotations](#)

[PEP 563 — Postponed Evaluation of Annotations](#)

Первый из них *PEP 3107 — Function Annotations*, является исторически первым из перечисленных выше документов. В нем описывается синтаксис использования аннотаций в функциях *Python*. Важным является то, что аннотации не имеют никакого семантического значения для интерпретатора *Python* и

предназначены только для анализа сторонними приложениями. Аннотировать можно аргументы функции и возвращаемое ей значение.

Следующий документ – *PEP 484 — Type Hints*. В нем представлены рекомендации по использованию аннотаций типов. Аннотация типов упрощает статический анализ кода, рефакторинг, контроль типов в рантайме и кодогенерацию, использующую информацию о типах. В рамках данного документа, определены следующие варианты работы с аннотациями: использование аннотаций в функциях согласно *PEP 3107*, аннотация типов переменных через комментарии в формате `# type: type_name` и использование *stub*-файлов.

В *PEP 526 — Syntax for Variable Annotations* приводится описание синтаксиса для аннотации типов переменных (базируется на *PEP 484*), использующего языковые конструкции, встроенные в *Python*.

PEP 563 — Postponed Evaluation of Annotations. Данный *PEP* вступил в силу с выходом *Python 3.7*. У подхода работы с аннотация до этого *PEP*’а был ряд проблем связанных с тем, что определение типов переменных (в функциях, классах и т.п.) происходит во время импорта модуля, и может сложится такая ситуация, что тип переменной объявлен, но информации об этом типе ещё нет, в таком случае тип указывают в виде строки – в кавычках. В *PEP 563* предлагается использовать отложенную обработку аннотаций, это позволяет определять переменные до получения информации об их типах и ускоряет выполнение программы, т. к. при загрузке модулей не будет тратиться время на проверку типов – это будет сделано перед работой с переменными.

Теперь более подробно остановимся на использовании аннотаций, опираясь на перечисленные выше *PEP*’ы.

18.4 Использование аннотаций в функциях

18.4.1 Указание типов аргументов и возвращаемого значения

В функциях мы можем аннотировать аргументы и возвращаемое значение. Выглядеть это может так:

```
def repeater(s: str, n: int) -> str:
    return s * n
```

Аннотация для аргумента определяется через двоеточие после его имени:

```
имя_аргумента: аннотация
```

Аннотация, определяющая тип возвращаемого функцией значения, указывается после ее имени с использованием символов ->:

```
def имя_функции() -> тип
```

Для *lambda*-функций аннотации не поддерживаются.

18.4.2 Доступ к аннотациям функции

Доступ к использованным в функции аннотациям можно получить через атрибут `__annotations__`, в котором аннотации представлены в виде словаря, где ключами являются атрибуты, а значениями – аннотации. Возвращаемое функцией значение хранится в записи с ключом *return*.

Содержимое `repeater.__annotations__`:

```
{'n': int, 'return': str, 's': str}
```

18.5 Аннотация переменных

18.5.1 Создание аннотированных переменных

Можно использовать один из трех способов создания аннотированных переменных:

```
var = value # type: annotation  
var: annotation; var = value  
var: annotation = value
```

Рассмотрим это на примере работы со строковой переменной с именем `name`:

```
name = 'John' # type: str  
name: str; name = 'John'  
name: str = 'John'
```

Приведем еще несколько примеров:

```
from typing import List, Tuple

# список
scores: List[int] = [1]

# кортеж
pack: Tuple[int, int, int] = (1, 2, 3)

# логическая переменная
flag: bool
flag = True

# класс
class Point:
    x: int
    y: int

    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y
```

18.5.2 Контроль типов с использованием аннотаций

Для проверки можно использовать уже знакомый нам инструмент *туру*.
Напишем вот такой код:

```
a: int = 10
b: int = 15

def sq_sum(v1: int, v2: int) -> int:
    return v1 ** 2 + v2 ** 2

print(sq_sum(a, b))
```

Сохраним его в файле с именем *work.py* и запустим *туру* для анализа:

```
> python -m mypy work.py
```

Если не указывать дополнительные ключи, то окно консоли будет чистым, т. к. *туру* не найдет никаких ошибок в вашем коде.

Но если заменить первую строку:

```
a: int = 10
```

на такую:

```
a: int = 10.3
```

и вновь запустить *туру*, то увидим вот такое сообщение:

```
work.py:7: error: Argument 1 to "sq_sum" has incompatible type "float"; expected "int"
```

При этом, естественно, код будет выполняться без всяких проблем, потому что интерпретатор *Python* в данном случае не обращает внимание на аннотации.

18.6 Отложенная проверка аннотаций

До выхода *Python* 3.7 определение типов в аннотациях происходило во время импорта модуля, что приводило к проблеме. Например, если выполнить следующий код:

```
class Rectangle:
    def __init__(self, height: int, width: int, color: Color) -> None:
        self.height = height
        self.width = width
        self.color = color
```

То возникнет ошибка *NameError: name 'Color' is not defined*. Она связана с тем, что переменная *color* имеет тип *Color*, который пока ещё не объявлен. Для решения этой проблемы, мы можем указать тип *Color* в кавычках, но это не очень удобно:

```
class Rectangle:
    def __init__(self, height: int, width: int, color: 'Color') -> None:
        self.height = height
        self.width = width
        self.color = color
```

Эту проблему можно решить воспользовавшись отложенной обработкой аннотаций из *Python 3.7*:

```
from __future__ import annotations

class Rectangle:
    def __init__(self, height: int, width: int, color: Color) -> None:
        self.height = height
        self.width = width
        self.color = color

class Color:
    def __init__(self, r: int, g: int, b: int) -> None:
        self.R = r
        self.G = g
        self.B = b

rect = Rectangle(1, 2, Color(255, 255, 255))
```

Можете проверить, что без строки *from __future__ import annotations* эта программа выполняться не будет.

Урок 19. Декораторы функций в Python

19.1 Что нужно знать о функциях в Python?

Для начала разберем два аспекта связанные с функциями в *Python*. Во-первых: функция – это объект специального типа, поэтому ее можно передавать в качестве аргумента другим функциям. Во-вторых: внутри функций можно создавать другие функции, вызывать их и возвращать как результат через *return*. Остановимся на этих моментах более подробно.

19.1.1 Функция как объект

В *Python* передача одной функции в качестве аргумента другой функции – это нормальная практика. Например, если у вас есть список целых чисел, и вы хотите на базе него получить другой список, элементами которого будут квадраты первого, то такую задачу можно решить в одну строчку:

```
>>> # исходный список
>>> a = [1, 2, 3, 4, 5]
>>> # функция, возводящая переданное ей число в квадрат
>>> sq = lambda x: x**2
>>> # проверим ее работу
>>> print(sq(5))
25
>>> # получаем список квадратов
>>> b = list(map(sq, a))
>>> print(b)
[1, 4, 9, 16, 25]
```

Здесь мы передали функции *map* в качестве первого аргумента функцию *sq*, которая будет применяться по очереди ко всем элементам списка *a*.

В *Python* функция – это специальный объект, у которого есть метод `__call__()`. Если мы создадим вот такой класс:

```
class DemoCall():
    def __call__(self):
        return 'Hello!'
```


То объект такого класса можно вызывать как функцию:

```
>>> hello = DemoCall()
>>> hello()
'Hello!'
```

19.1.2 Функция внутри функции

Вторым важным свойством функции, для понимания темы декораторов, является то, что их можно создавать, вызывать и возвращать из других функций. На этом построена идея [замыкания \(closures\)](#).

Например, создадим функцию, которая умножает два числа:

```
def mul(a):
    def helper(b):
        return a * b
    return helper
```

В ней реализованы два важных свойства которые нам понадобятся: внутри функции *mul()* создается еще одна функция, которая называется *helper()*; функция *mul()* возвращает функцию *helper* как результат работы.

Вызывается эта функция так:

```
>>>mul(4)(2)
8
```

Ее главная фишка состоит в том, что можно создавать на базе функции *mul()* свои кастомизированные функции. Например, создадим функцию “умножение на три”:

```
>>>three_mul = mul(3)
>>>three_mul(5)
15
```

Как вы можете видеть, мы построили функцию *three_mul*, которая умножает на три любое переданное ей число.

19.2 Что такое декоратор функции в *Python*?

Конструктивно декоратор в *Python* представляет собой некоторую функцию, аргументом которой является другая функция. Декоратор предназначен для добавления дополнительного функционала к данной функции без изменения содержимого последней.

19.2.1 Создание декоратора

Предположим у нас есть пара простых функций, вот они:

```
def first_test():  
    print('Test function 1')  
  
def second_test():  
    print('Test function 2')
```

Мы хотим дополнить их так, чтобы перед вызовом основного кода функции печаталась строка “*Run function*”, а по окончании – “*Stop function*”.

Сделать это можно двумя способами. Первый – это добавить указанные строки в начало в конец каждой функции, но это не очень удобно, т. к. если мы захотим убрать это, нам придется снова модифицировать тело функции. А если они написаны не нами, либо являются частью общей кодовой базы проекта, сделать это будет уже не так просто. Второй вариант – это воспользоваться знаниями из раздела “Что нужно знать о функциях в *Python*?”

Создадим вот такую функцию:

```
def simple_decore(fn):  
    def wrapper():  
        print('Run function')  
        fn()  
        print('Stop function')  
    return wrapper
```

Обернем наши функции в эту оболочку:

```
first_test_wrapped = simple_decore(first_test)  
second_test_wrapped = simple_decore(second_test)
```

Функции *first_test* и *second_test* остались неизменными:

```
>>> first_test()
Test function 1
>>> second_test()
Test function 2
```

Функции *first_test_wrapped* и *second_test_wrapped* обладают функционалом, которого мы добивались:

```
>>> first_test_wrapped()
Run function
Test function 1
Stop function
>>> first_test_wrapped()
Run function
Test function 1
Stop function
```

Если необходимо, чтобы так работали функций с именами *first_test* и *second_test*, то можно сделать так:

```
first_test = first_test_wrapped
second_test = second_test_wrapped
```

Проверим это:

```
>>> first_test()
Run function
Test function 1
Stop function

>>> second_test()
Run function
Test function 2
Stop function
```

То, что мы только что сделали и является реализацией идеи декоратора. Но вместо строк:

```
def first_test():
    print("Test function 1")
first_test_wrapped = simple_decore(first_test)
first_test = first_test_wrapped
```

Можно написать вот так:

```
@simple_decore
def first_test():
    print('Test function 1')
```

`@simple_decore` – это и есть декоратор функции.

19.2.2 Передача аргументов в функцию через декоратор

Если функция в своей работе требует наличие аргумента, то его можно передать через декоратор. Создадим декоратор, который принимает аргумент и выводит информацию о декорируемой функции и ее аргументе:

```
def param_transfer(fn):
    def wrapper(arg):
        print("Run function: " + str(fn.__name__) + "(), with param: " + str(arg))
        fn(arg)
    return wrapper
```

Для демонстрации ее работы создадим функцию, которая выводит квадратный корень переданного ей числа, в качестве декоратора, укажем только что созданный *param_transfer*:

```
@param_transfer
def print_sqrt(num):
    print(num**0.5)
```

Выполним эту функцию с аргументом 4:

```
>>> print_sqrt(4)
Run function: print_sqrt(), with param: 4
2.0
```

19.2.3 Декораторы для методов класса

Методы классов также можно объявлять с декоратором. Модифицируем декоратор *param_transfer*:

```
def method_decor(fn):
    def wrapper(self):
        print("Run method: " + str(fn.__name__))
        fn(self)
    return wrapper
```

Создадим класс для представления двумерного вектора (из математики). В этом классе определим метод *norm()*, который выводит модуль вектора:

```
class Vector():
    def __init__(self, px = 0, py = 0):
        self.px = px
        self.py = py
    @method_decor
    def norm(self):
        print((self.px**2 + self.py**2)**0.5)
```

Продemonстрируем работу этого метода:

```
>>> vc = Vector(px=10, py=5)
>>> vc.norm()
Run method: norm
11.180339887498949
```

19.2.4 Возврат результата работы функции через декоратор

Довольно часто, создаваемые функции возвращают какое-либо значение. Для того, чтобы его можно было возвращать через декоратор необходимо соответствующим образом построить внутреннюю функцию:

```
def decor_with_return(fn):
    def wrapper(*args, **kwargs):
        print("Run method: " + str(fn.__name__))
        return fn(*args, **kwargs)
    return wrapper
```

Этот декоратор можно использовать для оборачивания функций, которые принимают различные аргументы и возвращают значение:

```
@decor_with_return  
def calc_sqrt(val):  
    return val**0.5
```

Выполним функцию *calc_sqrt* с параметром 16:

```
>>> tmp = calc_sqrt(16)  
Run method: calc_sqrt  
>>> print(tmp)  
4.0
```

Урок 20. Объектная модель в *Python*

20.1 Что такое объектная модель *Python*?

Самым лаконичным ответом на вопрос: “что такое объектная модель *Python*?”, будет цитата из википедии, которую также приводит в своей книге Лучано Рамальо: *the properties of objects in general in a specific computer programming language, technology, notation or methodology that uses them*. Что можно перевести как “общие свойства объектов в конкретном языке программирования, технологии, нотации или методологии”.

Красота объектной модели в *Python* состоит в том, что если на уровне класса реализовать определенный функционал, то с объектами этого класса можно будет работать используя “**питонический стиль**” – стиль написания программного кода на *Python*, который отличается большей выразительностью, читаемостью и понятностью. Такой стиль называют идиоматический – использующий наиболее естественные для данного языка решения.

Для того, чтобы понять о чем идет речь, рассмотрим пример: списковое включение (*list comprehensions*).

Напишем код, который создает список из квадратов первых пять целых чисел:

```
sq = []  
for i in range(1, 6):  
    sq.append(i**2)
```

Теперь напишем этот же код с использованием спискового включения:

```
sq = [i**2 for i in range(1, 6)]
```

Не правда ли, второй вариант более лаконичный, красивый и понятный? (Хотя мы не отрицаем, что возможно есть люди, которым больше нравится первый вариант).

Всю мощь объектной модели *Python* можно осознать через изучение назначения и возможностей **специальных методов**. Это методы, которые **вызываются интерпретатором для выполнения различных операций над объектами**. В своем коде, программисты, как правило, напрямую не вызывают эти методы. Объектная модель – это своеобразный *API*, который вы можете

реализовать, чтобы ваши объекты можно было использовать максимально естественно для заданной экосистемы (в нашем случае – *Python*).

Синонимом “объектной модели” является “модель данных”. В *Python* документации есть целый раздел, посвященный этой теме: <https://docs.python.org/3/reference/datamodel.html>.

20.2 Специальные методы

Рассмотрим некоторые из специальных методов (самые интересные, как нам показалось). Более подробную информацию о них вы можете найти в [соответствующей документации](#) по *Python*.

`__repr__()`

Строковое представление объекта, позволяющее восстановить объект. Например, если вы пишете игру и у вас есть объект “Персонаж”, которому при его создании присваивается имя и пол, то метод `__repr__()` может вернуть следующую строку “*Person(“John”, “man”)*”. Эта информация позволит вам заново построить объект.

`__str__()`

Строковое представление объекта, используемое в функциях `format()` и `print()`. В отличие от `__repr__()`, в ней, при реализации, мы стараемся создать наиболее информативное представление объекта.

`__bytes__()`

Метод возвращает объект типа *bytes* – представление текущего объекта в виде набора байт.

`__bool__()`

Возвращает *True* или *False*.

`__call__()`

Позволяет использовать объект как функцию, т.е. его можно вызывать.

`__len__()`

Чаще всего реализуется в коллекциях и сходными с ними по логике работы типами, которые позволяют хранить наборы данных. Для списка (*list*) `__len__()`

возвращает количество элементов в списке, для строки – количество символов в строке. Вызывается функцией *len()*, встроенной в язык *Python*.

Функции сравнения

Функция	Операция	Функция	Операция	Функция	Операция
<code>__lt__()</code>	<	<code>__le__()</code>	<=	<code>__eq__()</code>	==
<code>__ne__()</code>	!=	<code>__gt__()</code>	>	<code>__ge__()</code>	>=

Эти функции используются, если объекты можно сравнивать между собой, как числа. Синтаксис работы с ними выглядит следующим образом: `x.__lt__(y)`, что соответствует записи `x < y`.

Функции, эмулирующие числовые типы

Арифметические операции

Функция	Операция	Описание / пример
<code>__add__()</code>	+	Сложение
<code>__sub__()</code>	–	Вычитание
<code>__mul__()</code>	*	Умножение
<code>__truediv__()</code>	/	Деление
<code>__floordiv__()</code>	//	Целая часть от деления
<code>__mod__()</code>	%	Остаток от деления
<code>__pow__()</code>	**	Возведение в степень

Помимо указанных в таблице, в объектной модели *Python*, есть методы, позволяющие выполнять инверсные арифметические операции (случай, когда объекты, над которыми производится действие меняются местами), имена таких методов совпадают с перечисленными выше с добавлением символа *r* перед именем (примеры: `__radd__()`, `__rsub__()`). Методы для арифметических операции присваивания, таких как `*=`, `+=` и т.п., имеют имена сходные с табличными с добавлением символа *i* перед именем (примеры: `__iadd__()`, `__isub__()`).

Унарные операции

Функция	Операция	Описание / пример
<code>__neg__()</code>	<code>-</code>	Отрицательное значение
<code>__pos__()</code>	<code>+</code>	Положительное значение
<code>__abs__()</code>	<code>abs()</code>	Модуль числа

Битовые операции

Функция	Операция	Описание / пример
<code>__lshift__()</code>	<code><<</code>	Поразрядный сдвиг влево
<code>__rshift__()</code>	<code>>></code>	Поразрядный сдвиг вправо
<code>__and__()</code>	<code>&</code>	Логическое И
<code>__or__()</code>	<code> </code>	Логическое ИЛИ
<code>__xor__()</code>	<code>^</code>	Исключающее ИЛИ
<code>__invert__()</code>	<code>~</code>	Инверсия

Для битовых операций, также как для арифметических, предполагается возможность реализовывать инверсные операции и битовые операции присваивания.

20.3 Примеры

Для иллюстрации использования возможностей объектной модели *Python* приведем канонический пример с классом, в рамках которого реализуется поддержка операций над векторами:

```
class Vector():
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def __add__(self, v):
        return Vector(self.x + v.x, self.y + v.y)
    def __sub__(self, v):
        return Vector(self.x - v.x, self.y - v.y)
    def __repr__(self):
        return "Vector({}, {})".format(self.x, self.y)
    def __str__(self):
        return "({}, {})".format(self.x, self.y)
    def __abs__(self):
        return (self.x**2 + self.y**2)**0.5
```

```

if __name__ == "__main__":
    v1 = Vector(3, 4)
    print("vector v1 = {}".format(v1))
    v2 = Vector(5, 7)
    print("vector v2 = {}".format(v2))
    print("v1 + v2 = {}".format(v1 + v2))
    print("v2 - v1 = {}".format(v2 - v1))
    print("|v1| = {}".format(abs(v1)))

```

Запустив эту программу, вы увидите следующее:

```

> python object_model.py
vector v1 = (3, 4)
vector v2 = (5, 7)
v1 + v2 = (8, 11)
v2 - v1 = (2, 3)
|v1| = 5.0

```

Операции сложения и вычитания векторов реализованы в методах `__add__()` и `__sub__()`. Модуль вектора – в `__abs__()`. Функция `print()`, если ей передать объект типа `Vector`, в первую очередь вызовет метод `__str__()`, если этого метода не будет, то вызовется `__repr__()`.

Для эксперимента можете удалить метод `__str__()` и получите следующий результат:

```

> python object_model.py
vector v1 = Vector(3, 4)
vector v2 = Vector(5, 7)
v1 + v2 = Vector(8, 11)
v2 - v1 = Vector(2, 3)
|v1| = 5.0

```

Вот ещё один пример, который демонстрирует работу с операторами сравнения:

```
class UserLevel():
    us_dict = {"Admin": 4, "Operator": 3, "Dispatcher": 2, "Guest": 1}

    def __init__(self, user_level):
        if user_level in self.us_dict:
            self.u_level = user_level
        else:
            print("Wrong user type!")
            self.u_level = "Guest"

    def diff(self, level1, level2):
        return self.us_dict[level1] - self.us_dict[level2]

    def __lt__(self, user):
        return True if self.diff(self.u_level, user.u_level) < 0 else False

    def __le__(self, user):
        return True if self.diff(self.u_level, user.u_level) <= 0 else False

    def __eq__(self, user):
        return True if self.diff(self.u_level, user.u_level) == 0 else False

    def __ge__(self, user):
        return True if self.diff(self.u_level, user.u_level) >= 0 else False

    def __gt__(self, user):
        return True if self.diff(self.u_level, user.u_level) > 0 else False

    def __repr__(self):
        return "UserLevel({})".format(self.u_level)

    def __str__(self):
        return self.u_level
```

```

if __name__ == "__main__":
    need_level = UserLevel("Operator")
    print("need level: {}".format(need_level))
    user_level = UserLevel("Dispatcher")
    print("user: {}".format(user_level))
    if user_level >= need_level:
        print("Access enabled!")
    else:
        print("Access denied!")

```

В этом примере реализован класс *UserLevel*, который используется для хранения уровня доступа. Объекты этого типа можно сравнивать между собой, используя классические операции сравнения, поддержка этого функционала реализуется через методы, перечисленные в блоке **Битовые операции** раздела **Специальные методы** этого урока.

Что еще почитать на эту тему?

На эту тему интересно написано у Лучано Рамальо в книге *“Python. К вершинам мастерства”*. Дополнительно рекомендуем обратить внимание на *“Python in a Nutshell”* Алекса Мартелли и *“Python Essential Reference”* Дэвида Бизли.