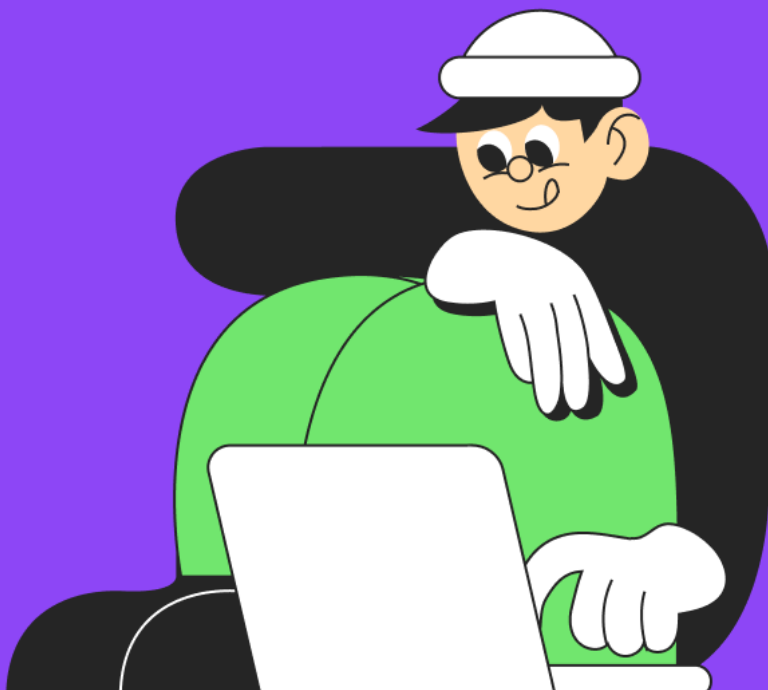
GeekBrains

Лекция 15. Обзор стандартной библиотеки Python

Погружение в Python



Оглавление

На этой лекции мы

1. Узнаем о составе стандартной библиотеки Python.
2. Разберёмся в настройках логирования
3. Изучим работу с датой и временем
4. Узнаем ещё пару полезных структур данных
5. Изучим способы парсинга аргументов при запуске скрипта с параметрами

Дополнительные материалы к лекции

Стандартная библиотека Python https://docs.python.org/3.11/library

Краткая выжимка, о чём говорилось в предыдущей лекции

На прошлой лекции мы:

1. Разобрались с написанием тестов в Python
2. Изучили возможности doctest
3. Узнали о пакете для тестирования unittest
4. Разобрались с тестированием через pytest

Подробный текст лекции
1. Обзор библиотеки целиком

Мы уже упоминали стандартную библиотеку Python на курсе. Даже использовали некоторый пакеты из неё. Вспомним, что же это такое.
Стандартная библиотека Python (The Python Standard Library) — распространяется вместе с интерпретатором Python. Следовательно для использования пакетов и модулей из неё достаточно написать import name.
Предлагаю для краткости называть Стандартную библиотеку Python просто библиотека в рамках этой лекции.
Библиотека очень обширна и позволяет решать огромный спектр задач. Внутри есть модули написанные на языке C. Благодаря ним у разработчиков есть лёгкий доступ к системным функциям. Другие модули написаны на Python и предлагают готовые решения для повседневных задач программирования.
Библиотека содержит:

- Встроенные функции
- Встроенные константы
- Встроенные типы
- Встроенные исключения
- Услуги обработки текста
- Службы двоичных данных
- Типы данных
- Числовые и математические модули
- Модули функционального программирования
- Доступ к файлам и каталогам
- Сохранение данных
- Сжатие данных и архивирование
- Форматы файлов
- Криптографические услуги
- Общие службы операционной системы
- Параллельное выполнение
- Сеть и межпроцессное взаимодействие
- Обработка данных в Интернете
- Инструменты обработки структурированной разметки
- Интернет-протоколы и поддержка
- Мультимедийные услуги
- Интернационализация
- Программные фреймворки
- Графические пользовательские интерфейсы с Tk
- Инструменты разработки

- Отладка и профилирование
- Упаковка и распространение программного обеспечения
- Службы выполнения Python
- Пользовательские интерпретаторы Python
- Импорт модулей
- Языковые службы Python
- Специальные службы MS Windows
- Специальные службы Unix
- Замененные (устаревшие) модули

И под каждым пунктом кроется несколько пакетов для решения соответствующих теме задач. Вы можете перейти по ссылке и увидеть детальный состав библиотеки самостоятельно https://docs.python.org/3.11/library/index.html.

Далее в рамках лекции рассмотрим несколько полезных модулей.

2. Модель logging

Модуль logging позволяет регистрировать события в приложениях. Для этого разработчику предоставляется гибкая система классов и функций. Традиционно начнём с простого примера.

```
import logging

logging.info('Немного информации')
logging.error('Поймали ошибку')
```

В результата получим строку:

```
ERROR:root:Поймали ошибку
```

Мы не увидели в консоли первое сообщение. По умолчанию модуль не реагирует на информационные сообщения. При этом логгер сообщил об ошибке от имени корневого регистратора — root.

Уровни логирования

По умолчанию логгер имеет следующие уровни журналирования

- NOTSET, 0 — уровень не установлен. Регистрируются все события.
- DEBUG, 10 — подробная информация, обычно представляющая интерес только при диагностике проблем.
- INFO, 20 — подтверждение того, что все работает так, как ожидалось.
- WARNING, 30 — указание на то, что произошло что-то неожиданное, или указание на какую-то проблему в ближайшем будущем (например, «недостаточно места на диске»). Программное обеспечение по-прежнему работает, как ожидалось.
- ERROR, 40 — из-за более серьезной проблемы программное обеспечение не может выполнять некоторые функции.
- CRITICAL, 50 — серьезная ошибка, указывающая на то, что сама программа не может продолжать работу.

Уровень NOTSET используется как значение по умолчанию при создании обработчиков событий. Все остальные уровни имеют одноимённые функции — регистраторы, которые позволяют зафиксировать события заданного уровня.
Кроме того уровни имеют числовой эквивалент от 0 до 50. Он нужен при создании своих уровней логирования, чтобы определить место обработчика относительно базовых.

🔥 **Важно!** На протяжении курса мы использовали функцию print() чтобы посмотреть отладочную информацию в том или ином коде. Обычно, но не всегда, рядом с таким принтом составлялся комментарий, что этот вывод для учебных целей, а не для реальных проектов. Правильно было бы использовать логирование уровня debug или info вместо функции print.

Базовые регистраторы

Имена функций для регистрации событий совпадают с названиями констант для определения уровня логирования. Перечислим их в примере кода.

```
import logging

logging.basicConfig(level=logging.NOTSET)
logging.debug('Очень подробная отладочная информация. Заменяем множество "принтов"')
logging.info('Немного информации о работе кода')
```

```
logging.warning('Внимание! Надвигается буря!')
logging.error('Поймали ошибку. Дальше только неизвестность')
logging.critical('На этом всё')
```

Разработчик сам выбирает какой уровень использовать для регистрации того или иного события в его коде.
Обычно в коде не используют прямое обращение к регистраторам через имя модуля. В официальное документации указано, что работать с регистраторами напрямую запрещено. Необходимо использовать функцию уровня модуля logging.getLogger(name) для получения регистраторов. В таком случае пример выше должен выглядеть так:

```
import logging

logging.basicConfig(level=logging.NOTSET)
logger = logging.getLogger(__name__)

logger.debug('Очень подробная отладочная информация. Заменяем множество "принтов"')
logger.info('Немного информации о работе кода')
logger.warning('Внимание! Надвигается буря!')
logger.error('Поймали ошибку. Дальше только неизвестность')
logger.critical('На этом всё')
```

Напомним, что переменная __name__ хранит имя модуля. Если мы запускаем модуль как исполняемый файл, в __name__ хранится __main__. Если файл импортирован, __name__ хранит имя файла.

А теперь рассмотрим ситуацию с несколькими файлами, когда мы работаем со сложным проектом.
Код основного файла:

```
import logging
from other import log_all


logging.basicConfig(level=logging.WARNING)
logger = logging.getLogger('Основной файл проекта')
logger.warning('Внимание! Используем вызов функции из другого модуля')
log_all()
```

Код другого файла в проекте

```
import logging
```

```
logger = logging.getLogger(__name__)


def log_all():
    logger.debug('Очень подробная отладочная информация. Заменяем
множество "принтов"')
    logger.info('Немного информации о работе кода')
    logger.warning('Внимание! Надвигается буря!')
    logger.error('Поймали ошибку. Дальше только неизвестность')
    logger.critical('На этом всё')
```

В основном коде определили уровень логирования - WARNING. Логер вывел сообщение и вместо root указал текст, переданный в функцию getLogger.
Далее мы вызываем импортированную функцию и...
В файле other так же импортирован модуль logging. Далее мы получаем регистратор с именем other, оно содержится в __name__. И не смотря на попытку использовать все уровни логирования, срабатывают только предупреждения и выше. Функция getLogger взяла конфигурацию basicConfig из основного файла проекта.

Подробнее о basicConfig

Функция basicConfig нужна для базовой настройки логгеров. Она должна быть вызвана в основном потоке раньше, чем будут вызывать логгеры любого уровня.
Функция принимает только ключевые параметры, так что важен не порядок, а имя.

- ➢ **Уровень логирования**

Как вы уже догадались ключевой параметр level принимает константу с числом для определения уровня логирования. Все значения меньше переданного будут игнорироватся регистраторами.

- ➢ **Файл журнала**

До этого момента информация логгеров выводилась в консоль. Но мы можем сохранять данные в файл, указав необходимые настройки в конфигурации.

```
import logging
from other import log_all
```

```
logging.basicConfig(filename='project.log.',        filemode='w',
encoding='utf-8',  level=logging.INFO)
logger = logging.getLogger('Основной файл проекта')
logger.warning('Внимание! Используем вызов функции из другого
модуля')
log_all()
```

Параметры работы с файлами аналогичны функции open().

- filename — путь и имя файла для сохранения журнала логирования.
- filemode — режим записи. Если передать 'w', каждый запуск будет удалять старые данные из журнала. Можно не указывать аргумента, тогда данные будут дописываться в конец файла
- encoding — кодировка для сохранения журнала
- errors — способ обработки ошибок при открытии файла. По умолчанию используется режим backslashreplace. Он заменяет искаженные данные управляющими последовательностями Python с обратной косой чертой.

➢ **Формат сохранения события**

Кроме того для базовой конфигурации используют параметр format для изменения стандартной строки логирования. Формат может быть задан в трёх стилях форматирования:

- “%” — старый стиль форматирования текста с использованием символа %. Это стиль по умолчанию для задания формата. Сохранён он для поддержки обратной совместимости код. Ведь логирование работа ещё в Python 1.5.2 созданном в прошлом тысячелетии.
- “{“ — форматирование с использованием фигурных скобок. Метод str.format.
- “$” — форматирование в стиле string.Template с использованием денежного символа для указания имени переменной.

🔥 **Важно!** Не путайте форматирование с использованием фигурных скобок в методе str.format с форматированием f-строк. Впрочем, они имеют много общего.

Рассмотрим стил с фигурными скобками как наиболее привычный современным разработчикам. Кстати, сам стиль надо указать при вызове конфигуратора в параметре style.

```
import logging
from other import log_all

FORMAT = '{levelname:<8} - {asctime}. В модуле "{name}" ' \
         'в строке {lineno:03d} функция "{funcName}()" ' \
```

```
        'в {created} секунд записала сообщение: {msg}'
logging.basicConfig(format=FORMAT, style='{', level=logging.INFO)
logger = logging.getLogger('main')
logger.warning('Внимание! Используем вызов функции из другого
модуля')
log_all()
```

Константа формат передаёт уровень логирования и время записи в читаемом формате. Далее указываем имя модуля и строку кода, откуда был вызван логгер. Если он находился в функции, то её имя и отдельно время срабатывания в секундах от 1 января 1970 года. В финале переменная msg выводит текст сообщения.
Обратите внимание, что к некоторым значениям было применено форматирование.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
import logging

logging.basicConfig(
    filename="log/log.log",
    encoding='utf-8',
    format='{asctime} {levelname} {funcName}->{lineno}: {msg}',
    style='{',
    level=logging.WARNING
)
```

3. Модуль datetime

Ещё один полезный модуль — datetime. Как понятно из названия он необходим для обработки даты и времени.

💡 **Внимание!** Вопрос работы с часовыми поясами специально опущен в этой главе. Но вы всегда можете обратиться к модулю zoneinfo, который появился в Python 3.9. Или установить внешний пакет, например pytz.

🔥 **Важно!** Прежде чем начать работать с датами, стоит помнить что модуль datetime представляет “вечный” Григорианский календарь. Т.е. он не учитывает другие календарные системы и считает что григорианский календарь всегда был и всегда будет.

Создаём дату и время

Следующий пример демонстрирует три основных типа данных модуля.

```
from datetime import time, date, datetime

d = date(year=2007, month=6, day=15)
t = time(hour=2, minute=14, second=0, microsecond=24)
dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
second=0, microsecond=24)
print(f'{d = }\t-\t{d}')
print(f'{t = }\t-\t{t}')
print(f'{dt = }\t-\t{dt}')
```

- date представляет дату в календаре от 1 января 1 года до 31 декабря 9999 года. Указывать год, месяц и число обязательно для создания объекта.
- time создаёт объект времени, который на вход принимает четыре необязательных параметра. Если любое из четырёх значений или даже все сразу отсутствуют, вместо него подставляется ноль
    - часы — от 0 до 23 часов
    - минуты — от 0 до 59 минут
    - секунды — от 0 до 59 секунд
    - микросекунды — от 0 до 999_999 микросекунд
- datetime является комбинацией даты и времени и при его создании действуют правила аналогичные для date и time.

Объекты являются неизменяемыми типами. Следовательно они возвращают хеш и могут использоваться как ключи словаря, элементы множества и т.д.

Разница времени

Ещё один важный тип данных — timedelta. Объект представляет из себя разницу во времени между различными датами и временными промежутками.
Простой пример создания временной разницы:

```
from datetime import timedelta

delta = timedelta(weeks=1, days=2, hours=3, minutes=4, seconds=5,
milliseconds=6, microseconds=7)
print(f'{delta = }\t-\t{delta}')
```

Независимо от данных, которые используются для создания дельты, она хранит три значения:

- days — хранит переданные дни. Недели преобразуются в 7 дней
- seconds — хранит секунды. Минуты превращаются в 60 секунд, а часы в 3600 секунд
- microseconds — хранит микросекунды. Миллисекунды превращаются в 1000 микросекунд.

При выводе на печать получаем количество дней, часов, минут, секунд и микросекунд.
В отличии от трёх временных типов, дельты могут принимать на вход любые числовые значения. Например количество минут может быть больше 60. Кроме того дельта может быть отрицательной.

```
from datetime import timedelta

delta = timedelta(weeks=53, days=500, hours=73, minutes=101,
                                  seconds=303,  milliseconds=67890,
microseconds=1234567)
neg_delta = timedelta(days=-3, minutes=-42)
print(f'{delta = }\t-\t{delta}')
print(f'{neg_delta = }\t-\t{neg_delta}')
```

При переполнении любого из трёх значений, излишек преобразуется в следующее по старшинству. Главное, чтобы дни остались в диапазоне от минус миллиарда до плюс миллиарда.
Обратите внимание, что при отрицательных значениях с минусом хранятся только дни, а два других параметра остаются положительными и при хранении и при выводе информации.

Математика с датами

Разница во времени появляется при математических операциях с датами и временем. И этажа разница может использоваться для изменения дат. Например

```
from datetime import datetime, timedelta

date_1 = datetime(2012, 12, 21)
date_2 = datetime(2017, 8, 19)
delta = date_2 - date_1
print(f'{delta = }\t-\t{delta}')

birthday = datetime(1503, 12, 14)
dlt = timedelta(days=365.25 * 33)
new_date = birthday + dlt
print(f'{new_date = }\t-\t{new_date}')
```

В первом случае нашли разницу во времени между двумя событиями. А во втором вычислили дату тридцатитрёхлетия.

Доступ к свойствам

Каждый из объектов позволяет прочитать хранимые свойства обратившись к ним по имени через точечную нотацию.

```
from datetime import time, date, datetime, timedelta

d = date(year=2007, month=6, day=15)
t = time(hour=2, minute=14, microsecond=24)
dt  =  datetime(year=2007,  month=6,  day=15,  hour=2,  minute=14,
microsecond=24)
delta = timedelta(weeks=53, hours=73, minutes=101,
                  seconds=303, milliseconds=60)
print(f'{d.month}')
print(f'{t.second}')
print(f'{dt.hour}')
print(f'{delta.days}')
```

Даже если свойство явно не передано, но объект хранит его, мы можем получить доступ на чтение.

Изменить значение напрямую не получится. Всё же перед нами неизменяемые объекты. Но мы можем воспользоваться методом replace для создания копии со значениями текущего объекта. Изменения затронут только указанные параметры.

💡 **Внимание!** timedelta не имеет метода replace.

```
from datetime import time, date, datetime, timedelta

t = time(hour=2, minute=14, microsecond=24)
dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
microsecond=24)

new_dt = dt.replace(year=2012)
one_more_hour = t.replace(t.hour + 1)
print(f'{new_dt}\n{one_more_hour}')
```

Другие методы работы с датой и временем

Мы можем использовать специальные методы для получения удобного вывода информации и наоборот, формировать объекты datetime из человекочитаемых строк.

Рассмотрим некоторые из них.

```
from datetime import datetime

dt = datetime(year=2007, month=6, day=15, hour=2, minute=14,
microsecond=24)

print(dt)
print(dt.timestamp())
print(dt.isoformat())
print(dt.weekday())
print(dt.strftime('Дата %d %B %Y. День недели %A. Время %H:%M:%S.
Это %W неделя и %j день года.'))
```

- timestamp — возвращаем время в секундах от начала времён. Под началом времён понимаем POSIX-время, т.е полночь 1 января 1970 года. Программисты использует его как особую точку отсчёта, позволяющую

хранить время как целое (если нужна точность до секунды) или вещественное число. Подобный подход используется для хранения времени в БД и для манипуляций со временем.

- isoformat — выводит дату в формате, соответствующем стандарту ISO 8601. Это международный стандарт описывающий формат даты и времени и рекомендации по его использованию.
- weekday — позволяет получит день недели в виде целого числа, где 0 - понедельник, а 6 — воскресенье.
- strftime — выводит дату в соответствии с переданным в виде str форматом. Обычный текст выводится без изменения, а символ % указывает на следующий после него литер форматирования. Ознакомится со всеми можно по ссылке.

А теперь несколько обратных методов, позволяющих создать объекты datetime.

```
from datetime import datetime

date_original = '2022-12-12 18:01:21.555470'
date_timestamp = 1181862840.000024
date_iso = '2007-06-15T02:14:00.000024'
date_text = 'Дата 15 June 2007. День недели Friday. Время 02:14:00. Это 24 неделя и 166 день года.'

original_date = datetime.fromisoformat(date_original)
timestamp_date = datetime.fromtimestamp(date_timestamp)
iso_date = datetime.fromisoformat(date_iso)
text_date = datetime.strptime(date_text, 'Дата %d %B %Y. День недели %A. Время %H:%M:%S. Это %W неделя и %j день года.')
print(original_date)
print(timestamp_date)
print(iso_date)
print(text_date)
```

- fromisoformat — метод анализирует строку текст и если она совпадает со стандартом ISO 8601, возвращает объект datetime. Стандартный вывод объектов на печать позволяет делать обратное преобразование этим методом.
- fromtimestamp — получаем объект даты из целого или вещественного числа. Само число — количество секунд после полуночи 1 января 1970.
- strptime — метод принимает на вход два параметра: строку для преобразования и строку с форматом. Если формат позволяет проанализировать строку, возвращается объект datetime.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from datetime import datetime, timedelta

d = datetime.now()
td = timedelta(hours=1)
for i in range(24*7):
    if d.weekday() == 6:
        break
    else:
        d = d + td
print(i)
```

4. Пара полезных структур данных

Модуль collections

Модуль предоставляет доступ к встроенным в Python типам данных, но “спрятанным” от начинающего разработчика. Внутри хранится много интересных и полезных структур данных. А в collections.abc огромный набор абстрактных типов. Но сегодня мы рассмотрим функцию namedtuple из модуля.

Фабричная функция namedtuple

Функция namedtuple является фабрикой классов. Из названия следует, что она создаёт именованные кортежи. Рассмотрим простой пример, чтобы понять что это.

```
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name',
'birthday'])
u_1 = User('Исаак', 'Ньютон', datetime(1643, 1, 4))
print(u_1)
```

```
print(f'{type(User)}, {type(u_1)}')
```

Функция принимает пару обязательных значений:

1. Имя класса. Это строка, содержащее точно такое же имя как и переменная слева от знака равно.
2. Список строк или строка с пробелами в качестве разделителей. Имена из списка превращаются в свойства класса.

На выходе получаем класс, аналогичный созданному вручную классу class. При этом в классе помимо указанных в списке свойств автоматически создаются некоторые дандер методы. Например мы с лёгкостью распечатали экземпляр u_1, потому что он определил свой дандер __repr__.
Как и с экземплярами класса, мы можем получить доступ к свойствам используя точечную нотацию.

```
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name',
'birthday'])
u_1 = User('Исаак', 'Ньютон', datetime(1643, 1, 4))
print(u_1)
print(u_1.first_name, u_1.birthday.year)
```

Обратите внимание, что свойство день рождение является объектом datetime со своими свойствами. Доступ к ним мы получаем также через точечную нотацию

При создании класса можно дополнительно передать список значений по умолчанию. И если дефолтных значений меньше, чем свойств в классе, назначение происходит справа налево

```
import time
from collections import namedtuple
from datetime import datetime

User = namedtuple('User', ['first_name', 'last_name',
'birthday'], defaults=['Иванов', datetime.now()])
u_1 = User('Исаак')
print(f'{u_1.last_name}, {u_1.birthday.strftime("%H:%M:%S")}')
time.sleep(7)
u_2 = User('Галилей', 'Галилео')
print(f'{u_2.last_name}, {u_2.birthday.strftime("%H:%M:%S")}')
```

Составление списка с именами полей и значениями по умолчанию движется справа налево, поэтому в birthday попадает текущая дата, а фамилию по умолчанию -

Иванов. Значения подставляются только в том случае, когда экземпляр не передаёт свои, как и с обычными классами.

💡 **Внимание!** Посмотрите на даты у каждого из экземпляров. Время совпадает до секунды несмотря на 7 секунд разницы в создании. Значения для birthday было вычислено один раз, в момент создания класса.

На самом деле ситуация с функциями неоднозначно. С одной стороны ничего не мешает присвоить экземпляру в качестве свойства созданную функцию. Но в отличии от классических классов, классы namedtuple рассчитаны на хранение свойств, а не методов.

🔥 **Важно!** Если вам нужен объект с методами, используйте классический ООП.

Как и в случае с неизменяемыми датами, экземпляры namedtuple также неизменны. Но если надо внести правку в какое-то поле, встроенный метод _replace создаст копию, заменив только указанные значения.

```
from collections import namedtuple


Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
a = Point(2, 3, 4)
b = a._replace(z=0, x=a.x + 4)
print(b)
```

Экземпляры можно сортировать. Метод проверки “меньше” определяется для свойств автоматически

```
from collections import namedtuple


Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
data = [Point(2, 3, 4), Point(10, -100, -500), Point(3, 7, 11),
Point(2, 202, 1)]
print(sorted(data))
```

Как вы могли заметить при совпадении значений первого по счёту свойства происходит сравнение второго и т.д. пока не определится меньший элемент.
И ещё одна интересная особенность namedtuple. Если все свойства являются объектами неизменяемого типа, экземпляр может быть ключом словаря, элементом множества и т.п.

```
from collections import namedtuple
```

```
Point = namedtuple('Point', 'x y z', defaults=[0, 0, 0])
d = {
    Point(2, 3, 4): 'first',
    Point(10, -100, -500): 'second',
    Point(3, 7, 11): 'last',
}
print(d)

mut_point = Point(2, [3, 4, 5], 6)
print(mut_point)
d.update({mut_point: 'bad_point'})   # TypeError: unhashable type:
'list'
```

Точка mut_point была создана и ошибки нет. namedtuple допускает изменяемые типы для свойств. Но такой экземпляр перестают быть хэшируемым. Как результат ошибка при добавлении точки в качестве ключа словаря.
И конечно же стоит упомянуть о таком плюсе namedtuple как экономия памяти. Экземпляры занимают в памяти столько же, сколько и обычные кортежи.

Модуль array

А вот ещё один модуль, предоставляющий доступ к типу данных массив. Как вы помните список list является массивом ссылок на объекты. А сами объекты хранятся отдельно, вне массива. array из модуля array является классическим массивом. Внутри него можно хранить целые или вещественные числа, а также символы Unicode.

```
from array import array, typecodes

byte_array = array('B', (1, 2, 3, 255))
print(byte_array)
print(typecodes)
```

Первый аргумент — строковой символ код типа. Буква указывает на то какие данные хранятся в массиве и выделяет нужное число байт под данные.
Вторым аргументом передают последовательность для помещения в массив.
Переменная typecodes выводит все допустимые коды типа:

- 'b' — целое со знаком, 1 байт
- 'B' — целое без знака, 1 байт

- 'u' — Юникод-символ в 2 или 4 байта
- 'h' — целое со знаком, 2 байта
- 'H' — целое без знака, 2 байта
- 'i' — целое со знаком, 4 байта
- 'I' — целое без знака, 4 байта
- 'l' — целое со знаком, 4 байта
- 'L' — целое без знака, 4 байта
- 'q' — целое со знаком, 8 байт
- 'Q' — целое без знака, 8 байт
- 'f' — вещественное обычной точности, 4 байта
- 'd' — вещественное двойной точности, 8 байт

Массивы поддерживают методы списка list, поэтому использование их интуитивно понятно. Привыкать надо лишь к указанию хранимого типа данных.

```
from array import array

long_array = array('l', [-6000, 800, 100500])
long_array.append(42)
print(long_array)
print(long_array[2])
print(long_array.pop())
```

При этом массив не позволит добавить значение, если оно выходит за пределы диапазона, заданного кодом типа при создании. Так же будет поднята ошибка при несоответствии типа.

```
from array import array

long_array = array('l', [-6000, 800, 100500])
long_array.append(2**32)   # OverflowError: Python int too large
to convert to C long
long_array.append(3.14)   # TypeError: 'float' object cannot be
interpreted as an integer
```

Массивы array являются более экономичной по памяти структурой данных, чем списки list.

Задание

Перед вами несколько строк кода. Напишите что выведет программа, не запуская код. У вас 3 минуты.

```
from collections import namedtuple

Data = namedtuple('Data', ['mathematics', 'chemistry', 'physics',
'genetics'], defaults=[5, 5, 5, 5])
d = {
    'Ivanov': Data(4, 5, 3, 5),
    'Petrov': Data(physics=4, genetics=3),
    'Sidorov': Data(),
}
```

5. Модуль argparse

В финале поговорим о модуле argparse. Мы упоминали его, когда речь шла о запуске скриптов с параметром через sys.argv. Поговорим о том чем же argparse лучше argv. Спойлер — всем. Модуль argparse по сути надстраивается над sys.argv. Он генерирует справку, определяет способ анализа аргументов командной строки, сообщает об ошибках и даёт подсказки. Чтобы разобраться во всём перечисленном по традиции рассмотрим простой пример.

```
import argparse

parser = argparse.ArgumentParser(description='My first argument
parser')
parser.add_argument('numbers',      metavar='N',      type=float,
nargs='*', help='press some numbers')
args = parser.parse_args()
print(f'В скрипт передано: {args}')
```

💡 **Внимание!** Тут и далее до конца главы запускать файл будет из терминала ОС. Примерно так:

$ python main.py ...

💡 где многоточие - передаваемые скрипту аргументы

Запустим скрипт с несколькими значениями:

```
$ python3 main.py 42 3.14 73
```

На выходе получаем объект `Namespace(numbers=[42.0, 3.14, 73.0])`. Как это работает?

1. Создаём объект парсер при помощи класса ArgumentParser с первоначальными настройками экземпляра.
2. Добавляем в полученный экземпляр аргументы для парсинга через метод add_argument. Количество аргументов может быть любым. И каждый может содержать свои настройки.
3. Выгружаем результаты, переданные при запуске скрипта в терминале и обработанные парсером в виде объекта Namespace. Для этого вызываем метод экземпляра parse_args.

Прежде чем разобрать каждый пунктов подробнее запустим скрипт ещё пару раз: с ключом --help и с каким-нибудь текстом.

Ключ --help или -h

После создания экземпляра парсера и задания ему аргументов формируется справочный текст. argparse добавляет ключи --help (длинная версия) и -h (короткая версия) автоматически. Другие ключевые параметры мы можем создать сами, но о них чуть позже.

```
usage: main.py [-h] [N ...]

My first argument parser

positional arguments:
  N           press some numbers

options:
  -h, --help  show this help message and exit
```

Первая строка даёт общее представление о строке запуска. Ниже идёт текст, который мы указали в по ключу description при создании экземпляра. Далее получаем информацию о позиционных аргументах. В нашем случае это аргумент N (metavar='N') и подсказки к нему (`help='press some numbers'`). В конце идёт необязательные параметры, в нашем случае - автоматически сгенерированный вызов помощи.

Запуск с неверными аргументами

При попытке передать в скрипт Hello world! получим:

```
usage: main.py [-h] [N ...]
main.py: error: argument N: invalid float value: 'Hello'
```

При создании аргумента мы указали, что хотим получать целые числа (`type=float`). Парсер автоматически создал валидатор и сообщил о несовпадении типов. Заметьте, что при передаче целых чисел ошибок не было, но они были преобразованы к вещественному типу.

Создаём парсер, ArgumentParser

При создании экземпляра из класса ArgumentParser можно 13 различных аргументов. Но большинство из них имеют оптимальные настройки по умолчанию. Если что-то и стоит добавить, то дополнительное описание, которое выводится при вызове справки.

```
import argparse

parser = argparse.ArgumentParser(prog='average',
                                 description='My first argument
parser',
                                 epilog='Returns the arithmetic
mean')
...
```

- prog — заменяет название файла в первой строке справки на переданное имя,
- description — описание в начале справки
- epilog — описание в конце справки

Выгружаем результаты, parse_args

Метод parse_args может принимать на вход два аргумента:

- строку для анализа. По умолчанию это sys.argv
- объект для сохранения результатов. По умолчанию это класс Namespace. Класс наследуется от object, не имеет ничего лишнего, но добавляет удобный вывод ключей и значений, помещённых в него.

Изменять значения по умолчанию приходится крайне редко, почти никогда.

```
import argparse

parser = argparse.ArgumentParser(description='My first argument parser')
parser.add_argument('numbers', metavar='N', type=float, nargs='*', help='press some numbers')
args = parser.parse_args()
print(f'Получили экземпляр Namespace: {args = }')
print(f'У Namespace работает точечная нотация: {args.numbers = }')
print(f'Объекты внутри могут быть любыми: {args.numbers[1] = }')
```

В этом случае код говорит сам за себя. Прочитайте три нижние строки. Уверен, что вопрос не должно остаться.

Добавляем аргументы, add_argument

А теперь самое интересное. Между созданием парсера и чтением результатов надо добавить желаемые аргументы.
Перед нами пример функции для решения квадратных уравнений. Параметры a, b, c собираем в терминале.

```
import argparse

def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
        return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 * a)
    if d == 0:
        return -b / (2 * a)
    return None
```

```
if __name__ == '__main__':
          parser   =   argparse.ArgumentParser(description='Solving
quadratic equations')
      parser.add_argument('param', metavar='a b c', type=float,
nargs=3,
                          help='enter a b c separated by a space')
    args = parser.parse_args()
    print(quadratic_equations(*args.param))
```

Вызов: $ python3 main.py 2 -12 10

Первая строка превращается в имя свойства. Если она начинается с одиночного или двойного дефиса, параметр считается необязательным. Далее:

- metavar — имя, которое выводится с справке
- type — тип, для преобразования аргумента. Тип помогает контролировать передачу нужных значений.
- nargs — указывает на количество значений, которые надо собрать из командной строки и собрать результат в список list. Целое число указывает количество. Кроме этого можно передать символ “?” — один аргумент, “*” — все имеющиеся аргументы, “+” — все имеющиеся аргументы, но не пустое значение.
- help - вывод подсказки об аргументе.

Если вызвать справку для нашего кода, увидим дублирование в первой строке `usage: main.py [-h] a b c a b c a b c`

Необязательные аргументы и значения по умолчанию

Изменим наш парсер и добавим ещё несколько параметров к аргументам.

```
import argparse


def quadratic_equations(a, b, c):
    d = b ** 2 - 4 * a * c
    if d > 0:
         return (-b + d ** 0.5) / (2 * a), (-b - d ** 0.5) / (2 *
a)
    if d == 0:
        return -b / (2 * a)
    return None


if __name__ == '__main__':
          parser   =   argparse.ArgumentParser(description='Solving
```

```
quadratic equations')
            parser.add_argument('-a',   metavar='a',   type=float,
help='enter a for ax^2+bx+c=0', default=1)
            parser.add_argument('-b',   metavar='b',   type=float,
help='enter b for ax^2+bx+c=0', default=0)
            parser.add_argument('-c',   metavar='c',   type=float,
help='enter c for ax^2+bx+c=0', default=0)
    args = parser.parse_args()
    print(quadratic_equations(args.a, args.b, args.c))
```

Вызов: $ python main.py -a 2 -b -12
Теперь необходимо указывать ключи a, б и с при передаче значений. Но дополнительный параметр default позволяет отказаться от передачи. В этом случае значения будут взяты из параметра по умолчанию.

Параметр action для аргумента

И напоследок ещё один интересный параметр уже без квадратных уравнений. Речь пойдёт о параметре action.

```
import argparse

parser = argparse.ArgumentParser(description='Sample')
parser.add_argument('-x', action='store_const', const=42)
parser.add_argument('-y', action='store_true')
parser.add_argument('-z', action='append')
parser.add_argument('-i',    action='append_const',    const=int,
dest='types')
parser.add_argument('-f',   action='append_const',   const=float,
dest='types')
parser.add_argument('-s',    action='append_const',    const=str,
dest='types')
args = parser.parse_args()
print(args)
```

Вызов: $ python3 main.py -x -y -z 42 -z 73 -i -f -s
Параметр action принимает одно из [определённых строковых значений] и срабатывает при наличии в строке вызова скрипта соответствующего параметра.

- store_const — передаёт в args ключ со значением из параметра const
- store_true или store_false — сохраняет в ключе истину или ложь
- append — ищет несколько появлений ключа и собирает все значения после него в список
- append_const — добавляет значение из ключа в список, если ключ вызван.

- параметр dest переопределяет имя ключа в Namespace на своё. В результате несколько разных ключей при вызове скрипта имеют одно имя при оценке результата.

Пожалуй это всё о основных способностях модуля argparse.

Вывод

Мы заканчиваем курс работой в командной строке. Тем, с чего начинали на первой лекции. Немного символично, как замыкание большого цикла знаний. Надеюсь вам было интересно и познавательно. Впереди ещё много нового и неизведанного. Но вы достаточно глубоко погрузились в Python, чтобы претендовать на ступеньку junior разработчика.

На этой лекции мы:

1. Узнали о составе стандартной библиотеки Python.
2. Разобрались в настройках логирования
3. Изучили работу с датой и временем
4. Узнали ещё пару полезных структур данных
5. Изучили способы парсинга аргументов при запуске скрипта с параметрами

Домашнее задание

Обратитесь к официальной документации по стандартной библиотеке Python. Выберите любой раздел по вашему желанию. Почитайте о модуле, попробуйте запустить примеры из документации.