Оглавление

1.1 Введение

1. МОДУЛИ

Модули - это фрагменты кода, написанные другими людьми для выполнения общих рутинных задач, таких как:

- генерация случайных чисел,
- выполнение числовых операций,
- обучение нейронных сетей и т.д.

Популярность языка Python обусловлена наличием обширного набора модулей для решения различных задач.

Чтобы использовать модуль, необходимо добавить в верхнюю часть вашего кода следующую строку:

```
import имя_модуля
```

Затем, для доступа к функции, необходимо использовать следующий синтаксис:

```
имя_модуля.название_функции
```

Приведем пример использования функции `randint` из модуля `random` для генерации случайных целых чисел:

```
import random

for i in range(5):

    value = random.randint(1, 6)

    print(value)
```

В **Python** существует три основных типа модулей:

1. Предустановленные вместе с **Python** - **стандартные библиотеки** (math, re, time, csv, json).
2. Устанавливаемые из внешних источников - **сторонние библиотеки** (requests, bs4, numpy, pandas, и др.).
3. Модули, которые вы пишете сами - ваши **собственные библиотеки**.

Первый тип называется стандартной библиотекой и включает множество

полезных модулей, таких как: `re`, `datetime`, `math`, `random`, `os`, `multiprocessing`, `subprocess`, `socket`, `email`, `json`, `doctest`, `unittest`, `pdb`, `argparse` и `sys`.

Стандартная библиотека может выполнять множество задач, включая синтаксический анализ строк, сериализацию данных, тестирование, отладку и манипуляции с датами, аргументами командной строки и многое другое!

2. ВИРТУАЛЬНОЕ ОКРУЖЕНИЕ В PYTHON

Виртуальное окружение - это изолированное пространство, предназначенное для управления зависимостями и пакетами Python для конкретного проекта. Оно позволяет избежать конфликтов между разными проектами, так как каждый проект может иметь свои собственные версии пакетов, независимые от системных или других проектов.

Зачем нужно виртуальное окружение?

1. **Изоляция проектов:** Виртуальное окружение позволяет вам изолировать зависимости и библиотеки для каждого конкретного проекта. Это важно, потому что различные проекты могут требовать разных версий библиотек, и использование общего глобального окружения может привести к конфликтам.

2. **Легкость управления зависимостями:** В виртуальном окружении вы можете легко устанавливать, обновлять и удалять пакеты, специфичные для вашего проекта, без воздействия на другие проекты или системные библиотеки.

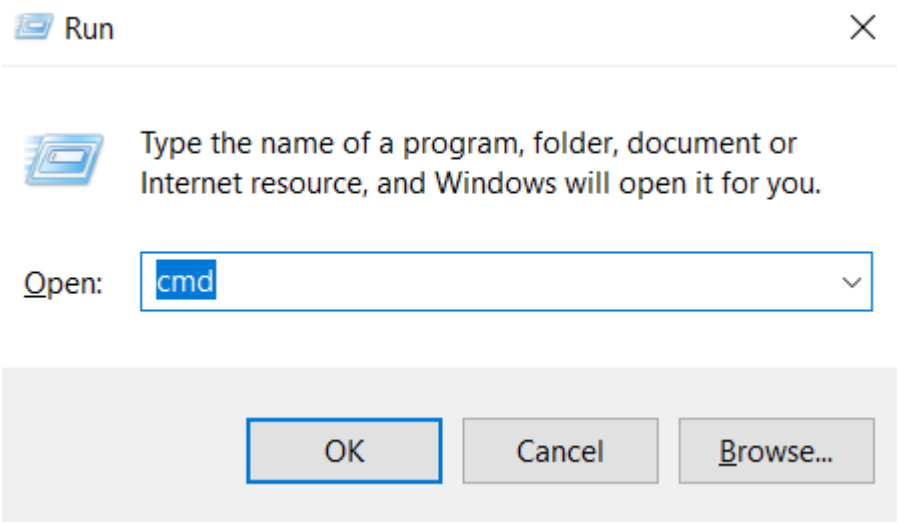
3. **Управление версиями Python:** Виртуальные окружения позволяют вам использовать разные версии интерпретатора Python для разных проектов. Это особенно полезно, когда вы работаете над старыми проектами, которые требуют устаревших версий Python.

Если вы работаете в **Google Colab**, понятие виртуального окружения не так важно, поскольку для каждого сеанса работы создается индивидуальное виртуальное окружение. Однако, значение виртуального окружения становится более значимым при работе на собственном компьютере. В этом случае рекомендуется создавать свое собственное виртуальное окружение для каждого проекта внутри каталога.

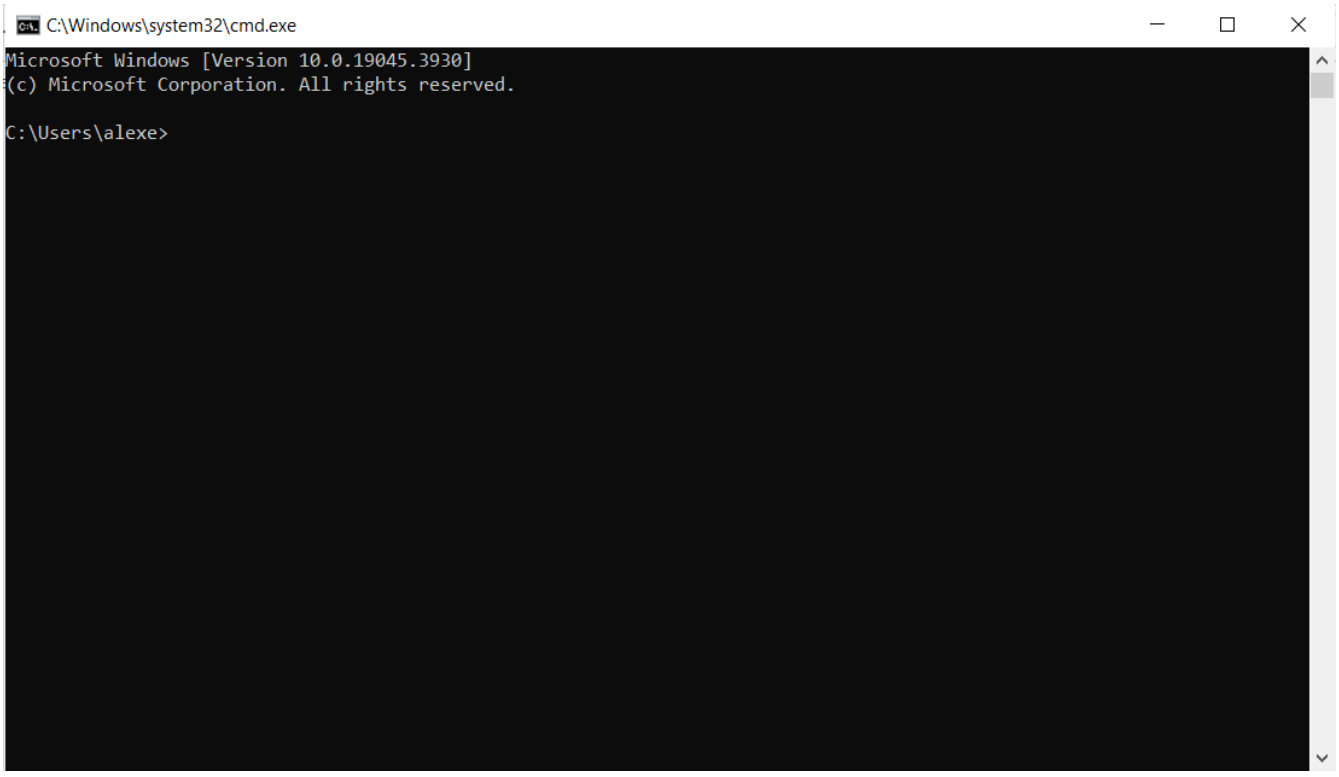
Для начала работы с Python на вашем компьютере необходимо установить сам язык Python, поскольку он не входит в стандартный набор программ Windows. Инструкции по установке Python легко найти на YouTube, например, вот [одно из видеоуроков об этом]

Как создать виртуальное окружение?

Если вы работаете в Windows, нажмите комбинацию клавиш `Win+R`, и в левом нижнем углу экрана откроется окно:



Введите команду `cmd` и нажмите **OK**. Перед вами откроется терминал:



Затем прейдите в директорию своего проекта с помощью команды `cd C:/Католог/вашего/проекта/`

После того как вы перейдете в каталог проекта вам необходимо выполнить следующую команду в командной строке терминала для создания виртуального окружения:

```
python3 -m venv имя_окружения
```

где `имя_окружения` - это имя, которое вы выбираете для вашего виртуального окружения. Рекомендуется использовать `venv` в качестве имени.

После создания виртуального окружения, его можно активировать с использованием команды, соответствующей вашей операционной системе:

Windows:

```
.\venv\Scripts\activate.bat
```

Linux/Mac:

```
source venv/bin/activate
```

`pydoc modules` - список встроенных библиотек

`pip list` - посмотреть список установленных пакетов

3 Пространства имен и уровни доступа

Пространство имен (Namespace) в Python - это концепция, которая определяет область видимости имен переменных в программе. Простыми словами, пространство имен - это место, где находятся имена переменных, функций, классов и других объектов в Python. Пространства имен позволяют организовывать код, управлять его областями видимости и избегать конфликтов имен.

Назначение пространства имен:

1.**Изоляция имен**: Пространства имен позволяют изолировать имена переменных, функций и классов от других частей программы. Это означает, что вы можете использовать одно имя для разных объектов в разных частях программы, и эти имена не будут конфликтовать друг с другом.

2.**Организация кода**: Пространства имен помогают организовывать код и делают его более читаемым и управляемым.

3.**Управление областью видимости**: Пространства имен определяют область видимости и доступности переменных и объектов в программе.

Примеры пространства имен в Python:

1.**Глобальное пространство имен**: Это пространство имен, которое доступно во всей программе. В нем находятся глобальные переменные, функции.

```
global_var = 10
```

```
def my_function():
    print(global_var)
```

Локальное пространство имен: Это пространство имен, ограниченное областью действия определенной функции или метода.

```
def my_function():
    local_var = 20
    print(local_var)
```

Пространство имен модуля: Это пространство имен, определенное внутри модуля. Все имена верхнего уровня в модуле являются частью пространства имен модуля.

```
# module.py
module_var = 30

def module_function():
    print(module_var)
```

Способы избегания перенасыщения пространства имен:

1.**Использование модулей**: Организуйте ваш код в модули. Модули позволяют разделить код на логические блоки и обеспечивают изоляцию пространства имен.

2.**Избегайте глобальных переменных**: Используйте локальные переменные в предпочтении глобальных, когда это возможно. Это помогает избежать загромождения глобального пространства имен.

3.**Использование пространства имен классов**: Определение методов и атрибутов в классах помогает избежать конфликтов имен в глобальном пространстве имен.

Уровни доступа:

1.**Public (Открытый)**: Переменные или функции, к которым можно получить доступ из любого места в программе.

2.**Protected (Защищенный)**: Переменные или функции, к которым можно получить доступ только из класса, где они определены, или из подклассов этого класса.

3.**Private (Приватный)**: Переменные или функции, к которым можно получить доступ только внутри класса, где они определены.

В Python уровни доступа обычно контролируются соглашениями между разработчиками, так как интерпретатор Python не имеет строгой поддержки модификаторов доступа, как в некоторых других языках программирования. Именно соглашения о том, какие имена следует считать "публичными", "защищенными" или "приватными", играют важную роль в организации кода и предотвращении

нежелательного доступа к данным и функциям. Эту тему мы изучим более подробно в следующем модуле, когда будем учится создавать классы.

Импорт стандартных модулей

Импорт стандартных модулей в Python является важной частью разработки приложений, так как стандартная библиотека Python предоставляет множество полезных инструментов для решения различных задач. Вот некоторые подробности о том, как происходит импорт стандартных модулей:

Импорт модулей

Импорт модулей в Python осуществляется с использованием ключевого слова `import`. Синтаксис следующий:

```
import имя_модуля
```

Например, для импорта модуля `math` (который предоставляет математические функции), используется следующий код:

```
import math
```

Чтобы посмотреть все функции содержащиеся в модуле `math` можно использовать функцию `dir, которая содержит в себе все функции в виде списка имен`

```
for fun in dir(math):
   print(fun)
```

Использование функций и переменных из модуля

После импорта модуля вы можете использовать его функции и переменные, обращаясь к ним через имя модуля, точку и имя функции или переменной. Пример:

```
import math

result = math.sqrt(25)  # используем функцию sqrt из модуля math
print(result)
```

Импорт определенных функций или переменных

Иногда бывает удобно импортировать только определенные функции или переменные из модуля, чтобы не загружать весь модуль. Это делается с

помощью следующего синтаксиса:

```
from имя_модуля import имя_функции, имя_переменной
```

Пример:

```
from math import sqrt, pi

result = sqrt(25)  # теперь мы можем использовать sqrt без 
префикса math
print(result)

print(pi)  # прямой доступ к переменной pi из модуля math
```

Импорт всего модуля с псевдонимом

Иногда модули имеют длинные имена, и их часто импортируют с использованием псевдонима (алиаса). Это делается с помощью ключевого слова `as`. Пример:

```
import very_long_module_name as vlmn

vlmn.some_function()
```

Импорт всего содержимого модуля (не рекомендуется)

Также возможен импорт всего содержимого модуля с использованием звездочки `*`. Однако такой способ не рекомендуется, так как он может внести в ваш код много неясности и конфликтов. Пример:

```
from math import *
```

Импорт модулей из подпакетов

Если у вас есть подпакет в стандартной библиотеке, вы можете импортировать модули из этого подпакета аналогичным образом, как вы импортируете модули. Например:

```
import os.path  # импорт модуля path из подпакета os
```

Импорт стандартных модулей в Python предоставляет множество возможностей для эффективной и чистой разработки. Хорошая практика - явно указывать, какие функции и переменные вы используете из каждого модуля, чтобы улучшить читаемость кода и избежать конфликтов имен.

Стандартные модули Python

Python предоставляет множество стандартных модулей, которые расширяют функциональность языка для различных задач. Вот некоторые из основных стандартных модулей Python:

1. **math**: Математические функции.
2. **random**: Генерация случайных чисел.
3. **datetime**: Работа с датой и временем.
4. **os**: Взаимодействие с операционной системой (например, работа с файлами и директориями).
5. **sys**: Взаимодействие с интерпретатором Python и системными настройками.
6. **json**: Работа с форматом данных JSON.
7. **urllib**: Работа с URL-адресами и выполнение HTTP-запросов.
8. **re**: Регулярные выражения.
9. **collections**: Дополнительные структуры данных (например, OrderedDict, Counter).
10. **itertools**: Инструменты для создания итераторов и операций с ними.
11. **sqlite3**: Взаимодействие с базой данных SQLite.
12. **pickle**: Сериализация и десериализация объектов Python.
13. **csv**: Работа с файлами формата CSV.
14. **email**: Работа с электронной почтой.
15. **socket**: Низкоуровневая работа с сетевыми сокетами.
16. **argparse**: Обработка аргументов командной строки.
17. **logging**: Система логирования.
18. **subprocess**: Запуск и управление внешними процессами.

Это лишь небольшой список. Существует множество других модулей, предоставляющих различные возможности для разработки в различных областях.

Список некоторых математических функций, предоставляемых модулем `math` в Python:

1. **Арифметические функции:**

- `math.ceil(x)`: Возвращает наименьшее целое число, не меньшее x.
- `math.floor(x)`: Возвращает наибольшее целое число, не большее x.
- `math.trunc(x)`: Возвращает целую часть числа x.

2. **Экспоненциальные и логарифмические функции:**

•`math.exp(x)`: Возвращает e в степени x.

•`math.log(x[, base])`: Возвращает натуральный логарифм x. Если указан параметр base, возвращает логарифм x по указанному основанию.

•`math.log10(x)`: Возвращает десятичный логарифм x.

3.**Тригонометрические функции:**

•`math.sin(x)`, `math.cos(x)`, `math.tan(x)`: Синус, косинус и тангенс угла x (в радианах).

•`math.asin(x)`, `math.acos(x)`, `math.atan(x)`: Обратные тригонометрические функции.

4.**Гиперболические функции:**

•`math.sinh(x)`, `math.cosh(x)`, `math.tanh(x)`: Гиперболический синус, косинус и тангенс.

5.**Степенные и корневые функции:**

•`math.sqrt(x)`: Квадратный корень из x.

•`math.pow(x, y)`: x в степени y.

6.**Угловые преобразования:**

•`math.degrees(x)`: Преобразует угол x из радиан в градусы.

•`math.radians(x)`: Преобразует угол x из градусов в радианы.

7.**Константы:**

•`math.pi`: Число π (пи).

•`math.e`: Число e (экспонента).

Это лишь часть функций, доступных в модуле `math`. Более подробную информацию о функциях и константах вы можете найти в документации по модулю math.

Некоторые методы библиотеки datetime

Библиотека `datetime` в Python предоставляет множество методов для работы с датами и временем. Вот некоторые из наиболее часто используемых методов:

1. **`datetime.now()`**: Возвращает объект `datetime`, представляющий текущую дату и время.

```
from datetime import datetime
```

```
current_datetime = datetime.now()
print(current_datetime)
```

2. **`datetime(year, month, day, hour, minute, second)`**: Создает объект `datetime` с указанными атрибутами.

```
custom_datetime = datetime(2023, 7, 15, 12, 30, 0)
print(custom_datetime)
```

3. **`datetime.strptime(date_string, format)`**: Преобразует строку в объект `datetime` в соответствии с указанным форматом.

```
date_string = '2023-07-15'
custom_datetime = datetime.strptime(date_string, '%Y-%m-%d')
print(custom_datetime)
```

4. **`datetime.strftime(format)`**: Преобразует объект `datetime` в строку в соответствии с указанным форматом.

```
custom_datetime = datetime(2023, 7, 15)
formatted_date = custom_datetime.strftime('%Y-%m-%d')
print(formatted_date)
```

5. **`datetime.timedelta(days, seconds, microseconds, milliseconds, minutes, hours, weeks)`**: Представляет разницу или интервал между двумя датами или временем.

```
from datetime import timedelta

# Создаем интервал в один день
one_day = timedelta(days=1)

# Добавляем интервал к текущей дате
tomorrow = datetime.now() + one_day
print(tomorrow)
```

6. Вы можете использовать объекты `datetime` для расчета разницы между двумя датами и получения количества дней, лет и т. д. между ними. Вот пример:

```
from datetime import datetime

# Задаем две даты
date1 = datetime(1990, 5, 15)
date2 = datetime(2022, 2, 12)

# Рассчитываем разницу между датами
delta = date2 - date1

# Получаем количество дней между датами
days_difference = delta.days
print("Количество дней между датами:", days_difference)

# Получаем количество лет между датами
years_difference = delta.days / 365
```

```
print("Количество лет между датами:", years_difference)
```

Это только небольшой обзор методов библиотеки `datetime`. Она также предоставляет множество других методов для работы с датами, временем, форматами и таймзонами.

1.3 Установка сторонних модулей

1. Подготовительные мероприятия

Перед начало установки библиотек в новом проекте обязательно создавайте новое виртуальное окружения.

Можно создать новое виртуальное окружение:

- среде pycharm при создании проекта
- в терминале через командную строку

`python3 -m venv имя_окружения` (рекомендуется называть виртуальное окружение просто **venv**)

Теперь нужно активировать виртуальное окружение. **Виртуальное окружение** (virtual environment) - это изолированная среда Python, которая позволяет вам управлять зависимостями и библиотеками для каждого проекта отдельно. Когда вы активируете виртуальное окружение, ваша команда Python `python` и `pip` будут использовать версии и пакеты, установленные в этом окружении, а не в глобальной установке Python.

`venv\Scripts\activate`

активацию также можно сделать либо в терминале Windows либо в терминале Pycharm

Менеджер пакетов pip

pip (Python Package Installer) — это менеджер пакетов для языка программирования Python. Он предназначен для управления установкой, обновлением и удалением пакетов, а также для управления их зависимостями.

Проверить, что в вашем проекте или на ПК доступен **pip**, можно применяя следующую команду **--version** или **-V**:

`pip --version`

`pip -V`

Однако, если по какой-то причине его нет, вы можете установить его следующей командой:

```
python -m ensurepip --default-pip
```

2. Где найти и как установить сторонние модули

Установка сторонних модулей в Python чрезвычайно важна для расширения функциональности вашего проекта.

1 Python Package Index (PyPI)

Большинство сторонних модулей **Python** хранятся в **Python Package Index (PyPI)**, общем репозитории для Python-пакетов. PyPI содержит тысячи библиотек, которые могут быть установлены с использованием инструмента `pip`:

```
pip install package_name
```

Например:

```
pip install requests
```

Это установит библиотеку `requests`, которая широко используется для выполнения **HTTP**-запросов.

Если вам нужна конкретная версия модуля, вы можете указать версию вместе с именем модуля:

```
pip install requests==2.26.0
```

можно использовать операторы `<=` (меньше или равно) и `>=` (больше или равно), чтобы установить модуль в определенном диапазоне версий:

```
pip install requests>=2.26.0
```

Это установит `requests` версии 2.26.0 или более новую.

```
pip install requests<=2.26.0
```

Это установит `requests` версии 2.26.0 или более старую.

```
pip install requests>=2.25.0,<=2.26.0
```

Это установит `requests` в диапазоне от 2.25.0 до 2.26.0 включительно.

Для обновления уже установленного модуля до последней версии используйте команду:

```
pip install --upgrade requests
```

Если пакет уже установлен и работает некорректно можно принудительно его переустановить:

```
pip install --force-reinstall requests
```

2 Установка из исходных кодов

Некоторые пакеты могут поставляться с исходными кодами. Для установки из исходных кодов используйте:

```
pip install git+https://github.com/username/repo.git
```

Здесь `https://github.com/username/repo.git` - URL репозитория на GitHub.

Установка сторонних модулей является обычной практикой в разработке на Python, и это делает ваш процесс разработки более гибким и расширяемым.

3 Архивация пакетов

При разработке сложных проектов может понадобиться установка большого количества модулей. Постоянно их скачивать из репозитория **PyPi** трудоемко. Для этого разработан способ загрузки пакетов локально. Они могут находиться в архивах (***.tar.gz**) или специальных файлах с расширением **.whl**. Это удобно и в том случае, если нет доступа в интернет у выбранной машины, и вы заранее создали пакет со всеми необходимыми библиотеками.

Для примера запакуем модуль **numpy** в «колесо» (**wheel**) и установим его оттуда.

```
pip wheel --wheel-dir=. numpy
pip install --no-index --find-links=. numpy
```

Вначале мы создали специальный локальный пакет **NumPy** и поместили его в текущую папку (о чем свидетельствует точка). В директории создался файл **numpy-1.19.2-cp38-cp38-win32.whl**. На его основании даже без интернета мы легко сможем установить данную библиотеку. Команда **«--no-index»** говорит о том, чтобы мы не искали модуль в репозитории **PyPi**, а **--find-links** принудительно указывает место расположения пакета. Когда речь идет о сотне пакетов, это очень удобно. Правда для этого необходимо освоить еще один инструмент: набор зависимостей (о нем – следующий раздел).

4 Установка из архивов (`.whl` файлов)

Некоторые пакеты могут предоставлять архивы с расширением `.whl` (wheel), которые можно использовать для быстрой установки. Пример использования:

```
pip install путь_к_файлу.whl
```

5 Установка из файла зависимостей

Разработчики обычно создают файл **`requirements.txt`**, который содержит список необходимых модулей и их версий.

Файл **`requirements.txt`** может выглядеть примерно так:

```
flask==2.1.0
requests==2.26.0
```

Этот файл позволяет другим разработчикам легко установить все необходимые пакеты и библиотеки для работы с проектом. Для установки этих зависимостей разработчик может воспользоваться командой:

```
pip install -r requirements.txt
```

Для обновления или создания файла **`requirements.txt`** с актуальными зависимостями проекта используется команда:

```
pip freeze > requirements.txt
```

Такой подход упрощает процесс установки всех необходимых компонентов перед началом работы над проектом.

Также рассмотрим более сложный пример. Предположим, что наше приложение может функционировать в питоне версий **2** и **3**. Разумеется, каждая из этих версий требует своего набора модулей. В **requirements.txt** все это можно прописать:

```
cached-property==1.5.2
certifi==2020.6.20 ; python_version < '3.6'
cffi==1.14.3 ; python_version >= '3.6'
chardet==3.0.4
```

Так, если у пользователя на компьютере установлен **Python 3.5**, то модуль **cffi** устанавливаться не будет. Подобный подход несколько портит красоту зависимостей, что ведет в ряде случаев к созданию дополнительного файла **constraints.txt** со списком ограничений. Он нужен для того, чтобы избегать установок лишних модулей в многоуровневых приложениях.

Приведем пример. Для работы нашего приложения требуются некоторые модули определенных версий. Перечислим их в файле **requirements.txt**.

```
Django==3.1.2
django-allauth==0.32.0
```

Для данных библиотек имеются некоторые зависимости. В зависимости от того, на какой версии питона будет работать приложение, могут потребоваться разные модули. Так, **python-openid** работает только на питоне **2** версии, а **python3-openid** нужен для **3**-ей версии. Все эти требования мы прописывает в **requirements.txt**, делая его нечитабельным и неудобным.

```
Django==3.1.2
django-allauth==0.32.0
oauthlib==3.1.0
python-openid==2.2.5 ; python_version < '3.0'
python3-openid==3.2.0; python_version >= '3.0'
```

Намного разумнее - создать **constraints.txt**, в котором прописываются дополнительные зависимости, а в файле **requirements.txt** перечислить основные библиотеки. Когда основные модули начнут подтягивать дополнительные, то они смогут понять, какие же версии им потребуются. Файлы примут такой вид:

```
Django==3.1.2
django-allauth==0.32.0

oauthlib==3.1.0
python-openid==2.2.5
python3-openid==3.2.0
```

Для установки всех требуемых модулей и их зависимостей применяется команда:

```
pip install -c constraints.txt
```

На практике **constraints.txt** используется лишь в крупных проектах, а для небольших разработок можно обойтись простым перечислением зависимостей.

3. Удаление пакетов

Удаление модулей в Python осуществляется с использованием команды `pip uninstall`. Для удаления одного пакета, например, `requests`, используйте следующую команду:

```
pip uninstall requests
```

Если необходимо удалить несколько пакетов, их можно перечислить через пробел:

```
pip uninstall пакет1 пакет2 пакет3
```

Чтобы избежать постоянного запроса подтверждения от пользователя при удалении библиотек, используется ключ `-y` или `--yes`:

```
pip uninstall -y пакет1 пакет2
```

Также можно удалить несколько модулей, перечислив их через пробел, или воспользоваться файлом `requirements.txt`:

```
pip uninstall -r requirements.txt -y
```

При установке нового пакета или его обновлении старая версия автоматически удаляется из библиотеки конкретного окружения. В этом процессе ключ `-y` позволяет избежать ручного подтверждения действия удаления.

5. Дополнительные команды pip

Наберем в командной строке

```
pip help
```

Перед нами предстанет перечень доступных команд в менеджере пакетов. По каждой команде также можно получить справочную информацию:

```
pip install --help
```

Команд и дополнительных ключей в **pip** имеется предостаточное количество. Ознакомиться с ними можно на сайте документации. Мы же рассмотрим основные команды и покажем их применение.

5.1. Команда download

Позволяет скачивать модули в указанную директорию для дальнейшего их использования локально (без повторного скачивания с репозитория).

```
# Устанавливаем модуль в текущую папку (получим файл с расширением «.whl»)
pip download numpy
# Загружаем NumPy в поддиректорию /mods/ (если ее нет, то она создастся)
pip download --destination-directory ./mods/ numpy
# В текущую папку скачаются все пакеты, указанные в файле зависимостей
pip download --destination-directory . -r requirements.txt
# Пример использования дополнительных команд (скачиваем библиотеку только для Линукс-систем и питона не ниже 3 версии)
pip download --platform linux_x86_64 --python-version 3 numpy
```

5.2. Команда list

Команда **list** позволяет просматривать все имеющиеся в виртуальном окружении или глобально модули с указанием их версий.

```
# Отображаем список всех установленных модулей
> pip list
Package    Version
---------- -------
numpy      1.19.0
pip        20.2.4
setuptools 50.3.2
# Отображаем список всех установленных модулей в формате json
> pip list --format json
[{"name": "numpy", "version": "1.19.0"}, {"name": "pip",
"version": "20.2.4"}, {"name": "setuptools", "version": "50.3.2"}]
# Перечисляем модули, которые имеются в системе, но не связаны с
другими (в ряде случаев таким способом можно выяснить модули,
которые вам не нужны либо еще не используемые в проекте)
> pip list --not-required
Package    Version
---------- -------
numpy      1.19.0
# Выводим перечень библиотек, которые требуют обновления (для них
вышла более новая версия)
> pip list -o
Package Version Latest Type
------- ------- ------ -----
numpy   1.19.0  1.19.2 wheel
```

5.3. Команда show

Показывает информацию о конкретном модуле или их группе.

```
# Вывод информации о библиотеке: версия, автор, описание,
расположение, зависимости и т.д.
> pip show numpy
Name: numpy
Version: 1.19.0
Summary: NumPy is the fundamental package for array computing with
Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
...
# Самые полные сведения о модуле
> pip show --verbose numpy
…
Metadata-Version: 2.1
Installer: pip
Classifiers:
```

```
  Development Status :: 5 - Production/Stable
  Intended Audience :: Science/Research
...
```

5.4. Команда check

Нужна для проверки зависимостей, нехватки модулей или их неверной версии.

```
> pip check
No broken requirements found.
```

Если будут найдены ошибки, то выведется соответствующая информация.

5.6. Команда hash

Вычисляет хеш локального модуля на основании определенного алгоритма (можно выбирать).

```
> pip hash numpy-1.19.2-cp38-cp38-win32.whl
numpy-1.19.2-cp38-cp38-win32.whl:--
hash=sha256:51ee93e1fac3fe08ef54ff1c7f329db64d8a9c5557e6c8e908be94
97ac76374b
```

Может понадобиться для проверки полноты скачивания модуля.

5.7. Команда cache

Предоставляет сведения о кеше менеджера пакетов и позволяет оперировать им.

```
# Очистка кеша и удаление всех whl-файлов
> pip cache purge
# Показать папку с кеш-файлами менеджера пакетов
> pip cache dir
c:\users\mmm\appdata\local\pip\cache
# Узнать размер, место расположения и количество файлов в
директории кеша
> pip cache info
Location: c:\users\mmm\appdata\local\pip\cache\wheels
Size: 0 bytes
Number of wheels: 0
```

5.8. Команда debug

Отражает отладочную информацию: о версии пайтона, менеджера пакетов, модулях, операционной системе и т.п.

```
> pip debug
pip version: pip 20.2.4 from d:\1\tmp\venv\lib\site-packages\pip
(python 3.8)
sys.version: 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08)
[MSC v.1926 32 bit (Intel)]
sys.executable: d:\1\tmp\venv\scripts\python.exe
```

```
sys.getdefaultencoding: utf-8
...
```

Сведения могут понадобиться при ошибках и сбоях в работе модулей или установщика.

5.9. Команда completion

Команда актуальная для пользователей *nix систем, чтобы в оболочке bash получить возможность автодополнения команд. Другими словами, при нажатии TAB вы сможете набрать первые 1-2 буквы команды, а остальное определится системой.

```
$ pip completion --bash >> ~/.profile
```

Фактически, вы дополняете профиль командной оболочки командами менеджера пакетов.

5.10. Команда config

Управление файлом настроек пакета модулей (**pip.ini**). Нужно быть аккуратнее, чтобы ничего не сломать в работе питона. Вмешиваться в настройки может понадобиться в случаях наличия нескольких пользователей в вашей операционной системе и при потребности их разграничить в средах программирования.

```
> pip config list
# При вызове команды получаем содержимое файла pip.ini, который
расположен либо в домашней директории, либо в папке виртуального
окружения.
global.require-virtualenv='true'
global.user='false'
```

Зачастую при установке новых пакетов могут возникать ошибки доступа или ограниченных прав того или иного пользователя. Правильное конфигурирование pip позволяет избежать многих проблем.

Задание №1

1.Создать новый проект с использованием нового виртуального окружения.

2.Активировать виртуальное окружение для текущего проекта.

3.Проверить список предустановленных пакетов в виртуальном окружении с помощью команды `pip list`.

4.Установить пакеты `requests`, `bs4`, `selenium`, `pandas`, `numpy` с помощью команды `pip install beautifulsoup4`.

5.Проверить, что указанные пакеты успешно установлены.

6.Создать файл `requirements.txt`, содержащий список всех установленных пакетов в виртуальном окружении.

7.Решение этой задачи поможет создать и настроить новый проект с необходимыми зависимостями, что является хорошей практикой при работе с Python. Это также обеспечивает портабельность и легкость воспроизведения окружения на других системах или устройствах.

1 Создание и импорт собственных модулей

Python предоставляет возможность создавать собственные модули, что позволяет организовать код в более читаемую и управляемую структуру. Модули - это просто файлы Python, содержащие определения и операторы. Давайте рассмотрим этот процесс на примере создания модуля для работы с геометрическими фигурами.

Пример: Модуль для работы с геометрическими фигурами

1 Создание модуля:

Создадим файл `geometry.py`, который будет содержать определения для различных геометрических фигур, таких как круг и прямоугольник.

```
# geometry.py

import math

def calculate_circle_area(radius):
    return math.pi * radius**2

def calculate_rectangle_area(length, width):
    return length * width
```

2 Импорт модуля:

Теперь мы можем создать другой файл, например, `main.py`, и использовать функции из модуля `geometry.py`.

```
# main.py
import geometry

radius = 5
circle_area = geometry.calculate_circle_area(radius)
print(f"Площадь круга с радиусом {radius}: {circle_area}")

length = 4
width = 6
rectangle_area = geometry.calculate_rectangle_area(length, width)
print(f"Площадь прямоугольника с длиной {length} и шириной
{width}: {rectangle_area}")
```

Таким образом, создание и импорт собственных модулей в Python позволяет легко организовывать код, делая его более читаемым, поддерживаемым и многократно используемым

2 Расположение собственных модулей

В Python существует стандартная структура каталогов. Размещение файлов собственных модулей в правильных местах помогает Python легко их находить. Вот общие рекомендации по расположению файлов:

Каталог вашего проекта:

•Создайте основной каталог для вашего проекта.

•Разместите файлы вашего модуля, например, `geometry.py`, в этом каталоге.

```
/your_project
└── geometry.py
```

Добавление пути в переменную окружения `PYTHONPATH`:

•Если вы разместили ваш модуль в отдельном месте, не являющемся каталогом проекта, добавьте путь к этому месту в переменную окружения `PYTHONPATH`.

```
export PYTHONPATH=/путь/к/вашему/модулю:$PYTHONPATH
```

•Выполнив эту команду, Python будет искать модули также и в указанном каталоге.

2.**Использование относительного импорта:**

•Если ваш проект имеет несколько подкаталогов, вы можете использовать относительный импорт.

```
/your_project
├── main.py
└── subdirectory
    └── geometry.py
```

В `main.py` вы можете использовать относительный импорт следующим образом:

```
# main.py
from subdirectory import geometry
```

•Важно, чтобы в подкаталоге был файл `__init__.py` (даже если он пустой), чтобы Python рассматривал каталог как пакет.

2.**Использование `sys.path`:**

•Добавьте путь к вашему модулю в `sys.path` в начале вашего скрипта:

```
# main.py
import sys
sys.path.insert(0, '/путь/к/вашему/модулю')

from geometry import calculate_circle_area
```

Это временное решение и не рекомендуется для постоянного использования.

Выбор между этими методами зависит от конкретных требований вашего проекта. В большинстве случаев рекомендуется использовать структуру каталогов и

правильные пути, чтобы упростить организацию кода и обеспечить легкость его поддержки.

3 Пакеты

Пакеты в Python - это способ организации модулей в иерархическую структуру. Пакеты представляют собой способ организации модулей по единой тематике, позволяя группировать связанные функции, классы и другие объекты в единый компонент. Каждый пакет обычно содержит модули, направленные на выполнение определенной задачи или решение определенной проблемы, что способствует более структурированной и удобной организации кода проекта.

Создадим пакет

Пакет в Python представляет собой каталог, который содержит модули и/или другие подпакеты. Он также содержит специальный модуль с именем `__init__.py`, который выполняет инициализацию пакета и может содержать общие для всех модулей в пакете объекты, код или настройки. Этот файл также необходим для того, чтобы Python интерпретировал каталог как пакет.

Рассмотрим демонстрационный пример пакета. Создадим пакет из каталога `mypackage` внутри проекта `project`:

```
project/
|
├── mypackage/
|   ├── __init__.py
|   ├── constants.py
|   ├── module1.py
|   ├── module2.py
└── main.py
```

В файл `__init__.py` добавим следующий код:

```
# Этот файл может быть пустым, но он говорит Python, что каталог `mypackage` - это пакет

from .constants import *

# Можно также объявить переменные здесь напрямую, но в данном примере они импортируются из файла constants.py
```

Обратите внимание, что отдельно импортировать файл `__init__.py` не нужно. При первом обращении к пакету Python самостоятельно импортирует модуль `__init__.py`.

Файл `constants.py`:

```
# Файл с объявлением констант, которые будут использоваться в модулях пакета

CONSTANT_1 = 100
CONSTANT_2 = "Hello, world!"
```

Файл `module1.py`:

```
# Пример модуля в пакете mypackage

from .constants import CONSTANT_1

def func1():
    print("Function 1 in module1.py")

def func2():
    print(f"Constant from constants.py: {CONSTANT_1}")
```

Файл `module2.py`:

```
# Еще один пример модуля в пакете mypackage

from .constants import CONSTANT_2

def func3():
    print("Function 3 in module2.py")

def func4():
    print(f"Constant from constants.py: {CONSTANT_2}")
```

Теперь давайте создадим программу для демонстрации использования модулей из нашего пакета:

```
# Пример использования модулей из пакета mypackage

from mypackage import module1, module2

module1.func1()
module1.func2()
module2.func3()
module2.func4()
```

4 Варианты импорта пакетов

Пакеты как и модули можно импортировать:

•Квалифицированный импорт - «импорт модуля целиком»

•Импорт отдельных определений

Квалифицированный импорт:

```
import mypackage.module1
import mypackage.module2

mypackage.module1.func1()
mypackage.module1.func2()
mypackage.module2.func3()
mypackage.module2.func4()
```

Импорт отдельных определений удобнее, потому что не нужно каждый раз прописывать имя пакета и модуля.

```
# Пример использования модулей из пакета mypackage

from mypackage.module1 import func1, func2
from mypackage.module2 import func3, func4

func1()
func2()
func3()
func4()
```

Кроме того, импорты в Python бывают двух видов:

•Абсолютные

•Относительные

В следующем шаге посмотрим как это работает

5 Варианты импортов

Рассмотрим на предыдущем пример ну уже с подпакетом:

```
project/
|
├── mypackage/
|   ├── __init__.py
|   ├── constants.py
|   ├── module1.py
|   ├── module2.py
|   └── subpackage/
|       ├── __init__.py
|       ├── sub_module1.py
|       └── sub_module2.py
└── main.py
```

subpackage/sub_module1.py:

```
def sub_func1():
    print("Sub Function 1 from sub_module1.py")

def sub_func2():
    print("Sub Function 2 from sub_module1.py")
```

subpackage/sub_module2.py:

```
def sub_func3():
    print("Sub Function 3 from sub_module2.py")

def sub_func4():
    print("Sub Function 4 from sub_module2.py")
```

Абсолютный импорт

В абсолютном импорте нужно прописывать **полный путь до модуля**, включающего все пакеты и подпакеты. Полные пути гарантируют простоту чтения и однозначность — так всем будет понятно, что и откуда импортируется. Чтобы вам было удобнее читать код, во всех примерах выше мы использовали абсолютный импорт.

Относительный импорт

Относительные импорты выглядят так:

```
from . import module
from .module import function
from .subpackage.module import CONSTANT
```

В относительном импорте используется точка, которая означает **импорт модуля из текущей директории**.

Например, мы работаем в файле `main.py` и хотим импортировать `.module`. При этом мы знаем, что `main.py` и `.module` хранятся в одной и той же директории. Тогда можно не прописывать абсолютный путь к `.module`, а воспользоваться `. import`. По этой точке Python автоматически определит, что и откуда мы хотим импортировать.

Относительный импорт помогает писать быстрее, но слишком сильно запутывает код и негативно сказывается на читаемости. Именно поэтому в сообществе Python-разработчиков есть распространенный совет для новичков: старайтесь пользоваться абсолютным импортом, даже в самых простых и очевидных случаях.

6 Учебное задание: Создание пакета с подпакетами в Python

Цель: Научиться создавать собственный пакет с подпакетами в Python

1.Создание основного пакета и подпакетов:

•Создайте директорию с именем **`figure`**.

•Внутри директории **`figure`** создайте следующие директории: **`circle`**, "triangle", "rectangle".

•В каждой из директорий "**`circle`**", "**`triangle`**", "**`rectangle`**" создайте файл **`__init__.py`** , чтобы Python рассматривал эти директории как пакеты.

2.Создание функций:

•Внутри каждой из директорий "**`circle`**", "**`triangle`**", "**`rectangle`**" создайте файлы с названиями "area.py" и "perimeter.py".

•В каждом из файлов "**`area.py`**" и "**`perimeter.py`**" напишите функции для вычисления площади и периметра фигуры соответственно. Ниже приведены названия функций и их аргументы:

•Для "circle":

•Функция для вычисления площади: `calculate_area(radius)`

•Функция для вычисления периметра: `calculate_perimeter(radius)`

•Для "triangle":

•Функция для вычисления площади: `calculate_area(side1, side2, side3)`

•Функция для вычисления периметра: `calculate_perimeter(side1, side2, side3)`

•Для "rectangle":

•Функция для вычисления площади: `calculate_area(length, width)`

•Функция для вычисления периметра: `calculate_perimeter(length, width)`

3.Использование пакета:

•Создайте основной скрипт, например, "main.py", в котором импортируйте функции из созданных пакетов.

•Продемонстрируйте использование функций для вычисления площади и периметра различных фигур (круга, треугольника, прямоугольника).

Реализация

Структура пакета будет выглядеть так:

```
figure/
    __init__.py
    circle/
        __init__.py
        area.py
        perimeter.py
    triangle/
        __init__.py
        area.py
        perimeter.py
    rectangle/
        __init__.py
```

```
        area.py
        perimeter.py
```

Каждый из подпакетов (`circle`, `triangle`, `rectangle`) содержит файлы `area.py` и `perimeter.py`, которые определяют функции для вычисления площади и периметра соответствующей фигуры.

В файле `figure/__init__.py` вы можете импортировать функции из подпакетов и предоставить их пользователю через основной пакет `figure`. Вот как это можно сделать:

```
# В файле figure/__init__.py

from .circle.area import calculate_area as circle_area
from .circle.perimeter import calculate_perimeter as
circle_perimeter

from .triangle.area import calculate_area as triangle_area
from .triangle.perimeter import calculate_perimeter as
triangle_perimeter

from .rectangle.area import calculate_area as rectangle_area
from .rectangle.perimeter import calculate_perimeter as
rectangle_perimeter
```

Теперь, если пользователь импортирует пакет `figure`, он может использовать функции для вычисления площади и периметра круга, треугольника и прямоугольника напрямую из пакета `figure`.

Пример использования:

```
import figure

# Вычисление площади и периметра круга
circle_radius = 5
circle_area = figure.circle_area(circle_radius)
circle_perimeter = figure.circle_perimeter(circle_radius)
print(circle_area)
print(circle_perimeter)

# Вычисление площади и периметра треугольника
triangle_side1 = 4
triangle_side2 = 3
triangle_side3 = 5
triangle_area = figure.triangle_area(triangle_side1,
triangle_side2, triangle_side3)
triangle_perimeter = figure.triangle_perimeter(3, 4, 5)
print(triangle_area)
print(triangle_perimeter)

# Вычисление площади и периметра прямоугольника
rectangle_length = 6
```

```
rectangle_width = 8
rectangle_area = figure.rectangle_area(rectangle_length,
rectangle_width)
rectangle_perimeter = figure.rectangle_perimeter(rectangle_length,
rectangle_width)
print(rectangle_area)
print(rectangle_perimeter)
```

Таким образом, пакет `figure` использует функции из подпакетов `circle`, `triangle` и `rectangle`, предоставляя им доступ через основной пакет.

7 Публикация своей Python библиотеки на PyPI

Вы разработали собственную библиотеку Python и считаете, что она может быть полезна другим разработчикам. Публикация вашей библиотеки является отличным способом внести свой вклад в сообщество Python. Любой разработчик сможет легко установить ваш пакет и начать использовать его, импортировав необходимые модули или функции. Поделитесь своим кодом и дайте другим возможность воспользоваться вашими решениями и улучшить их для общего блага.

1. Создаём элементы библиотеки

Давайте изучим публикацию собственных библиотек, на примере разработанного на предыдущем шаге пакета `figure`

Для публикации на PyPi корневой каталог (в нашем случае `figure`) должен содержать следующие файлы и директории:

•`setup.py` – содержит метаданные пакета,

•`setup.cfg` – конфигурационный файл, используемый для хранения настроек,

•подпапку с тем же именем, что и родительская папка (в данном примере `figure`),

где хранится фактический код вашего пакета на Python.

Примеры содержания файлов в нашем случае:

setup.py

```
from setuptools import setup

setup(name='figure',
      version='1.0',
      description='circle, tiangle, rectangle',
      packages=['figure.circle', 'figure.triangle',
'figure.rectangle'],
      author_email='abc@def.ru',
      zip_safe=False)
```

setup.cfg

```
[egg_info]
tag_build =
tag_date = 0
```

Любая публикуемая на **PyPI** библиотека обязана иметь три вышеуказанных элемента. Помимо этого, пакет должен выполнять следующие условия:

•Уникальное имя библиотеки. Никакие два существующих пакета Python не могут одинаково называться.

•Файл `setup.py` должен содержать параметры `name` и `packages`. Они также должны иметь то же имя, что и пакет (см. пример выше).

•Параметр `version`: в случае если что-то в пакете изменилось, нужно изменить и значение `version`.

•Файла `setup.py` должен иметь параметр `author_email` с адресом электронной почты для связи. Используйте действующий e-mail, это важно для дальнейшей регистрации программы в репозитории **PyPI**.

•В нашем случае есть подкаталоги, чтобы они тоже вошли в сборку, нужно их указать в списке всех пакетов packages=`['figure.circle', 'figure.triangle', 'figure.rectangle']`

2. Подготавливаем код

Как указано выше, внутри вложенной папки должен находиться фактический код библиотеки. Если вы откроете подпапку `figure`, вы увидите, что она содержит файл `_init_.py`. По умолчанию ядро Python ищет файл `_init_.py` в качестве отправной точки при чтении кода. Файл `_init_.py` связан со всеми другими сценариями Python в подпапке. Например, в нашем файле `_init_.py`:

```
# В файле figure/__init__.py

from .circle.area import calculate_area as circle_area
from .circle.perimeter import calculate_perimeter as
circle_perimeter

from .triangle.area import calculate_area as triangle_area
from .triangle.perimeter import calculate_perimeter as
triangle_perimeter

from .rectangle.area import calculate_area as rectangle_area
from .rectangle.perimeter import calculate_perimeter as
rectangle_perimeter
```

3. Создаём аккаунт PyPI

Пакеты Python, доступные для общего использования, хранятся в репозитории PyPI. При установке пакета на ваш компьютер с помощью инструмента pip, вы скачиваете его из репозитория PyPI. Для публикации нового пакета вам, наоборот, необходимо загрузить его на сервер PyPI. Для этого требуется создать учетную запись на сайте PyPI (это бесплатно). При создании учетной записи убедитесь, что адрес электронной почты совпадает с тем, что указан в файле setup.py в параметре author_email.

4. Публикуем библиотеку

Публикация осуществляется из командной строки или терминала. Команды идентичны в Windows и Linux.

Для публикации должны быть две библиотеки – `setuptools` и `twine`. `setuptools` - обычно уже установлена

```
pip install setuptools
```

```
pip install twine
```

Переходим к родительскому каталогу **`\figure`**

Развёртываем пакет запустив `setup.py`

```
python setup.py sdist
```

Обратите внимание, что теперь в родительской папке будут созданы две новых директории (`egg-info` и `dist`).

Теперь с помощью `twine` развёртываем пакет на PyPI:

```
twine upload dist/*
```

Для размещения пакета на сервере PyPI необходимо ввести token, его можно сгенерировать в личном кабинет PyPI .

После успешного размещения, пакет становится доступным для использования. Чтобы убедиться, что пакет успешно загружен на сервер, зайдите в вашу учетную запись на PyPI и перейдите в раздел "Your projects". Здесь вы сможете увидеть вашу библиотеку, готовую к использованию.

5. Используем

Давайте перейдем к выбранному клиенту Python, такому как **`Colab`**, **`Pycharm`**или просто в терминале на ваше компьютере и установим пакет с помощью **`pip install figure`**.

После установки, мы можем импортировать его и использовать по своему усмотрению.