**B6**

**B6.1 Введение**

Правило STOP, мотивирующая бредятина.

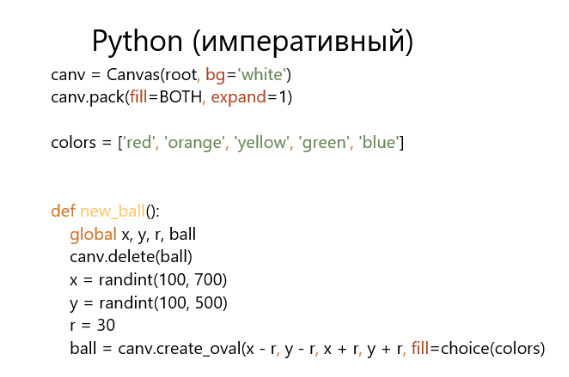
**B6.2 Парадигмы и принципы программирования**

Какой-то стиль программирования.

**Императивная парадигма программирования –** основана на том, что разработки пишет для компьютера инструкции, которым он следует. Эм… и что?

К нему относят: **процедурное, аспектно-ориентированное, объектно-ориентированное программирование.**

Вот некоторый код:



Он связан с библиотекой **Tkinter**.

<https://ru.wikiversity.org/wiki/%D0%9A%D1%83%D1%80%D1%81_%D0%BF%D0%BE_%D0%B1%D0%B8%D0%B1%D0%BB%D0%B8%D0%BE%D1%82%D0%B5%D0%BA%D0%B5_Tkinter_%D1%8F%D0%B7%D1%8B%D0%BA%D0%B0_Python>

И как минимум мне не нравится, что в этом написанном коде идет обращение к глобальной переменной **canv**. Я считаю, что такой параметр должен передаваться в функцию через аргумент.

Библиотека кстати нужна для добавления графических объектов, можно сделать графический интерфейс или простую игру.

В данном участке кода создается **холст –** объект библиотеки **tkinter**. С помощью него можно воссоздать скажем геометрические фигуры. В данном случае функция создает случайной точке холста круг со радиусом 30. Но не понимаю зачем в этом участке кода **global**. Во первых они не видно чтобы были объявленными до этого, ладно если объявлены, но в этой же функции им явно присваивают значения.

**Декларативная парадигма программирования**

В этой методике сначала не пишутся инструкции. Описывается проблема и ожидаемый результат.

К нему относят **функциональное** и **логическое** программирование**.**

****

**Это не код python, кажется это SQL.**

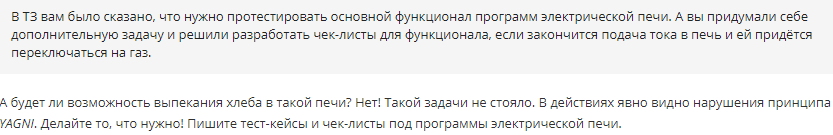
Имеются некоторые принципы программирования. Вот их список.



Если их разобрать.

**YAGNI –** Избегать сложных шаблонов, пакетов или фреймворков, которые не будут использоваться.

Ну по крайней мере первая часть этого принципа имеет смысл.



Ну как бы да. Но на самом деле зависит от случая. Тестировщик, должен попробовать все действия, чтобы выявить баги. Но конечно в рамках заложенного функционала.

**KISS –** писать простые конструкции. Избегать чрезмерного проектирования и усложнения.

**DRY –** или **DIE(Duplication is Evil) –** дублирование зло. Не повторяться. Иногда конечно бывает проще написать похожую функцию, чем еще переусложнять текущую. Так что это палка о двух концах. Но в целом да, незачем повторять один и тот же код.

**SOLID –** самый длинный акроним. Расшифровывать его нахер не буду. Но его смысл… В нем несколько смыслов.

На каждый класс должна быть возложена только одна обязанность.

Принцип открытости и закрытости. Код, который пишешь должен масштабируемым, но при внесение добавлений не должно требовать изменений уже написанного кода.

Каждый подтип класса должен дополнять, оригинальный тип, а не изменять его.

Принцип разделения интерфейса. Ни один клиент не должен зависеть от методов, которые он не использует. То есть не добавлять доп. функции в существующем интерфейс, вместо этого создать новый.



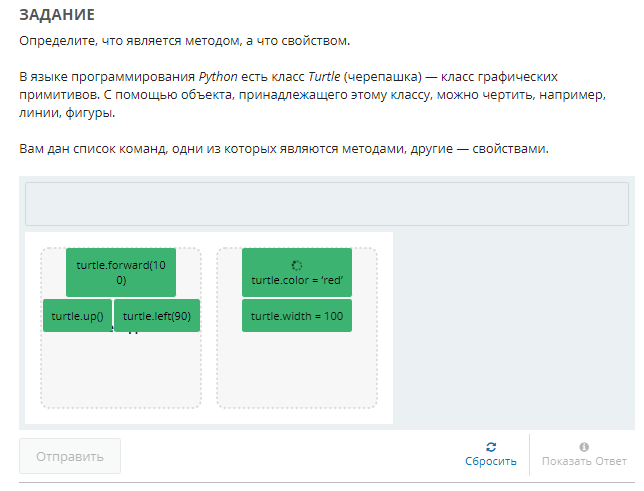
Первое типо kiss. Тупое и не интуитивное задание.

**B6.3. Концепция ООП и значимость принципов**

Основа ООП это два понятии – **классы** и **объекты.**

Говоря о **классе** подразумевается какими свойствами и поведением обладает **объект**.

А **объект** это собственно экземпляр со своим определенным набором свойств.



Черепашка, да помню.

**Принципы ООП.**

**Наследование –** каждый новый класс описывается на основе уже существующего (родительского), то есть перенимает свойства родителя, а также получает и свои методы.

**Абстракция –** выделение главных и наиболее значимых характеристик и наоборот, отбрасывание второстепенных.

**Инкапсуляция –** свойство системы, позволяющее объединить данные и методы, работающие с ними. Скрыть детали реализации от пользователя. Ограничения доступа к данным и возможностям их изменения.

В программировании есть свойства которые можно изменить и те, которые пользователь не может изменить.

**Полиморфизм –** свойство системы, позволяющее иметь множество реализаций одного интерфейса. Два разных авто, но оба могут поворачивать, хотя у одного есть к примеру гидроусилитель руля.

**B6.4. Классы**

****

Ну давай.

Вот скажем куча словарей со свойствами.



Чем больше их будет становится, тем большая жопа настанет с кодом. Ну и как это исправим?

Ну эту проблему можно решить с помощью классов. Кажется я как-то писал класс оружия и пистолета. Надо будет повторить наверно.

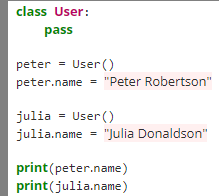
**Класс –** заготовка для создания объектов. После описания класса его можно использовать для создания любого количества объектов. Такие объекты называют **экземплярами класса.**

Синтаксис

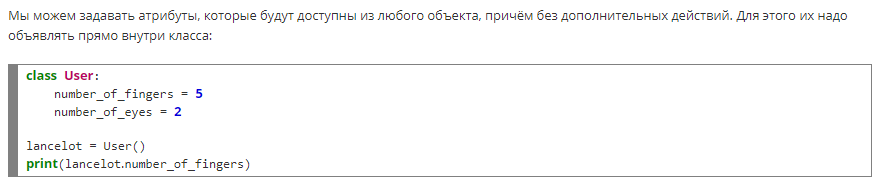


Для классов принято использовать структуру наименования **CamelCase.**

Чтобы создать экземпляры этого класса нужно использовать этот класс как функцию и записать ее в переменную.



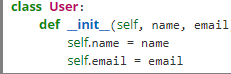
Как видно тут сразу и имя присваивается, хотя оно не прописано в качестве свойств класса. Но тем не менее это работает. Любопытно. А так их принято объявлять именно внутри объекта.



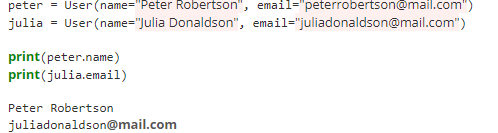
**B6.5. Магический метод \_\_init\_\_**

Без смешных картинок, но с волшебными словами пока что?

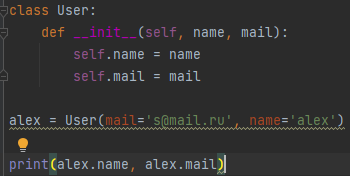
Конструктор класса. Кажется с помощью него создать класс у уже определенными полями.



**Этот метод вызывается каждый раз, когда мы создаём экземпляр.** Духота пошла. Теперь создавая экзепляр требуется обязательно передать эти параметры;



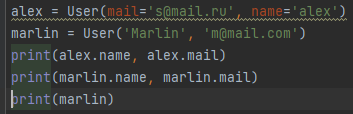
Они тут именные аргументы используют, значит и местами их можно поменять, да?



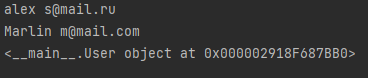


Да

А если без указания, то просто по порядку думаю.



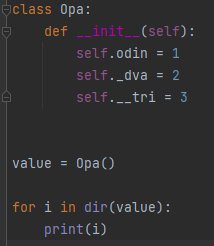
Да



В отличии от других языков в **Python нельзя создать жестко приваные поля с закрытым доступом.**

Есть только условное правило.. если название переменной начинается с двух нижних подчеркиваний, то она не предназначена для использования извне.

Так вот, кое что насчет **наименований**.



У этого класса есть 3 атрибута:

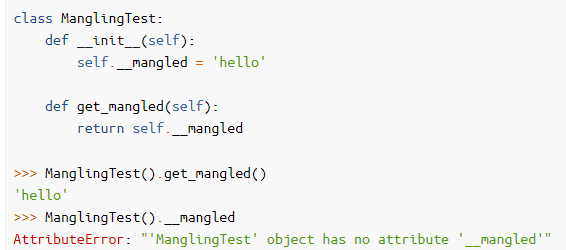
**odin, \_dva, \_\_tri.**

С помощью команды **dir** можно получить список всех атрибутов класса. И если присмотрется, то можно заметить, что атрибут с именем \_\_tri выглядит несколько иначе.

дандер преобразует свойство в объединение названия класса и свойства.

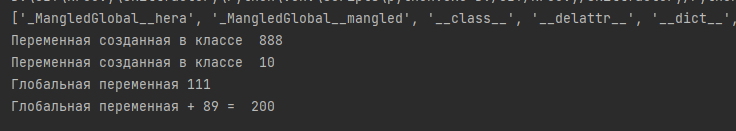
Это поведение свойство полностью прозрачно внутри класса.

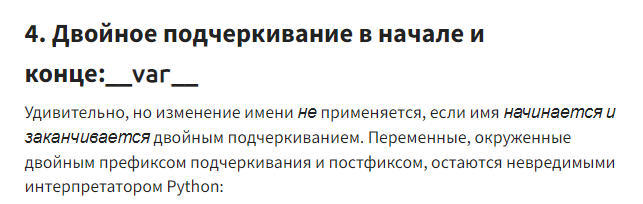


Но если обратится из вне, то будет ошибка, так как такого названия он не знает. Потому что оно изменено. По измененному названию можно обратится без проблем.

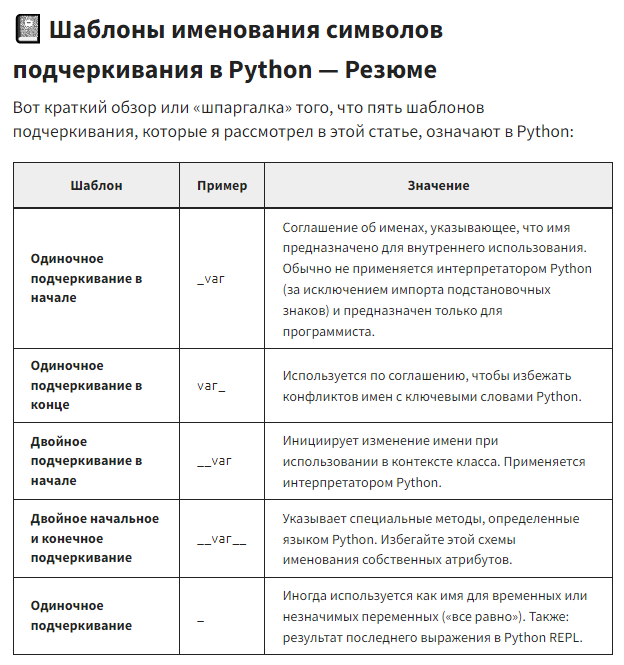
В совместительстве с глобальным переменными вообще трэш может произойти. Вот код, и можно сказать с помощью него я более менее понял это поведение.







Понятно… Оригинальное название остается. Но подобные конструкции принято использовать с конструкторами типо **\_\_init\_\_.**  **dunder методы. Их кто-то зовел волшебными,** но не будем до этого опускаться. Лучше избегать таких имен, т.к. в будущих версиях Python из-за таких программ могут появится конфликты.

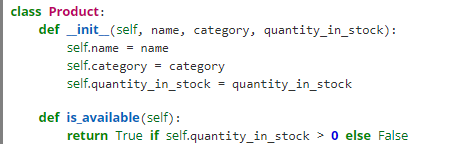


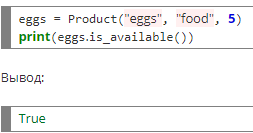
Это правда интересная статья.

**B6.6. Методы и функции**

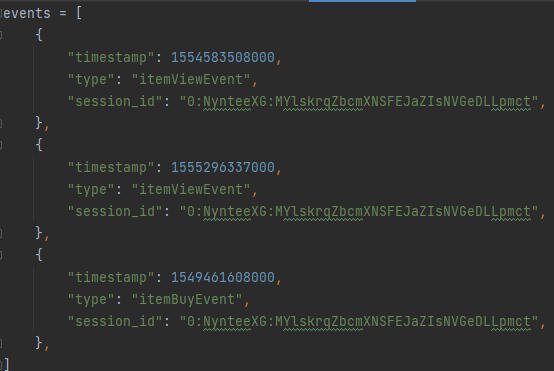
Забыл упомянуть очевидную вещь. **SELF == THIS.**

Метод – функция, в качестве первого параметра принимающая **self.**

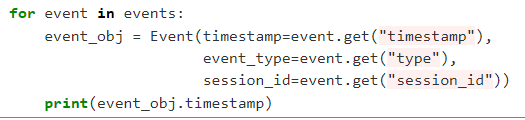
****

**self** в функции передается автоматически. Передавать его не нужно, я имею ввиду при вызове. 

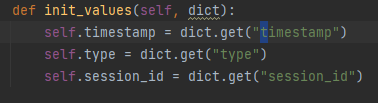
Парсить данные в объекте довольно удобно. Допустим есть массив со словарями.

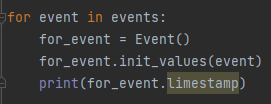


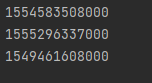
Чтобы обработать эти данные можно создать класс с такими полями. Но если передавать условно в цикле каждый раз эти поля при инициализации будет не очень красиво.



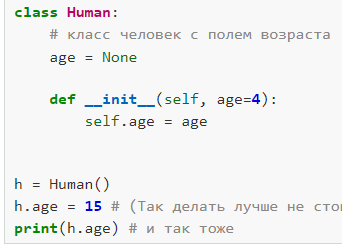
Вместо этого лучше задать этим полям при инициализации значения по умолчанию, и создать метод, который будет эти поля инициализировать просто принимая словарь. В таком случае цикл и создания объекта класса будет куда проще.





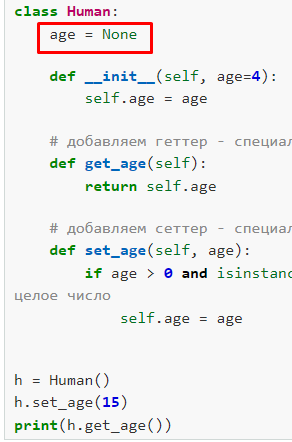


Далее идет речь об инкапсуляции. А если конкретно почему нельзя делать так:

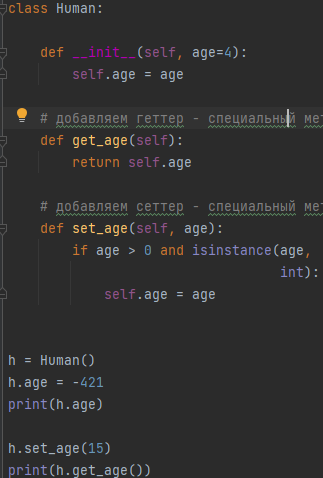


Если кратко, то потому что это небезопасно, к примеру так можно задать отрицательный возраст или вообще из символов.

Конечно ладно… Но зачем вы тут дважды возраст инициализируете глупые!



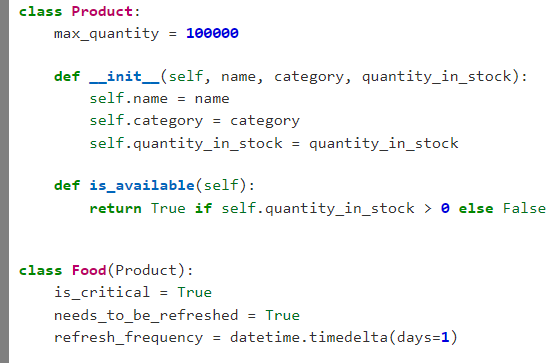
Ну практика на самом деле не плохая, но это никак не защищает от того, что кто-то без сеторов и геторов попробует обратится к этим полям.



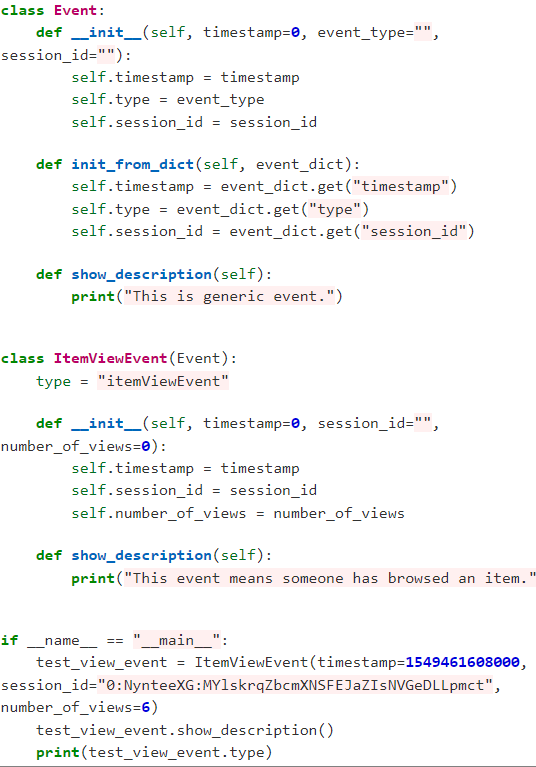
Более подробно к инкапсуляции вернемся когда речь зайдет о декораторах класса. А пока имеем, что имеем.

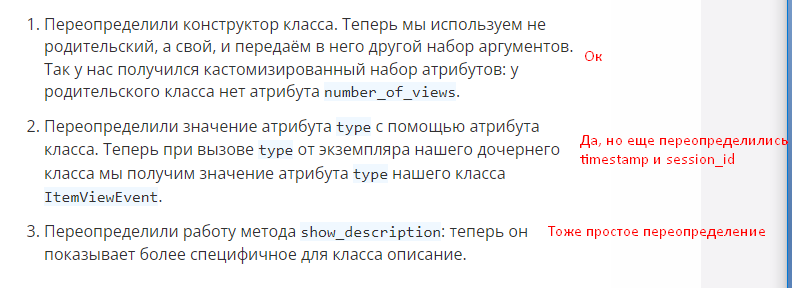
**B6.7. Наследование**

Классы умеют в наследование. Как оно работает? Чтобы один класс унаследовал методы и поля другого класса при написании класса нужно поместить название родительского класса в скобках:

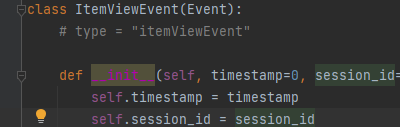


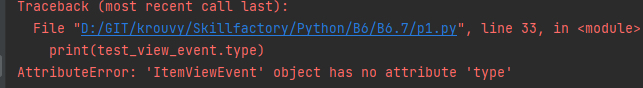
Важно отметить, что если назвать атрибут или метод в дочернем классе точно также, как и в родительском, то произойдет переопределение и наследования не будет.



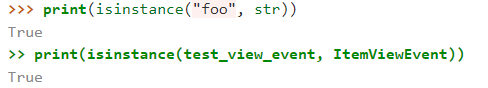


Хочется также добавить, что так как переопределился конструктор родителя, то те поля, которые инициализировались в родители тоже не передались. А значит если у дочернего класса закомментировать объявление поля type, то при попытке обратится к этому полю будет ошибка, так как это поле не унаследовалось, из-за того, что оно принадлежит другому конструктору. Но это касается только полей, которые связаны с родительским конструктором.





С помощью функции **isinstance**, можно проверить принадлежит ли какой-то конкретный экземпляр к тому или иному классу. В качестве атрибутов принимает экземпляр и тип данных.



Но слишком на это не полагаться. В каком-то смысле все в Python это объекты. Поэтому можно также наблюдать следующий вывод:



Наследовать можно и более одного класса. Важно только понимать, что порядок наследования важен, так как может происходить переопределение.

**B6.8. Наследование класса**

Блин, опять свою фигню написать захотел. Хорош. И написал ведь. Короче. Научился вызывать функции и переменные, которые инициализированы в файле по имени:



И аналогичное, но с полями классов. Но там кода больше. Заодно научился возвращать количество аргументов в объявленной функции. И количество дефолтных. И чуть попрактиковал классы. Могу в одном классе создать список, который будет заполняться создаваемыми в этом классе классами. Вот.