Методические материалы к вебинару
"Особенности организации практических занятий курса"

Горшков Сергей Сергеевич, ФКН ВШЭ, МНМЦ ВШЭ

В данном пособии изложены основные моменты, связанные с проведением семинаров по курсу «Основы программирования на Python». Основной упор сделан на непосредственно тематике занятий, последовательности изложения материала и аспектах, которые необходимо подчеркнуть в ходе ведения практических занятий. В конце представлен список тем, которые предлагается изучить обязательно на одном из семинаров (без привязки к теме занятия) и список дополнительных тем.

Общая цель семинаров (практических занятий)

Семинарские занятия дополняют on-line курс «Основы программирования на Python», представленный на платформе Coursera НИУ «Высшая школа экономики» (лектор Михаил Сергеевич Густокашин). Однако, как показывает практика, студенты часто не смотрят лекции перед соответствующим семинаром, и поэтому на семинарах стоит изложить тот материал, который предлагается в лекционной части. На Coursera есть множество задач, которые разделены на обязательные и тренировочные. Желательно, конечно, требовать решения обязательных задач (необходимо для получения сертификата), преподавателю же стоит прорешать эти задачи, чтобы быть в состоянии ответить на вопросы студентов. В курсе предлагается решить сто обязательных задач, но они достаточно простые, практикущий Python-программист может их прорешать часов за десять. На ранних стадиях изучения языка программирования (да и программирования в целом, Python для многих первый язык) очень важно набивать руку на решении простых задач. Для нетехнических специальностей это ещё более важно, поскольку программирование имеет ряд отличительных особенностей от других учебных дисциплин в образе мышления. Студентам может быть достаточно непривычно (и даже контринтуитивно) то, что в строгой технической дисциплине могут не соблюдаться базовые правила математики, например, связанные с вычислениями вещественных чисел с плавающей точкой. Нелинейности, такие как ветвления и циклы, также нетривиальны для понимания, в отличие от, например, линейных конструкций, с которых курс и начинается.

В рамках практической части курса важно не только научиться решать задачки, но и дать общее представление о программировании как инструменте, используемом в работе самыми разными специалистами. Стоит описать место

языка Python в современном программировании. Многие студенты ошибочно считают, что программисты много пишут код (например, при создании компьютерных игр), но это не самая большая часть современного рынка. В настоящее время множество разничных IT-профессий используют Python как один из основных языков программирования – это и веб-разработка (можно рассказать об отличиях back-end и front-end разработки), аналитика данных, машинное обучение и проч. Термин машинное обучение сейчас на слуху, и стоит сказать о том, что в машинном обучении язык Python является уже де-факто самым распространенным языком программирования в этой перспективной отрасли. Помимо этого, Python используется специалистами в биологии, медицине, физике, имеет обширную библиотеку и большое комьюнити. Многие менеджеры также должны владеть языком Python (и/или часто SQL), чтобы они могли самостоятельно проверять простейшие гипотезы.

Что касается непосредственно написания кода, надо заострять внимание на четырех деталях и напоминать о них на занятиях.

1) У Python есть две версии, код на которых пишется и поддерживается сейчас – это Python2 и Python3. Эти версии не совместимы друг с другом, и во время изучения курса необходимо рассказывать о каких-то аспектах, отличающихся во второй версии относительно третьей, которую мы и будем изучать.
2) Необходимо следовать стайл-гайдам (style-guides) – требованиям к стилю написания кода. В Python общепризнанным является PEP8. При прохождении очередной темы необходимо давать представление о том, как следует писать те или иные конструкции.
3) Важно не просто писать код, важно писать читабельный код, уметь разбивать программу на подпрограммы (функции и классы). Необходимо отдельное внимание уделить на минусы дублирования кода и необходимость задавать именованные константы (если в будущем придется менять код/константу, то достаточно будет сделать это в одном месте. Если менять код в нескольких местах, то можно забыть где-то поменять и потом долго искать ошибку).
4) Параллельно с языком знакомить слушателей с архитектурой компьютера, в частности, с устройством памяти. Объяснить слушателям зачем это нужно понимать и что знание того, как реализовано что-то внутри позволяет писать более производительный код. Это и отличает специалиста от человека, прошедшего пару онлайн-курсов и умеющего делать какие-то базовые в программировании вещи.

Организация домашних заданий

В качестве домашних заданий для студентов предлагается решение задач в системе Яндекс.Контест. Задачи разделены на 12 соревнований. Предлагается проводить семинар по тематике домашнего задания, после чего открывать соответствующее домашнее задание. Можно разобрать на семинаре пару задач из предстоящей домашней работы (или же из прошедшей в случае возникновения вопросов по сложным задачам). Предполагается, что для решения задач в очередном контесте (соревновании) можно использовать только средства языка и инструменты, изученные на соответствующем семинаре и ранее. Для тех, кто к началу курса уже владеет языком Python, подобное требование может показаться странным, потому что они могут решить задачу другим способом, который может быть более эффективным. Однако, следует решать простые задачи простыми методами, время для более сложных конструкций ещё придёт. Вполне возможно, что какую-то задачу через пару-тройку семинаров можно будет решить куда изящнее, и в рамках занятий можно обсудить это. Например, заменить циклы на списковые включения (list comprehension). Python – язык, в котором одну и ту же логику можно, как правило, реализовать несколькими способами (хотя это и не стало девизом языка, как у Perl). И с ростом знаний и навыков разработчика решения будут становиться короче, изящнее и эффективнее. Ближе к концу курса можно посмотреть на решения простых задач первых трех контестов и обсудить, как это можно было бы сделать с учетом новых знаний. Это покажет, какой путь мы прошли за время курса. Отдельно стоит сказать про решения в одну строчку. Иногда это выглядит красиво и понятно, а иногда наоборот ничего не понятно в длинной строке и становится write-only кодом. Стоит не забывать об ограничении на длину строки в PEP8, однако, решение задач в одну строку позволяет развивать навыки разработки и думать более нестандартно. Главное – не злоупотреблять этим, ведь код намного чаще читается, чем пишется. Об этой простой истине не стоит забывать.

В Высшей школе экономики для помощи преподавателям есть институт учебных ассистентов. Им не даём доступов администратора к контестам, однако, они могут помогать студентам с домашними заданиями, корректировать их код, отвечать на различные вопросы. Также в их обязанности входит проведение консультаций по запросам студентов и преподавателя.

Начало работы

Основные шаги по установке Python описаны в онлайн-курсе. Стоит описать основные альтернативы в выборе средства разработки. Это могут быть IDE, например, PyCharm. В PyCharm нужно показать возможности отладки программ (debug), это очень мощный инструмент, если научиться им пользоваться, что в принципе, нетрудно. Для анализа данных и визуализации удобно использовать Jupyter notebook. И, наконец, текстовые редакторы, такие как Sublime Text, или незабвенные Vim и nano. В любом случае, программа на Python – это просто текстовый файл, расширение .py не делает текст питоновской программой. Обязательно надо показать запуск программы через консоль (python file_name.py). Если установлены две версии питона на компьютере, то возможно необходимо писать python3. Запуск программ «по кнопке» может сформировать неправильное представление о том, как исполняется код программы. Надо обязательно показать и запуск команд в интерпретаторе (либо открыть в терминале, либо есть Python Console в PyCharm) и построчное исполнение программ в интерпретаторе. В качестве первой программы разумно использовать `print("Hello world")`, а также использование интерпретатора питона как калькулятора. Если установлены и Python2, и Python3, то будет показательно написать программу `print "Hello world!"`, которая выполнится в Python2 и выдаст ошибку в Python3. Может возникнуть вопрос о том, как ищется команда python/python3 в терминале, стоит ответить про переменную окружения PATH. К слову, могут быть ошибки при установке питона из-за того, что не изменена переменная PATH. Можно показать полный путь к интерпретатору, в Unix-подобных операционных системах (в т.ч. MacOS) это делается командой which.

Очень важный инструмент программиста – его любимый поисковик. При возникновении вопросов или ошибок необходимо уметь правильно составить запрос к поисковой системе. Не стоит сразу же задавать вопрос преподавателю, наверняка у кого-то уже возникала эта проблема, тем более на ранних этапах изучения языка Python. Также стоит отметить, что самым частым естественным языком в программировании является английский, поэтому информация по поисковому запросу на английском языке, как правило, более исчерпывающая, чем на русском. Основной источник знаний о языке программирования – его документация (https://docs.python.org/3/), на русском языке есть хороший сайт https://pythonworld.ru/, ответы на вопросы часто есть на StackOverflow (есть версия сайта на русском, но там не всегда качественный перевод и, как кажется, освещено меньше вопросов).

Далее последует описание материала, необходимого для изложения на семинарах в разбивке по темам, соответствующим темам домашних заданий в Яндекс.Контесте. На первом семинаре необходимо познакомить студентов с тестирующей системой и показать решение одной из задач с отправкой кода, а также кратко рассказать об ошибках, которые могут возникнуть в тестирующей системе.

1. Арифметика

На первом семинаре рассматриваем понятия переменной и целочисленную арифметику. Можно начать с философской фразы о том, что программирование в некотором роде есть перекладывание переменных из одной в другую по некоторым правилам. Переменная – есть самая базовая абстракция в программировании. Первым делом показываем использование питона как простейшего калькулятора целых чисел. Однако есть вопрос – куда сохранять результат вычислений? Тут на помощь и приходят переменные. Проговариваем требования к названиям переменных – это последовательность букв, цифр и знаков нижнего подчеркивания, начинающаяся не с цифры. Такие ограничения есть, как правило, во многих популярных языках программирования. Переменные должны носить информативные названия, но для простейших программ это неважно. В python3 в качестве названий переменных можно использовать русские буквы, но делать так не следует (можно попугать различными кодировками и непонятными символами, которые встречаются при открытии файлов с другой кодировкой в Windows, например). Рассказываем об основных арифметических операциях, не забывая о том, что деление может быть целочисленным и нецелочисленным – `//` и `/` (и сказать об отличиях от Python2), а также об остатке от деления по модулю `%`. В Python есть специальный оператор возведения в степень `**`, о котором тоже стоит сказать, равно как и о том, что в питоне нет ограничения на длину целых чисел. Не забываем про унарные `+` и `—`.

Далее, рассказываем на пальцах об устройстве памяти и ссылок в python. Показываем команду `id()`, и говорим о том, что это некий адрес в памяти компьютера, где размещена переменная. Для продолжения разговора можно взять программу:

```
>>> a = 1
>>> b = 1
>>> id(a), id(b)
(4486386816, 4486386816)
>>> b = 2
>>> id(a), id(b)
(4486386816, 4486386848)
```

Объясняем, почему сначала переменные хранились по одному адресу, а потом стали по разным. Упоминаем про модель Copy-on-Write, её преимущества и счётчик ссылок в языке Python. У обучающихся, писавших ранее на других языках типа C++ может возникнуть вопрос про освобождение памяти. Можно сказать про garbage collector и автоматическую сборку мусора.

Следующий очень важный вопрос связан с представлением целых чисел в памяти компьютера. Сначала обсуждаем позиционные и непозиционные системы счисления, просим привести пример непозиционной (римская). Показываем как минимум два алгоритма перевода целых положительных чисел в двоичную систему (делением в столбик и разложение по степеням двойки). Далее, упоминаем часто встречаемые 8-ричную и 16-ричную системы счисления и мотивацию их использования – читабельность. Про big-endian/little-endian структуры лучше не говорить, дабы не путать обучающихся. Это представление в двоичной системе называется прямым кодом, в нем хранятся положительные числа (для любителей C/C++ можем отметить, что числа в Python являются знаковыми. Также есть обратный код (побитовая инверсия) и дополнительный код (инверсия и +1), в котором хранятся отрицательные числа. Показываем примеры и убеждаемся, что это так (например, переводим в двоичную систему число N и число –N и складываем. Должен получиться 0). Теперь рассказываем о битовых операциях в языке Python и их использовании (побитовое И – &, побитовое ИЛИ – |, побитовое исключающее ИЛИ (XOR) – ^, а также побитовая инверсия ~). Не забываем и про битовые сдвиги >>, <<, говорим об их связи с умножением и делением на степень двойки. Можно построить таблицу истинности, рассказать о том, почему операции так называются. Для демонстрации хранения отрицательных чисел в памяти компьютера может быть крайне полезна следующая демонстрация:

```
>>> a = 4567
>>> ~a
-4568
```

Очень важны операторы сокращенного присваивания `+=` и т.п., предпочтительнее использовать `a += 5`, нежели `a = a + 5`.

Начнем разговор о типах и об утиной типизации в Python. Вскользь можно упомянуть про то, что это одна из причин, почему программы на Python уступают по скорости многим языкам со строгой типизацией. Тип для целых чисел – `int`. Есть ещё логический тип – `bool`, он принимает два значения – `True` и `False`. Ненулевые значения интерпретируются как `True`, нулевые – `False`. На примере

```
>>> a, b = 45, 0
>>> bool(a), bool(b)
(True, False)
```

можно показать, что, во-первых, возможно множественное присваивание в языке Python, и во-вторых, приведение типов работает так, как мы и описали.

Далее рассказываем о сравнении элементов в Python – шесть операций. В Python, в отличии от многих других языков, можно делать двустороннее сравнение:

```
>>> a = 5
>>> 1 <= a <= 4
False
>>> 6 >= a >= 1
True
```

Несмотря на то, что все ненулевые значения интерпретируются как `True`, результаты сравнений будут следующими (как будто сделали сравнение с `int(True)`):

```
>>> 1 == True
True
>>> 2 == True
False
>>> 2 > True
True
>>> True >= -1
True
```

Наконец, в завершение семинара рассказываем про специальный тип None и как можно смотреть тип.

```
>>> a, b, c = 4, False, None
>>> type(a), type(b), type(c)
(<class 'int'>, <class 'bool'>, <class 'NoneType'>)
```

None – это не значение, а отсутствие значения. Необходимость подобного мы поймём в следующих семинарах. Отметим, что при сравнении None с помощью обычных операций сравнения будет появляться ошибка. Проверку на None стоит делать только с помощью `is None / is not None:`

```
>>> a, b, c = 0, False, None
>>> a is None, b is not None, c is None
(False, True, True)
```

В Python2 было странное поведение для сравнения с `None`: `None` было меньше любого целого числа (и `None` можно было сравнивать с числами обычными операциями сравнения соответственно). Однако при работе с Python2 не следует этим пользоваться НИКОГДА.

Итак, мы изучили три простейших типа и можем уже решить немало задач. Напомню, в первом контесте запрещаем пользоваться условными операторами, циклами и прочим ещё неизученным. Разумеется, стоит рассказать про то как считать строку и привести её к целому числу – `int(input())` и напечатать числа – `print(arg1, ...)`

2. Условный оператор

На втором занятии речь пойдёт об условном операторе. Первый написанный нами код был линейным, команды выполнялись друг за другом. Теперь же часть программы может выполняться в зависимость от того, выполняются ли некоторые условия. Говорим про обычную форму `if`, `if-else`, потом добавляем `if-elif-else` с простейшими условиями. Стоит проговорить тонкий момент с инициализацией, почему так нельзя объявлять переменные (если надеются, что у `a` будет значение по умолчанию, но нет):

```
>>> if False:
...     a = 5
...
>>> a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'a' is not defined
```

Так подводим к тернарному оператору, который удобно использовать в подобных кейсах, например,

```
>>> b = False
>>> a = 5 if b else 0
```

Переходим далее к не-`bool` условиям. Для изученных типов `0, False, None` интерпретируются как ложь (условие не выполнено), остальное – как правда (условие выполнено). Однако, не стоит пользоваться тем, что None – ложь, всё равно стоит проверять через `is None/is not None`, часто бывает важно отличать `False` от `None`. Условия можно объединять в логические цепочки с помощью `and`, `or`, а также добавлять на условие отрицание с помощью `not`. Стоит написать несколько примеров подобных условий. Также сказать о приоритете логических операций (наиболее приоритетно `not`, потом `and`, потом `or`). Говорим о том, что скобки позволяют группировать выражения, рассматриваем несколько примеров, например,

```
>>> a, b, c, d, e, f = 9, 0, None, -34, 4, True
>>> (not a and d and e) or b or (not c and f)
True
```

Обсуждаем какие скобки нужны (конкретно здесь никакие) и что тут лучше поменять (проверку на `None` надо сделать явной, например), написать несколько сложных `if-elif-else` для тренировки. Самое время сказать про приоритет операций языка Python (например, https://tirinox.ru/python-op-precedence/). Кстати, если мы хотим пропустить сделать пустую ветку в условном операторе (т.е. чтобы при истинности условия не выполнялось ничего, но необходимо использовать конструкцию `pass`.

На протяжении всего семинара использовались отступы (хотя кто-то слышал, что используют табы – знак табуляции). Момент про 4 пробела стоит особенно проговорить. А также не забываем, что условные операторы могут быть вложенные и показать примеры. Немного искусственно, но для демонстрации сойдёт:

```
>>> a = 9
>>> if a is not None:
...     if a < 0:
...         print("a < 0")
...     elif not a:
...         pass
...     else:
...         print("a > 0")
...
a > 0
```

Здесь можно обсудить, что эффективнее – сделать так, или убрать внешнее условие, и в каждое условие на нынешнем втором уровне вложенности добавить `a is not None and`. Код должен быть оптимальным!

Наконец, можно немного порисовать стрелочки переходов и упрощенно рассказать, как это работает внутри, и попросить написать подобный псевдокод для `if-elif-else`:

```
if expr:                    expr
    body_if          l_start:
else:                        ifFalse{expr} jmp l_else
    body_else                body_if
                             jmp l_end
                     l_else:
                             body_else
                     l_end:
```

Здесь `l_...:` – метки, на которые может производиться переход, `jmp` – переход на метку, `ifFalse{expr} jmp` – переход на метку, если ложно условие `expr`, `body_` – некоторая последовательность команд.

3. Цикл while

Цикл while – простейший цикл с предусловием (на каждой итерации сначала проверяется условие, потом выполняется тело цикла). Можем описать эту последовательность действий на том же псевдоязыке:

```
while expr:          l_start:
    body_while               expr
                             ifFalse{expr} jmp l_end
                             body_while
                             jmp l_start
                     l_end:
```

Очень важными инструкциями в циклах являются ключевые слова `break` и `continue`. `break` позволяет немедленно выйти из цикла, а `continue` – перейти на следующую итерацию цикла, не выполняя *в данной итерации* код в теле цикла после `continue`. Можно написать несколько простеньких циклов с разными условиями, `break` / `continue`, в том числе бесконечный цикл. Можно попросить обучающихся показать, почему эквивалентны циклы:

```
while a > 0:          while True:
    a -= 1                a -= 1
    print(a)              print(a)
                          if a == 0:
                              break
```

В Python у `while` есть `else` часть, в которую переходит управление после выхода из цикла, если выход из цикла произошел естественным путем (не по `break`). Это удобная особенность языка.

Итак, теперь мы умеем обрабатывать в цикле различные события. Давайте решим задачу, в которой у нас будет происходить ввод со стандартного потока ввода некоторой последовательности натуральных чисел (ввод заканчивается пустой строкой), в которой одно число встречается один раз, а остальные – два. Найти это одинокое число (задача с собеседований). Для решения задачи нам ещё нужно знать, как выглядит пустая строка – `""`.

```
xor = 0
while True:
    a = input()
    if a == "":
        break
    xor ^= int(a)
print(xor)
```

Идея алгоритма проста – если два числа равны, то их XOR == 0. Тогда XOR всех чисел, встречающихся два раза, будет равен 0.

4. Вещественные числа и строки

В рамках этого семинара познакомимся с ещё двумя стандартными типами языка Python. Начнём с вещественных чисел. Начинаем разговор с представления чисел с плавающей точкой в памяти компьютера – в виде знакового бита, порядка и мантиссы. Для начала можно перевести в подобное представление с размерами, скажем, как в типе float языка Си (1 бит – знак, 8 – порядок, 23 – мантисса). Сначала для числа, дробная часть которого конечна в двоичной записи, например, 21.25, а потом – бесконечна, к примеру, 0.2. Обсудим, какие могут быть проблемы с точностью из-за подобного представления чисел. Кстати, в языке Python целая часть отделяется от дробной точкой и только точкой. Вместе с тем, если какая-то часть равна нулю, то она может отсутствовать. Например, записи ниже корректны:

```
>>> a, b = .9, 3.
>>> print(a, b)
0.9 3.0
```

Также рассказываем и об экспоненциальной форме записи числа

```
>>> c, d = 1e3, 1E-3 # большая или маленькая e — неважно
>>> print(c, d)
1000.0 0.001
```

И приведём пример ошибки, связанной с конечностью мантиссы. Этот момент стоит осознать, потому что на первый взгляд это контринтуитивно:

```
>>> a, b = 1e23, 1e-23
>>> a + b == a
True
```

Сравнивать вещественные числа необходимо не напрямую, а несколько хитрее. Считаем, что два числа равны, если модуль их разности (функция `abs()`) меньше некоторого маленького числа ε. Выбор этого маленького числа ε, как правило, зависит от величин сравниваемых чисел. Есть несколько подходов к выбору этого числа – константа (например, ε=`1e-9`), или число, зависящее от порядка сравниваемых чисел – например, одно из этих чисел, умноженное на `1e-9`.

```
>>> abs(a-b) < a*1e-9
```

Для увеличения точности можем воспользоваться модулем `decimal`. Однако, его использование не всегда удобно и не функции, работающие с числами, будут работать и для объектов класса `Decimal`. Пока это обучающимся может ни о чем не говорить, но идея, что этот модуль – не панацея, должна усвоиться.

Тип вещественных чисел – float. Можно конвертировать в него различные другие типы (если значения подходящие). Можно приводить к целым числам – при этом будет отбрасываться дробная часть, а также округлять:

```
>>> a, b = 1.33, -1.67
>>> print(int(a), int(b), round(b))
1 -1 -2
```

С точки зрения конвертации в логический тип, 0. – это ложь, остальное истина.

Теперь поговорим о строках. Строки записываются в одинарных или двойных кавычках. Если хотим, чтобы внутри строки были переводы строки, то обрамляем их тремя одинарными или тремя двойными кавычками с каждой стороны. Введём понятие изменяемых (mutable) и неизменяемый (immutable) типов. Все рассмотренные нами выше типы, как и строки, являются неизменяемыми, т.е. когда мы изменяем значение переменной этого типа, то создается новая переменная, равная измененному значению (и она уже записывается в старую переменную). Если значение переменной изменилось, то оно будет занимать другое место в памяти. Если нет – то не изменится.

```
>>> a = 1
>>> id(a)
4342531200
>>> a += 1
>>> id(a)
4342531232
>>> a += 0
>>> id(a)
4342531232
```

Что же это означает применимо к строкам? При любом изменении строки создается как бы её копия, она модицифицируется и становится новым значением строки. Вместе с этим нельзя присвоить отдельному символу строки новое значение. Обращение к элементу по индексу с помощью `[]`:

```
>>> s = "Hello world!"
>>> s[0]
'H'
>>> s[0] = "e"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Можно узнать длину строки с помощью метода `len`. Индексация происходит с начала строки, если индекс превосходит длину строки, то возникает ошибка. Можно индексироваться с конца строки с помощью отрицательных индексов. С этим бывает путаница, поэтому можно объяснить следующее тождество

`s[-N] == s[len(s)-N]`. У строки можно брать срезы – некоторые подстроки, – синтаксис которых выглядит как `s[start:stop:step]`, где каждый из агрументов может быть опущен. Здесь следует показать несколько примеров и дать подумать обучающимся как с помощью среза, к примеру, развернуть строку (`s[::-1]`). Строки можно складывать и умножать на целое число (столько раз строка повторится). Тип строки – str, в него и из него можно конвертировать данные в ранее изученные нами типы. Стоит сделать обзор основных методов строк (https://docs.python.org/3/library/stdtypes.html#string-methods), сложно сказать на каких методах остановиться, все крайне важны. Можно отметить, что при вводе строки хорошо бы делать `.strip()`, чтобы убрать все пробельные символы в начале и конце (на всякий случай). К слову, `.strip()` без аргументов удаляет все пробельные символы, что очень удобно. Пока стоит просто отметить, что функции вызываются от строки через точку, можно сказать, что это методы класса, если кто-то знает.

В питоне есть множество средств для реализации форматных строк (https://docs.python.org/3/tutorial/inputoutput.html#fancier-output-formatting). Стоит рассмотреть основные способы – format и так называемые f-строки, появившиеся в Python 3.6 – очень мощный инструмент для форматирования.

Рассказываем об escape-последовательностях(спецсимволах), нам важнее всего \n, \t, с которыми можно столкнуться, а также \r и его использование при переводе строки в Windows. Если не хотим, чтобы эти символы так интерпретировались, то необходимо использовать r-строки (raw-string, сырые строки), где эти последовательности воспринимаются просто как набор символов. Символы можно экранировать, например, если в строке есть кавычка, то ставим перед ней обратный слэш и она остаётся кавычкой.

```
>>> s = "Hello "HSE""
  File "<stdin>", line 1
    s = "Hello "HSE""
                  ^
SyntaxError: invalid syntax
>>> s = "Hello \"HSE\""
>>> print(s)
Hello "HSE"
```

Теперь мы начинаем догадываться, почему ставить табы в отступах – плохая идея. Символ \t – горизонтальная табуляция – может быть равен 2, 4 или 8 пробелам, или, вообще говоря, любому числу пробелов в зависимости от текстового редактора, платформы и проч. А питон требует четыре пробела отступа, но во многих редакторах табы заменяются на пробелы и размер таба можно установить в настройках самостоятельно.

5. Функции и рекурсия

До этого момента мы писали весь код в рамках одной программы, разбавляя линейный код условными операторами и циклами. Пришло время начать выделять подпрограммы из нашей программы. В питоне они называются функциями. Они необходимы для решения многих задач:

- Декомпозиция программы на части, каждая из которых реализует конкретный функционал и имеет значащее название
- Возможность переиспользования отдельных функций
- Уменьшение дублирования кода
- Упрощение отладки и тестирования

Рассказываем общий синтаксис функций, возможность создания параметров по умолчанию. Стоит отдельно проговорить, что параметр может иметь значение по умолчанию, если только все параметры справа от него имеют значения по умолчанию, показать это на примере. Показать вызов функции с именованными и неименованными аргументами, например,

```
>>> def f(a, b, c=2, d=4, e=5):
...     print(a + 10*b + 100*c + 1000*d + 10000*e)
>>> print(f(7, 8, e=9, d=1))
91287
None
```

Кстати, как мы видим, функция может ничего не возвращать с помощью `return`. В этом случае возвращаемое значение будет `None`. Из функции можно вернуть несколько аргументов и распаковать их при возврате из функции (количество возвращаемых значений должно равняться числу переменных). Можно использовать специальную запись `_`, которая означает пропуск значения (не совсем так, но в этом кейсе сойдет).

```
def g():
    return 1, 2, 3
a, _, c = g()
```

Показать, что можно объявлять функции внутри функций, можно присваивать функции в переменные, передавать функции аргументом функции и проч. Отдельно стоит уделить внимание неименованным функциям – lambda-функциям. Часто забывают, что их можно использовать как функции от нескольких переменных, например,

```
>>> h = lambda x, y : x**y
>>> h(3,4)
81
```

Рассмотрим известную нам функцию `print`. Она может принимать два именованных агрумента – `sep` (какой строкой разделять элементы внутри

одного `print`, по умолчанию пробел) и `end` (какую строку выводить после `print`, по умолчанию перевод строки). Эти аргументы можно переопределять, изменяя формат вывода, что крайне удобно. При вызове функций часто пишут `*args, **kwargs` – это наборы неименованных и именованных аргументов, но об этом мы поговорим позже.

Рекурсивный вызов функции – это вызов, при котором функция вызывает сама себя. У рекурсии ВСЕГДА должно быть условие выхода из рекурсии. Напишем решения стандартных задачек на рекурсию – это вычисление чисел Фибоначчи и нахождение наибольшего общего делителя. После перепишем этот код без рекурсии и поговорим о том, что же происходит в случае рекурсии и чего стоит опасаться. Как известно, при вызове функции на стеке вызовов будет создаваться фрейм, в который будут помещаться аргументы функции, адрес возврата, локальные переменные (если кто-то спросит, вызывающая или вызываемая функция это делает, лучше всего ответить, что зависит от соглашения о вызовах). При рекурсивном вызове под каждый вызов на стеке будет выделяться очередной фрейм и место на стеке может закончиться. В питоне установлено ограничение на максимальную глубину рекурсии (которую можно поменять, но лучше не надо). Можно сказать о том, что в памяти компьютера есть стек и куча, они находятся в одном адресном пространстве и заполняются навстречу друг другу. В куче выделяются, как правило, большие участки памяти под динамические переменные и им нужно много места (упрощение, но в целом отражает картину мира). Так что каждый вызов функции сопряжен с расходами памяти на стеке, и рекурсии по возможности стоит избегать.

Помимо функций, в питоне есть генераторы. Если функции при return завершают своё исполнение, то генераторы с помощью yield возвращают значение и продолжают исполнение.

```
def gen(n):
    i = 0
    while i < n:
        i += 1
        yield i
>>> g = gen(2)
>>> next(g)
1
>>> next(g)
2
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

`StopIteration`

Когда значения в генераторе закончились, выдается ошибка `StopIteration`. В генератор можно передавать значения с помощью `yield from`, а не только возвращать, можно показать соответствующий пример.

В модуле `functools` есть очень удобные инструменты для работы с функциями, например, `partial`. Чтобы обратиться к этому имени надо сделать `from functools import partial`. С помощью `partial` мы можем создать функцию, которая будет являться другой функцией с какими-то зафиксированными параметрами. Например, известная нам функция `int()` на самом деле может принимать ещё и именованный аргумент – основание системы счисления, в которой находится число, если первым аргументом является строка. Определим функцию `int2`, которая позволит переводить строку, в которой записано число в двоичном формате в десятичное число:

```
>>> int2 = partial(int, base=2)
>>> int2("1100")
12
```

Наконец, пару слов о декораторах. Декораторы позволяют дополнить функциональность функции без изменения её кода. Декоратор есть функция, принимающая на вход функцию и возвращающая функцию.

```
def decorator(func):
    def wrapper():
        print("Hello")
        func()
        print("world")
    return wrapper

@decorator
def privet():
    print("privet")

>>> privet()
Hello
privet
world
```

То есть, указывая имя декоратора после символа `@` перед объявлением функции, можем дополнить её функциональность, не изменяя её саму. Этот же декоратор можно применить и к другим функциям. Декораторы могут использоваться, например, для измерения времени или логирования действий. Декораторы бывают параметрическими, позволяя передавать параметры в оборачиваемую функцию.

6. Цикл for

Цикл `for` позволяет перебирать все элементы в какой-то последовательности, по которой можно итерироваться. Таковыми являются, например, строки и генераторы, причем итерирование по генератору заканчивается, когда в нем заканчиваются элементы (цикл `for` обрабатывает возникающую ошибку). Синтаксис цикла `for` следующий – `for elem in iterable:`. В переменной `elem` на i-й итерации цикла будет находиться i-й элемент iterable-объекта. На самом деле, итерирование по объектам происходит точно так же, как и в случае с генератором, вызывая каждый раз метод функцию `next` от iterable-объекта. Строку мы может сделать iterable-объектом явно, по сути итерирование с помощью `for` есть то же самое:

```
>>> s = "abc"
>>> its = iter(s)
>>> next(its)
'a'
>>> next(its)
'b'
>>> next(its)
'c'
>>> next(its)
Traceback (...):
  File "<stdin>", ...
StopIteration
```

```
>>> s = "abc"

>>> for elem in s:
...     print(elem)

a
b
c
```

Также есть очень удобная функция enumerate, которая позволяет получать элемент и его индекс в цикле for:

```
>>> for idx, elem in enumerate("abc"):
...     print(idx, elem)

0 a
1 b
2 c
```

Функция `zip` позволяет совершать совместное итерирование по нескольким iterable-объектам. Также очень полезно рассмотреть функции модуля itertools (https://pythonworld.ru/moduli/modul-itertools.html). В цикле `for` можно использовать `break`, `continue`, `else`, ровно так же, как и в цикле `while`.

Далее необходимо рассмотреть цикл `for` с `range`, рассказать о версиях `range` с 1, 2 и 3 параметрами. Отличным упражнением является написание генератора `range` самостоятельно. К слову, в Python2 `range` создавал соответствующий список (или кортеж) и итерировался по нему (занимая память), а генерировать на лету можно было с помощью аналога `xrange`. О списках мы поговорим на следующем занятии.

7. Списки

Список (`list`) в Python – это динамически изменяемый массив, то есть последовательность элементов, возможно, разных типов (это стоит отметить), длина которой может изменяться. Необходимо рассказать всё о создании списков, операциях над ними. Можно изменять отдельные элементы списка, т.к. он является изменяемым типом (mutable). В остальном можно делать всё то же, что и со строками: узнавать длину, делать срезы, индексацию, итерирование. Можно добавлять элементы в конец с помощью `.append()` и удалять из конца с помощью `.pop()`. Есть очень удобный метод для разбиения строки на список элементов по разделителю – `split`, и объединение списка в строку через разделитель – `join`. Очень полезный подход, связанный с неизменяемостью строк. Если нужно собрать строку из множества подстрок (на каждой итерации цикла добавляем ровно одну подстроку), то лучше не добавлять новую подстроку много раз (т.к. тогда будет много лишних копирований строки), а добавлять всё в список, который потом объединить в строку с помощью `"".join`. **(1)**

```
>>> s = ''                  >>> l = []
>>> for i in range(10):     >>> for i in range(10):
...     s += str(i)         ...     l.append(str(i))

>>> s                       >>> s = ''.join(l)
'0123456789'                >>> s
                            '0123456789'
```

В модуле collections есть очень полезный класс deque, который является двусторонней очередью, у которой можно быстро добавлять элементы и в начало, и в конец.

Есть неизменяемый аналог списка – это кортеж (`tuple`), он занимает меньше памяти, является immutable-типом с соответствующими свойствами (не поддерживает присваивание по индексу) и обладает некоторыми другими замечательными свойствами. Его можно конвертировать в список и обратно. Однако, мы можем модифицировать кортеж, точно так же как и в случае со строкой, с копированием и перезаписью. Встречаются ошибки, связанные с созданием кортежа из одного элемента. `t = (1)` создает просто целое число `t`, `t = (1,)` – кортеж из одного элемента. Очень важно разобрать пример:

```
>>> l = [1, 2]              >>> t = (1, 2,)
>>> id(l)                   >>> id(t)
140272795126112             140272795245856
>>> l += l                  >>> t += t
>>> l                       >>> t
[1, 2, 1, 2]                (1, 2, 1, 2)
```

```
>>> id(l)
140272795126112
>>> l[1] = 4
>>> l
[1, 4, 1, 2]
```

```
>>> id(t)
140272794799088
>>> t[1] = 4
TypeError: 'tuple' object does
not support item assignment
```

Список/кортеж можно разворачивать в последовательность и запаковывать последовательность в кортеж с помощью *:

```
>>> def f(*args):
...     print(args[::-1])

>>> f(1, 2, 3, 4)
(4, 3, 2, 1)
```

Очень удобно выводить на печать не список, а последовательность элементов

```
>>> l = ['a', 3, None, 43, False]
>>> print(l)
['a', 3, None, 43, False]
>>> print(*l)
a 3 None 43 False
```

Список можно отсортировать либо с помощью `.sort()`, при этом изменится список, либо функцией `sorted()`, которая вернет отсортированный список, не модифицируя исходный. От списка можно вызвать также функции `min`, `max`, `sum`, которые вернут соответственно минимальный, максимальный элемент (если для элементов задано сравнение), и сумму (если элементы массива можно складывать с числами) соответственно.

Наконец, очень интересный способ создания списков – это списковые включения (list comprehension). Можно задавать список как последовательность, которая генерируется налету. Например, код из примера (1) перепишется следующим образом:

```
>>> s = ''.join([str(i) for i in range(10)])
>>> s
    '0123456789'
```

Очень лаконично, красиво и быстро.

Стоит написать очень много различных программ, которые используют эту идею. Например, посчитать сумму квадратов всех нечетных натуральных чисел до ста, не заканчивающихся нулём **(2)**

```
sum([i**2 for i in range(100) if i % 10])
```

Более того, здесь мы узнаём, что переменная _ на самом деле есть переменная без имени, куда можно загрузить некоторое значение и оперировать им в рамках выражения. Здесь нам неважна переменная `i`, и код будет следующим:

```
sum([_**2 for _ in range(100) if _ % 10])
```

8. Сортировка и линейный поиск

Как мы уже знаем, сортировка может производиться с помощью `sort` или `sorted`. У этих функций есть два параметра, `key` и `reverse`. `key` определяет значения, по которым будет производиться сортировка, `reverse` – будет ли она по убыванию (по умолчанию `reverse=False` и сортировка производится по возрастанию). По возрастанию чего? Что определяет `key`? Часто нам необходимо придумать свой способ сортировки, при этом мы не хотели бы менять исходные данные. В параметр `key` мы передаем ту функцию, которая будет применена к каждому элементу исходной коллекции и полученные элементы будут отсортированы по возрастанию. При этом используется тот факт, что кортежи сортируются сначала по первому элементу, при равенстве первого – по второму, и т.д. Например, пусть мы хотим отсортировать строки по длине, а при равенстве длины по второму символу:

```
>>> l = ["asd", "vfb", "de", "drt", "kjf"]
>>> sorted(l, key=lambda x: (len(x), x[1]))
['de', 'vfb', 'kjf', 'drt', 'asd']
```

Отсортировать массивы сначала по последнему элементу, а при равенстве по убыванию первого элемента (это то же самое что по возрастанию -1* на этот элемент):

```
>>> a = [[1, 2, 3], [1, 3, 2], [4, 5, 0], [3, 4, 5],
[3, 3, 3]]
>>> sorted(a, key=lambda x: (x[-1], -x[0]))
[[4, 5, 0], [1, 3, 2], [3, 3, 3], [1, 2, 3], [3, 4, 5]]
```

Необходимо рассмотреть ещё несколько различных примеров с нетривиальными функциями сравнения. Также рекомендуется обратить внимание на функцию `itemgetter` из модуля `operator`.

По умолчанию в Python используется метод сортировки Timsort (https://ru.wikipedia.org/wiki/Timsort). Рекомендуется рассмотреть как он работает, поговорить о различных алгоритмах сортировки и как они работают, какую имеют временную сложность. Казалось бы, один из самых простых и самых медленных алгоритмов сортировки – сортировка пузырьком – может работать быстрее всех остальных на некоторых наборах данных. Предлагается студентам вспомнить, в чём он заключается и предположить, на каких данных он будет выигрывать (это почти отсортированные данные, т.е. данные, в которых лишь небольшое число элементов неотсортированы. Яркий пример – пользовательские логи, отдельные элементы из которых могли с некоторой задержкой прийти на сервер и прийти позже, чем более новые события. В таком случае несколько проходов сортировкой пузырьком с легкостью справятся с задачей отсортировать значения).

В отсортированном списке очень сильным инструментом является бинарный поиск. С его помощью можно быстро найти элемент в списке, равно как и вставить элемент в отсортированный список так, чтобы тот не потерял свойства отсортированности. В Python необходимый функционал реализован в модуле bisect.

Вместе с тем, если список не отсортирован, то поиск может быть осуществлён лишь путем последовательного прохода по всем элементам списка в худшем случае. Можно посчитать количество элементов списка, равных заданному, с помощью `.count()`.

Поговорим и о другом методе сортировки, эффективном на специфичных данных – сортировке подсчётом. Если надо отсортировать, скажем, массив чисел от 0 до 99, то заведем массив на 100 элементов и будем считать, сколько раз встретился каждый из элементов. Потом выведем элементы в отсортированном порядке. Такой подход крайне эффективен, если в данных немного уникальных значений. В более общем виде эта задача решается полностью аналогично, но с помощью словарей, о которых речь пойдет дальше.

9-10. Множества и словари

В рамках данных материалов я объединил эти две темы, равно как и объединял их на своих семинарах, поскольку множество есть частный случай словаря, по сути. Создание множества решает следующую задачу – создать структуру данных, в которой за небольшое константное число операций можно будет находить элемент (добавлять элемент, удалять элемент). А словарь позволяет к каждому такому элементу как ключу (key) сопоставить некоторое значение (value), и хранить эту пару совместно. В начале семинара надо рассказать про то, как хэш-таблицы решают эту задачу, рассказать о коллизиях и методах борьбы с коллизиями. Особенное внимание уделить тому, что элементы неупорядочены в общем случае (для обучающихся, программировавших на C++, set – это упорядоченное множество, в Python set – неупорядоченное множество, у них разные асимптотики работы и разное строение). В хэш-таблице для нас незнакомым является понятие хэш-функции. Во время рассказа про хэш-функции необходимо чётко объяснить, почему только неизменяемые объекты являются хэшируемыми (если бы изменяемые объекты можно было положить в качестве ключей в хэш-таблицу, то при изменении элемента хэш поменялся бы, и либо мы по тому же хэшу найдем уже другое, измененное значение, либо мы измененный элемент положим в таблицу в соответствии с его новым хэшом, а старое значение мы никогда не

найдем уже). Таким образом, если мы хотим положить список в множество/словарь, то его необходимо сделать неизменяемым, т.е. привести к кортежу. В пару к множеству `set` есть неизменяемый тип `frozenset`, к нему стоит приводить множество, если хотим посчитать у него хэш. Основное свойство хэша заключается в следующем – равные объекты имеют равные хэши, обратно это вообще говоря, неверно. Отметим также, что все ключи в множестве и словаре уникальные, и если при создании множества какой-то ключ встречался несколько раз, то на множестве это никак не отразится.

По аналогии со списковым включениями, можем генерировать множества и словари.

```
{i**2 for i in range(100) if i % 10} # множество
{i: i**2 for i in range(100) if i % 10} # словарь
```

Необходимо рассмотреть все операции над множествами (https://pythonworld.ru/tipy-dannyx-v-python/mnozhestva-set-i-frozenset.html) и словарями (https://pythonworld.ru/tipy-dannyx-v-python/slovari-dict-funkcii-i-metody-slovarej.html).

Отдельно стоит рассмотреть крайне удобные методы словарей `.get()` и `.setdefault()`, которые позволяют обрабатывать случаи отсутствия значения в словаре, либо выдавая значение по умолчанию, либо записывая значение по умолчанию в словарь и возвращая его. Также предлагается подумать над тем, какой элемент удаляет метод `.pop()` у множеств и словарей (поскольку множества неупорядочены, то неизвестно).

Операция принадлежности ключа множеству/словарю – `in`, проверка того, что ключа нет – `not in`. Итерироваться по словарю удобно через `.items()`:

```
for k, v in d.items(): # k – ключ, v - значение
    print(k, v)
```

Очень много полезных классов есть в модуле collections (https://pythonworld.ru/moduli/modul-collections.html). `Counter` – позволяет создать счётчик, в котором будут храниться пары ключ и сколько раз он встретился. `OrderedDict` помнит порядок, в котором ему приходили элементы. Очень удобен `defaultdict`, позволяющий задать значения по умолчанию для значения (value) у словаря. Попробуем сделать некоторый аналог счетчика с помощью `defaultdict`:

```
d = defaultdict(int)
l = [1, 2, 3, 3, 2, 4, 4, 3, 4, 5]
for i in l:
    d[i] += 1
```

А что было бы, если бы здесь был обычный словарь?

11. Функциональное программирование

Функциональное программирование в Python3, по сути, представлено двумя функциями – `map` и `filter`. `map` принимает на вход функцию и iterable-объект, применяет к каждому элементу этого объекта функцию и возвращает iterable-объект. `filter` же принимает на вход функцию, возвращающую `True` или `False` и iterable-объект и применяет к каждому его элементу функцию. Если значение функции на объекте истинно, то он будет выдаваться `filter` в качестве элемента выходного iterable-объекта, иначе он не входит в результирующую последовательность. Обратите внимание, что если мы хотим получить на выходе список, то необходимо явно привести результат к нему.

```
>>> map(int, "123456789")
<map object at 0x7f93ce2294d0>
>>> list(map(int, "123456789"))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Перепишем пример (1) из пункта 7. Списки с помощью `map` и `filter`. Получили ещё более лаконичную запись:

```
>>> s = ''.join(map(str, range(10)))
>>> s
'0123456789'
```

А теперь пример (2) оттуда же:

```
sum(map(lambda x: x**2, filter(lambda y: y % 10, range(100))))
```

Как видим, в Python действительно можно написать одну и ту же логику разными способами. Предлагается посмотреть решения некоторых задач третьего контеста и обсудить, как их можно было бы переписать с помощью включений или же элементов функционального программирования.

Этими двумя функциями функциональное программирование не ограничивается. Например, функция `reduce` была в стандарте языка Python2, однако в Python3 она была оттуда выведена и помещена в модуль `functools`. Функция reduce позволяет применять последовательно операцию к певрым двум элементам, результату и третьему элементу, результату и четвертому и т.д., объединяя всё в одно значение. Это, как правило, не очень эффективно и сейчас редко используется. Можно снова посмотреть множество функций из модуля `itertools`, содержащего, в частности, `accumulate`. Это некоторый аналог `reduce`, который выдаёт все промежуточные накопленные суммы, а не только одно значение – результат.

12. Основы объектно-ориентированного программирования

Тема ООП в Python очень обширна, и в нашем курсе представлены только самые базовые аспекты. Начинаем семинар с обсуждения трех основных составляющих ООП – инкапсуляции, наследования и полиморфизма. Сразу же оговоримся о том, что полиморфизм в Python весьма специфичен (в питоне же утиная типизация!). Очень важно объяснить различие между классом и экземпляром класса, понятие абстракции в ООП. Первое, что необходимо запомнить – в питоне всё есть класс. Класс можем объяснить как набор некоторых полей и методов (функций, имеющих доступ к этим полям), позволяющий работать с этим набором полей и методов как единой сущностью. Согласитесь, когда мы создавали числа, мы и представить нем могли, что у них есть методы! (хотя на первом семинаре при вызове type() мы и видели, что типом является какой-то класс). Чтобы посмотреть все методы объекта, необходимо передать его в функцию dir().

```
>>> a = 1
>>> dir(a)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__', '__class__',
'__delattr__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__',
'__floor__', '__floordiv__', '__format__', '__ge__', '__getattribute__',
'__getnewargs__', '__gt__', '__hash__', '__index__', '__init__',
'__init_subclass__', '__int__', '__invert__', '__le__', '__lshift__',
'__lt__', '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__or__',
'__pos__', '__pow__', '__radd__', '__rand__', '__rdivmod__', '__reduce__',
'__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '__rshift__',
'__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__',
'__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__',
'__xor__', 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
>>> a.__add__(67) == a + 67
True
```

Все операции, которые мы использовали – есть применение методов к объекту, даже сложение есть применение метода. Естественно, это поведение мы можем переопределить, но об этом позже. Предлагается обсудить часто встречающиеся методы, которые есть и в этом классе. Вызов методов производится, как мы могли догадаться, производится через точку. `"".join(['1','2','3'])` – это применение метода `join` к пустой строке, а последовательность строк есть лишь аргумент этого метода. Давайте создадим свой новый класс и посмотрим на то, какие методы он уже имеет:

```
>>> class C:
...     pass
...
```

```
>>> dir(C)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__',      '__setattr__',      '__sizeof__',      '__str__',
'__subclasshook__', '__weakref__']
```

Обсудим эти методы и зачем они нужны. Может последовать вопрос – а как, например, найти название метода, который отвечает на взятие элемента по индексу, например? Ответом здесь будет – либо поискать в интернете, либо посмотреть методы у какого-нибудь класса, которому присуще такое поведение. Кажется, такой метод есть у списков и кажется, название метода `__getitem__` вполне отражает то, что мы ожидаем. Давайте проверим себя: видим, что у списка есть ещё и `__setitem__`, который, скорее всего, позволяет присвоить значение по индексу. В кортеже можно доставать элемент по индексу, а присваивать нельзя. Проверим методы у какого-нибуль кортежа и да, первый метод там есть, а второго нет. Полезно посмотреть, какими ещё методами отличаются эти классы. Давайте создадим объект класса `C`: `c=C()` – это происходит так и посмотрим методы экземпляра `c`.

Далее, рассмотрим метод `__init__`. Рассказываем про то, как происходит создание объекта (и что такое `__new__`), определяем `__init__` в своём классе. Здесь же рассказываем про `self` (для знающих C++ разумно упомянуть про this). Помимо полей определим методы класса. Показываем get/set подход и объясняем, почему в питоне он не особо применим (нет сокрытия полей в стандартном понимании). Реализуем класс комплексных чисел, переопределяя различные magic-методы (они же dunder методы – методы, обрамленные двумя нижними подчеркиваниями) и добавляя функциональность методами такими как модуль числа, угол (из экспоненциальной формы записи комплексного числа) – неплохой повод познакомиться с библиотекой `math`.

Можем немного пофилософствовать на следующую тему: класс есть пространство имен, с помощью которого можно создавать объекты. Поля класса имеют доступ ко всем полям класса. Можно показать пример, когда атрибут объекта перекрывает атрибут класса с тем же именем. Надо четко разделить поля класса и поля экземпляра (я здесь использую разную терминологию для одного и того же, чтобы обучающиеся привыкли, поистине каждый называет части класса и созданные единицы класса как хочет).

Выглядит уместным рассказать про локальные и глобальные переменные, можно посмотреть на то, что вернут функции `locals()` и `globals()`.

Рассмотрим отношение наследования: при нём мы имеем те же поля, что и родительский класс и можем добавить новые. Обязательно надо обсудить вызов `super()`– этот метод возвращает пространство, в котором есть все поля базового класса (базовых классов), но при этом ошибочно думать, что создается экземпляр базового класса (у него например, `__new__` не вызывается при `super()`). Обсудим множественное наследование. Пожалуй, стоит упомянуть про проблему ромбовидного наследования и MRO, но углубляться не стоит. Предлагается написать класс транспортного средства и определить его наследников – машину, велосипед и грузовик, например. Обсудить, можно ли выделить класс транспортного средства с мотором и как его встроить в иерархию наследования.

В рамках курса мы не рассматриваем, но, если останется время, можно обсудить методы класса, статические методы, абстрактные классы (не забываем, что чтобы сделать такой класс, нужно не только навесить соответствующий декоратор на метод, но и отнаследоваться от базового абстрактного класс, модуль `abc`).

Разное

В этот курс входит не всё, что хотелось бы рассказать. ОБЯЗАТЕЛЬНО на одном из семинаров нужно осветить следующие темы:

- Работа с файлами (обязательно `readline()`, `readlines()`)
- Контекстные менеджеры (в конце можно написать класс, который можно использовать в контекстном менеджере)
- Модульная архитектура
- Модули sys и os (в т.ч. аргументы командной строки и переменные окружения, чтение `for line in sys.stdin`)
- Многофайловые проекты (в т.ч. что есть .pyc файл, зачем нужен)
- Механизм исключений, написать класс собственное исключение
- Регулярные выражения, в том числе группировки

Также можно рассказать о библиотеках для анализа данных, для создания веб-приложений, работы с телеграм API. Можно написать телеграм-бота, это сейчас модно и совсем несложно в Python. Также Python предоставляет широкие возможности для создания графических приложений.

Успехов!!!