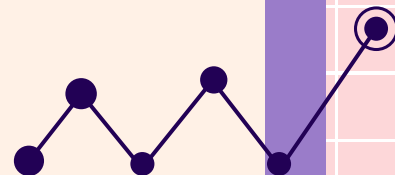


>(□□0△//△/□□)<

# Программирование в лингвистике

Модули, установка модулей, pip



# Что такое модули?

**Модуль** - единица организации программ **наивысшего уровня**, которая **упаковывает программный код** для многократного использования и предоставляет **изолированное пространство имен**, что сводит к минимуму конфликты имен объектов внутри программ.

Обобщая все изученное, стоит напомнить, что в мире Python все - это **объекты**. У большей части объектов есть свои **атрибуты**. Модуль - это **тоже объект** (очень высокого уровня), атрибуты модуля - это **все объекты**, **которые в нем находятся**, то есть, переменные, функции и классы.

```
import os
import collections
```

os.path	# Атрибут os - вложенный модуль
os.listdir()	# Атрибут os - функция
collections.Counter()	# Атрибут collections - класс

# Что такое модули?

Любой скрипт Python (.py-файл) – это **модуль**. Его можно подгрузить в другой скрипт и использовать там функции и классы, которые в нем содержатся. Среди предустановленных модулей `math`, `os`, `random`, `string`, `re`, `json`, `collections`, `sys`. Все это – **.py-файлы**, код из которых мы подгружаем и используем.

```
random.py 1 X
E: > Soft_installed > Python > Lib > random.py > ...
332     return istart + istep * self._randbelow(n)
333
334     def randint(self, a, b):
335         """Return random integer in range [a, b], including both end points.
336         """
337
338         return self.randrange(a, b+1)
339
340
341     ## ----- sequence methods -----
342
343     def choice(self, seq):
344         """Choose a random element from a non-empty sequence."""
345         # raises IndexError if seq is empty
346         return seq[self._randbelow(len(seq))]
347
```

Модуль random внутри

# Внешний модуль

Для пользователей Visual Studio Code:

hello.py

```
6 term > python > hello.py > ...  
1 def hello():  
2     print('hello')
```

```
import hello
```

```
hello.hello()
```

✓ 0.2s

hello

В отдельном скрипте .py мы пишем наш **пользовательский модуль hello**, в который добавляем функцию **hello**.

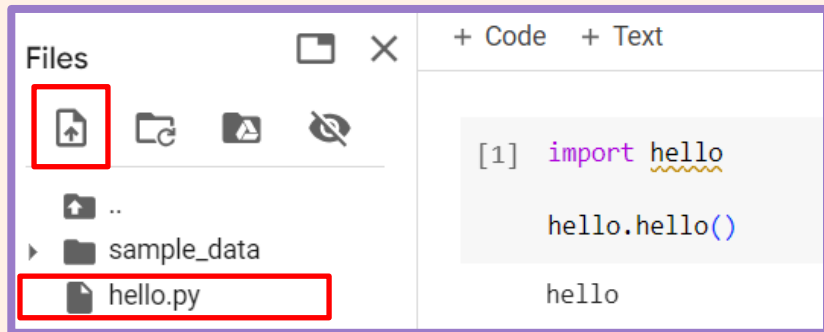
В новом скрипте .py или тетрадке .ipynb мы **импортируем наш модуль** (он должен находиться в той же папке, что и основной скрипт), а затем **используем функцию** из него.

# Внешний модуль

Для пользователей Google Colaboratory:

□ △ // } 00

Создаем **текстовый документ**, переименовываем его в **hello.py**.  
**В блокноте** пишем функцию hello и сохраняем файл.



С помощью **файлового менеджера Colab** загружаем наш модуль в сессию и **используем функцию** из него.



# Подгрузка модулей

```
import collections          # Импорт модуля
import collections as col   # Импорт модуля под названием col
from collections import Counter # Импорт одного класса из модуля
from collections import Counter as C # Импорт класса из модуля под именем C
from collections import *   # Импорт всего содержимого модуля
```

*Import \* использовать **не рекомендуется** – мы забываем неймспейс всеми именами из модуля.*

Как мы видим, есть два оператора подгрузки – **import** и **from x import y**.

В первом случае в момент, когда запускается оператор **import**, **скрипт импортируемого модуля выполняется**, чтобы в текущем скрипте появились **все определенные в нем объекты**. Имя этого модуля добавляется в **неймспейс** текущего скрипта, и все его объекты делают его **атрибутами**.

```
import os

os.listdir()
```

Здесь, после импорта **os**, в неймспейсе текущего скрипта появляется имя **os**, а у него, например, **os.listdir()**.

# Неймспейс

**Неймспейс** – пространство имен. Python делит все **имена переменных, функций и классов** внутри себя на определенные сегменты, чтобы они друг другу не мешались. Так, у текущего скрипта есть **свое пространство имен**, в котором живет все, что мы определили в этом скрипте, **+ импортированные имена модулей**. Что содержится в пространстве имен скрипта, можно посмотреть с помощью функции **dir()**.

dir без аргумента выведет все, что есть в неймспейсе **текущего скрипта**:

```
dir()
```

```
['collections',  
'hello',  
'os',      И Т.Д...
```

# Имена модулей

У любого скрипта есть магический атрибут `__name__`. Это то имя, под которым скрипт существует в текущем неймспейсе. Если мы этот скрипт запускаем, атрибуту `__name__` присваивается строка `'__main__'`. Модули, которые мы импортируем, будут иметь те имена, под которыми мы их подгружаем.

**Напоминание:** наш текущий скрипт тоже является модулем – его имя `'__main__'`.

Мы можем проверить имена наших модулей в текущем неймспейсе:

```
[3] print(hello.__name__) # Имя импортируемого скрипта
    hello

[4] print(__name__)      # Имя текущего скрипта
    __main__
```



# If `__name__ == '__main__'`

Итак, когда мы **импортируем какой-то модуль**, наш текущий скрипт делает следующее:

1. **Ищет модуль** с таким названием
2. **Выполняет код** модуля для создания объектов, которые в нем определены

Именно поэтому иногда в скриптах пишут **такую вещь**:

```
if __name__ == '__main__':  
    # Some debug commands
```

Это делается для того, чтобы код в модуле **не выполнялся**, если этот модуль **просто импортировали**. Например, вы написали свой собственный токенизатор и хотите его **тестировать**. Все функции для тестирования могут вызываться внутри `if __name__ == '__main__':`, а когда будете импортировать свой токенизатор, они вызываться **не будут**.

# Внешние модули Python

О том, где и как Python ищет **внешние модули**:

1. Сперва **в домашней папке скрипта**;
2. В каталогах **стандартной библиотеки**;
3. В **подкаталоге site-packages**, куда устанавливается все, что мы ставим через pip, conda или другие менеджеры пакетов.

Поэтому **не рекомендуется** называть свои скрипты **так же, как называются внешние модули**, которые вы собираетесь импортировать. Например, называть свой скрипт pandas не очень умно, если вам потом понадобится сам pandas!

```
import pandas
```

```
pandas.DataFrame()
```

⊗ 0.7s

-----  
**AttributeError**

Traceback (most recent call last)



# Установка внешних модулей

Если нам нужен модуль, которого нет в ванильном Python, то мы можем установить его **через менеджер пакетов pip**. Pip вызывается **в командной строке (терминале)**.

## Как открыть терминал?

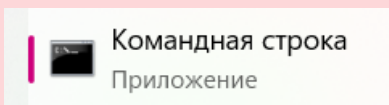
### Windows

Первый вариант:


1. Нажать горячие клавиши Win+R.
2. Написать в окне 'cmd'.

Второй вариант:


1. Открыть меню Пуск.
2. Ввести в поиске 'Командная строка' или 'cmd'.



### MacOS

Первый вариант: 

1. Нажать значок Launchpad в панели Dock.
2. Ввести в окне 'Терминал'.

Второй вариант: 

1. В окне Finder открыть папку 'Программы/Утилиты'.
2. Выбрать приложение 'Терминал'.



# Установка внешних модулей

Pip может вести себя **по-разному** с разными установками Python. Так, чтобы выполнять команды pip, Вам могут понадобиться **префиксы**.

pip **команда**

pip3 **команда**

python pip **команда**

py -m pip **команда**

python -m pip **команда**

Нужные команды **pip**:

pip list	Вывод установленных модулей.
pip install <b>модуль</b>	Установка модуля.
pip uninstall <b>модуль</b>	Удаление модуля.

```
C:\Users\user>pip install pandas
```

```
C:\Users\user>py -m pip install pandas
```

Хорошая практика использовать флаг **-m** при установке пакетов.



# Установка внешних модулей

Для пользователей **Google Colaboratory** все просто:

```
!pip install pandas
```

Pip вызывается с восклицательным знаком перед ним.

```
import pandas as pd  
  
df = pd.DataFrame()
```

# Anaconda

Продвинутые программисты используют приложение **Anaconda**, чтобы заводить **виртуальные среды** и устанавливать в них пакеты.

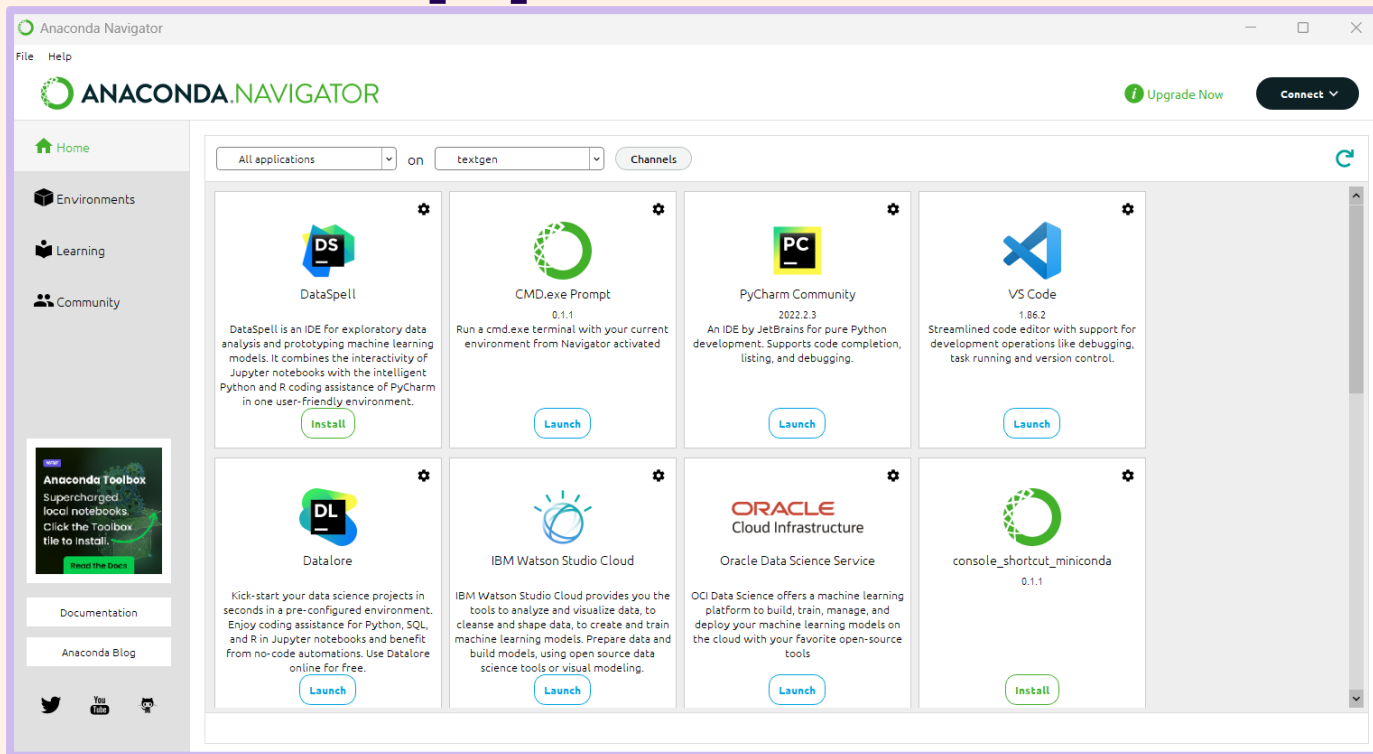
**Виртуальные среды** – это, проще говоря, **изолированные мини-Python'ы** нужной версии. Они нужны, чтобы предотвращать **конфликты** модулей **между собой** или с **исходной версией Python**.

Приложение Anaconda часто используют в паре с продвинутыми IDE вроде **PyCharm** или **Visual Studio Code**.

[Инструкция к установке](https://docs.anaconda.com/free/anaconda/install/windows/)

<https://docs.anaconda.com/free/anaconda/install/windows/>

# Интерфейс Anaconda



# PyPI.org

Для Python есть большой репозиторий [pypi.org](https://pypi.org), откуда pip берет и устанавливает библиотеки в `python/Lib/site-packages`.

На [pypi.org](https://pypi.org) можно посмотреть [доступные внешние модули](#), а также их документацию и инструкции по установке.

