

# Занятие 18

ООП

# Что напечатают эти операторы?

```
print(11 > 0 is True)    # chained comparisons  
print(0 < 0 == 0)  
print(1 in range(2) == True)
```

```
print( (11 > 0) is True)  
print((0 < 0) == 0)  
print((1 in range(2)) == True)
```

# Задача 17-1

Создайте регулярное выражение, которое вылавливает только четные целые числа

## Задание 17-2

Создайте декоратор, которые переводит все текстовые аргументы функции в верхний регистр и возвращает их в виде списка текстовых аргументов.

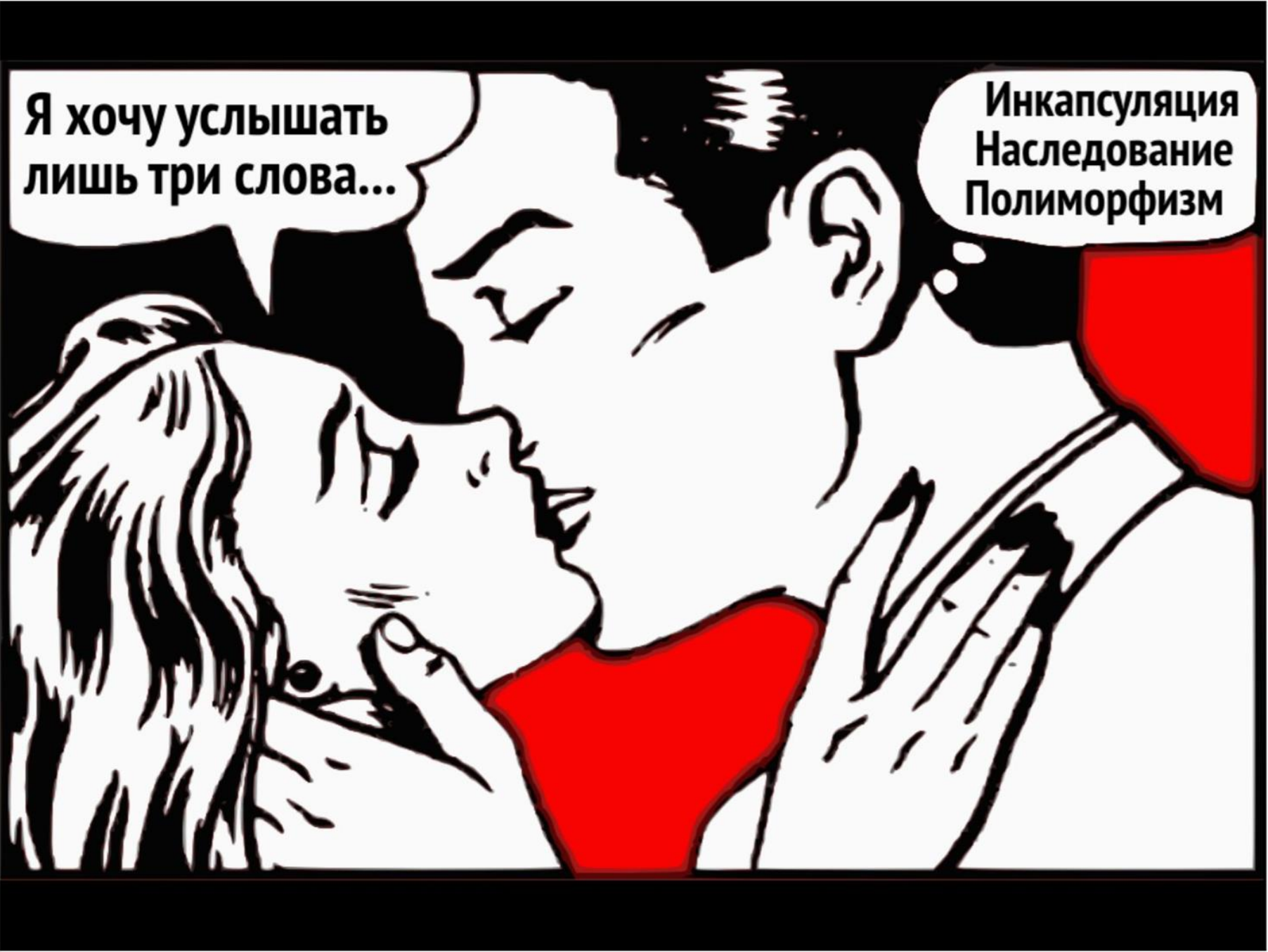
# Задание 17-3

- Создайте класс Shape, объекты которого имеют атрибуты Colour строка, например, «Красный», «Синий»; Square – площадь объекта
- Создайте несколько методов:
  1. Установить цвет объекта
  2. Запросить цвет объекта и напечатать его
  3. Задать площадь объекта
  4. Запросить площадь объекта

# ООП в Python

# Определение ООП

- Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
- Процедурное программирование — программирование на императивном языке, при котором последовательно выполняемые операторы можно собрать в подпрограммы, то есть более крупные целостные единицы кода, с помощью механизмов самого языка
- Когда надо выбирать ООП, когда ПП?



Я хочу услышать  
лишь три слова...

Инкапсуляция  
Наследование  
Полиморфизм



# Инкапсуляция, Наследование, Полиморфизм

- Полиморфизм: в разных объектах одна и та же операция может выполнять различные функции
- Инкапсуляция: можно скрыть ненужные внутренние подробности работы объекта от окружающего мира
- Наследование: можно создавать специализированные классы на основе базовых

# Классы

- Класс это пользовательский тип
- Для создания классов предусмотрен оператор class
- Члены класса называются атрибутами, функции класса – методами
- class ИМЯКЛАССА:

    переменная = значение

def \_\_init\_\_(self, x = 1, y = 'abc' ):    # конструктор для создания экземпляра.

Именованные параметры удобно использовать, чтобы не задавать параметры для создания экземпляра

def ИМЯМЕТОДА(self, ...):

    self.переменная = значение

    # атрибуты и методы можно добавлять по мере необходимости.

# Задание

- Создадим класс Shape
- Зададим его атрибуты – периметр и цвет
- И метод perі, который возвращает печатает периметр.
- В дальнейшем мы унаследуем от этого класса - два класса Треугольник и Прямоугольник, для которых будем уже вычислять периметр.

# Деструктор \_\_del\_\_

```
class Student:
```

```
    # конструктор
```

```
    def __init__(self, name):
```

```
        print('Inside Constructor')
```

```
        self.name = name
```

```
        print('Object initialized')
```

```
    def show(self):
```

```
        print('Hello, my name is', self.name)
```

```
    # деструктор
```

```
    def __del__(self):
```

```
        print('Inside destructor')
```

```
        print('Object destroyed')
```

```
# создать объект
```

```
s1 = Student('Emma')
```

```
s1.show()
```

```
# input()
```

```
# удалить объект
```

```
#del s1
```

```
#s1.show()
```

```
# Давайте выполним эту программу и посоздаем, и поудаляем объекты
```

# Класс

- Класс — универсальный, комплексный **тип данных**, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей).
- В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю.

# Объект (экземпляр, instance)

- Объект - это отдельный представитель класса, имеющий конкретное состояние и поведение, полностью определяемое классом.
- Каждый объект имеет конкретные значения атрибутов и методы, работающие с этими значениями на основе правил, заданных в классе.
- А можно ли где-то хранить список созданных объектов класса?

# Определение отношения объекта к классу

- Для определения, к какому классу относится объект, можно вызвать внутренний атрибут объекта `__class__`. Также можно воспользоваться функцией `isinstance` (рекомендуется этот вариант).
- `# Определить класс объекта`
- `print(isinstance(animal, Animal))`
- `print(animal.__class__)`
- `# Поля объекта`
- `print(dir(animal))`
- `# Давайте проверим это`

# Класс, объект

- Когда речь идёт об объектно-ориентированном программировании в Python, первое, что нужно сказать - это то что любые переменные любых типов данных являются объектами, а типы являются классами.
- Каждый объект имеет набор атрибутов, к которым можно получить доступ с помощью оператора “.”. Мы уже имеем опыта работы с атрибутами, пример списков:
  - `lst = [1, 2, 3]`
  - `lst.append(4)`



# Наследование

- Наследование – это свойство системы, позволяющее описать новый класс на основе уже существующего с частично или полностью заимствующейся функциональностью.
- Класс, от которого производится наследование, называется предком, базовым или родительским.
- Новый класс – потомком, наследником или производным классом.

# Одиночное наследование

- `class Tree(object):` `#Родительский класс помещается в скобки после имени класса`
- `def __init__(self, kind, height):`
- `self.kind = kind`
- `self.age = 0`
- `self.height = height`
- `def grow(self):`
- `self.age += 1`
- `class FruitTree(Tree):` `# Объект производного класса наследует все свойства родительского.`
- `def __init__(self, kind, height):`
- `super().__init__(kind, height) # Мы вызываем конструктор родительского класса`
- `def give_fruits(self):`
- `print ("Collected 20kg of {}".format(self.kind))`
- `f_tree = FruitTree("apple", 0.7)`
- `f_tree.give_fruits()`
- `f_tree.grow()`
- # **Создайте апельсиновое дерево, пусть оно дает 20 кг апельсинов**

# Задание

- Создайте два класса Triangle и Rectangle
- Добавьте необходимые атрибуты, стороны фигур
- Переопределите расчеты периметра каждой фигуры

# Rectangle / Class

```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

class Square(Rectangle):
    def __init__(self, length):
        # Для квадрата просто нужно передать один параметр length.
        # При вызове 'super().__init__()' установим атрибуты 'length' и 'width'.
        super().__init__(length, length)

# Класс 'Square' явно не реализует метод 'area()' и
# будет использовать его из суперкласса 'Rectangle'
sqr = Square(4)
print("Area of Square is:", sqr.area())
# Area of Square is: 16

rect = Rectangle(2, 4)
print("Area of Rectangle is:", rect.area())
# Area of Rectangle is: 8
```

Давайте реализуем класс Square, используя конструктор класса Rectangle

Проверьте, что метод периметр продолжает работать правильно

# Множественное наследование

- При множественном наследовании дочерний класс наследует все свойства родительских классов. Синтаксис множественного наследования очень похож на синтаксис обычного наследования.
- `class Horse: # А если определить метод Igogo?`
- `isHorse = True`
- `class Donkey: # А если определить метод Iaia?`
- `isDonkey = True`
- `class Mule(Horse, Donkey):`
- `pass`
- `mule = Mule()` # Какой из методов будет доступен? Igogo или Iaia? Проверьте это
- `mule.isHorse` # True
- `mule.isDonkey` # True

# Многоуровневое наследование

- Мы также можем наследовать класс от уже наследуемого. Это называется многоуровневым наследованием. Оно может иметь сколько угодно уровней.
  - В многоуровневом наследовании свойства родительского класса и наследуемого от него класса передаются новому наследуемому классу.
- 
- class Horse():
    - isHorse = True
  - class Donkey(**Horse**):
    - isDonkey = True
  - class Mule(**Donkey**):
    - pass
  - mule = Mule()
  - mule.isHorse # True
  - mule.isDonkey # True

# Задание

- В нашем классе Triangle создайте метод, который при создании объекта проверяет три переменный  $x$ ,  $y$ ,  $z$ , что из них можно составить треугольник.
- Требования: все числа должны быть больше нуля, сумма любых двух должны быть больше третьего.

# Полиморфизм

- Полиморфизм – это свойство системы использовать объекты с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.
- Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование одного и того же интерфейса для задания единого набора действий.
- У разных классов могут быть методы с одним и тем же названием, но выполнять они могут абсолютно различные действия.
- Например, только один подкласс должен иметь свой специфический метод, а остальные могут использовать метод суперкласса, тогда только у него переопределяется метод.



# Переопределим функцию сложения для разных классов

- `class X(str):`
- `def __init__(self, s):`
- `self.s = s`
- `def __add__(self, other):`
- `return other.s + self.s`
- `class Y(int):`
- `def __init__(self, s):`
- `self.s = s`
- `def __add__(self, other):`
- `return self.s * other.s`
- `x = X('aaa')`
- `z = X('bbb')`
- `y = Y(11)`
- `print(x + z) # Что напечатает?`
- `print(y + y)`

# Строковые методы `__str__` `__repr__`

- Дандеры для отображения объекта в виде строки вызываются при работе с функциями **`print()`** **`str()`**

- ```
class Car:
    def __init__(self, model, color, vin):
        self.model = model
        self.color = color
        self.VIN = vin

    def __str__(self):
        return f"Модель { self.model} с VIN номером {self.VIN}"        # user-friendly

    def __repr__(self):
        return f"Модель: { self.model} , VIN номер : {self.VIN}, цвет: {self.color}"        # более формальный
```

```
car = Car("Mercedes-benz", "silver", "WDB1240221J081498")
print(car)
print(str(car))
```

- # Выполните эти строчки. Переопределите методы `__str__` и `__repr__`. Проверьте их работу

# Строковые методы `__str__` `__repr__`

- ```
class Car:  
    def __init__(self, model, color, vin):  
        self.model = model  
        self.color = color  
        self.VIN = vin
```

```
car = Car("Mercedes-benz", "silver", "WDB1240221J081498")
```

- ```
print(car)
```

`<__main__.Car object at 0x7fe009b78a60>`
- ```
print(str(car))
```

`<__main__.Car object at 0x7fe009b78a60>`
- 1. Для преобразования строк в классах можно использовать дандер методы `__str__` и `__repr__`
- 2. В свои классы всегда следует добавлять метод `__repr__`.

# Принцип «Утиной типизации»

- Если кто-то крякает, как утка и ходит, как утка, то считаем, что это утка.
- Если у объекта есть функции, методы и свойства какого-то класса, то мы считаем, что его можно использовать как объект этого класса.

Например:

- Sequence: это как список list? Можно перебирать, можно индексировать?
- Iterable: можем ли мы использовать их в циклах?

Итерации iterable являются более общим понятием, чем последовательности. Все, что вы можете зациклить с помощью цикла `for .. in`, является итеративным.

Списки, строки, кортежи, множества, словари, файлы, генераторы, объекты диапазона, объекты `zip` и многое другое в Python являются итерируемыми iterable.

# Инкапсуляция

- Инкапсуляция – это свойство системы, позволяющее объединить данные и код в объекте и скрыть реализацию объекта от пользователя. При этом пользователю предоставляется только спецификация (интерфейс) объекта.
- Пользователь может взаимодействовать с объектом только через этот интерфейс.
- Пользователь не может использовать закрытые данные и методы.

# Инкапсуляция

- По умолчанию атрибуты в классах являются общедоступными(public), а это значит, что из любого места программы мы можем получить атрибут объекта и изменить его.
- class Person:
- def **\_\_init\_\_**(self, name):
- self.name = name # устанавливаем имя
- self.age = 1 # устанавливаем возраст
- 
- def display\_info(self):
- print(f"Имя: {self.name}\tВозраст: {self.age}")
- 
- 
- tom = Person("Pupkin")
- tom.age = 50 # изменяем атрибут age
- tom.display\_info() # Имя:Pupkin Возраст: 50

# Инкапсуляция

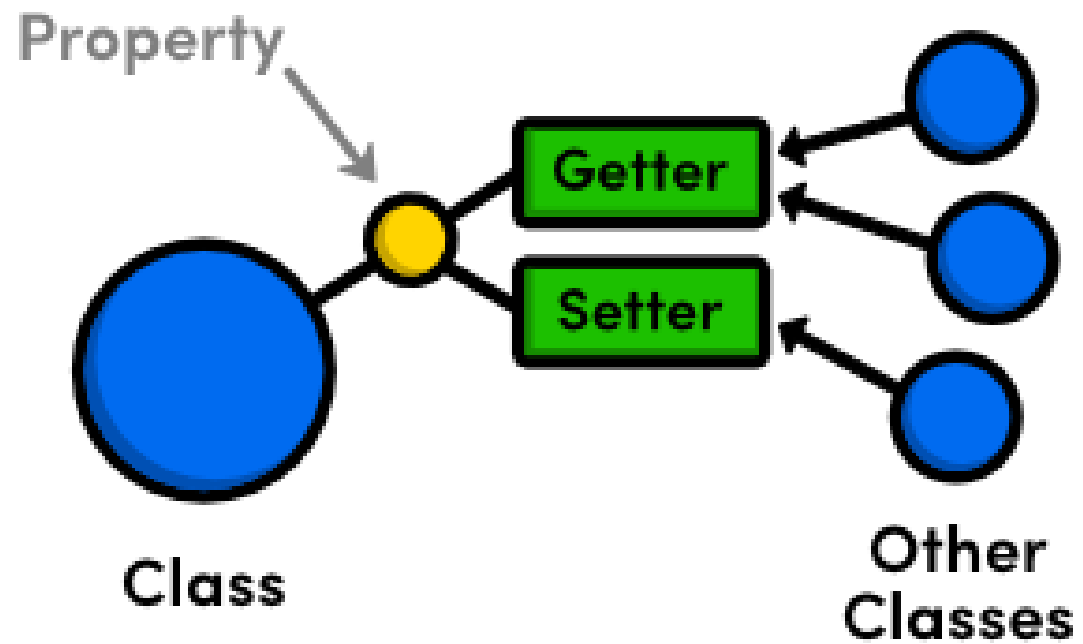
- Все объекты в Python инкапсулируют внутри себя данные и методы работы с ними, предоставляя публичные интерфейсы для взаимодействия.
- Атрибут может быть объявлен **приватным** (private) с помощью **нижнего подчеркивания** перед именем, но настоящего скрытия на самом деле не происходит – все на уровне соглашений.
- `class SomeClass:`
- `def _private(self):`
- `print("Это внутренний метод объекта")`
- `obj = SomeClass()`
- `obj._private()` # это внутренний метод объекта

# Два нижних подчеркивания (**double underscore**) - дандер

- # Если поставить перед именем атрибута два
- # подчеркивания, к нему нельзя будет обратиться напрямую.
- class SomeClass():
- def \_\_init\_\_(self):
- self.\_\_param = 42 # приватный атрибут
- obj = SomeClass()
- obj.\_\_param # AttributeError: 'SomeClass' object has no attribute '\_\_param'
- obj.\_SomeClass\_\_param # - обходной способ. Проверьте, что работает



# Методы доступа к свойствам



- class Person:
- def \_\_init\_\_(self, name):
- self.\_\_name = name # устанавливаем имя
- self.\_\_age = 1 # устанавливаем возраст
- def set\_age(self, age):
- if 1 < age < 110:
- self.\_\_age = age
- else:
- print("Недопустимый возраст")
- def get\_age(self):
- return self.\_\_age
- def get\_name(self):
- return self.\_\_name
- def display\_info(self):
- print(f"Имя: {self.\_\_name}\tВозраст: {self.\_\_age}")
- tom = Person("Tom")
- tom.display\_info() # Имя: Tom Возраст: 1
- tom.set\_age(25)
- tom.display\_info() # Имя: Tom Возраст: 25

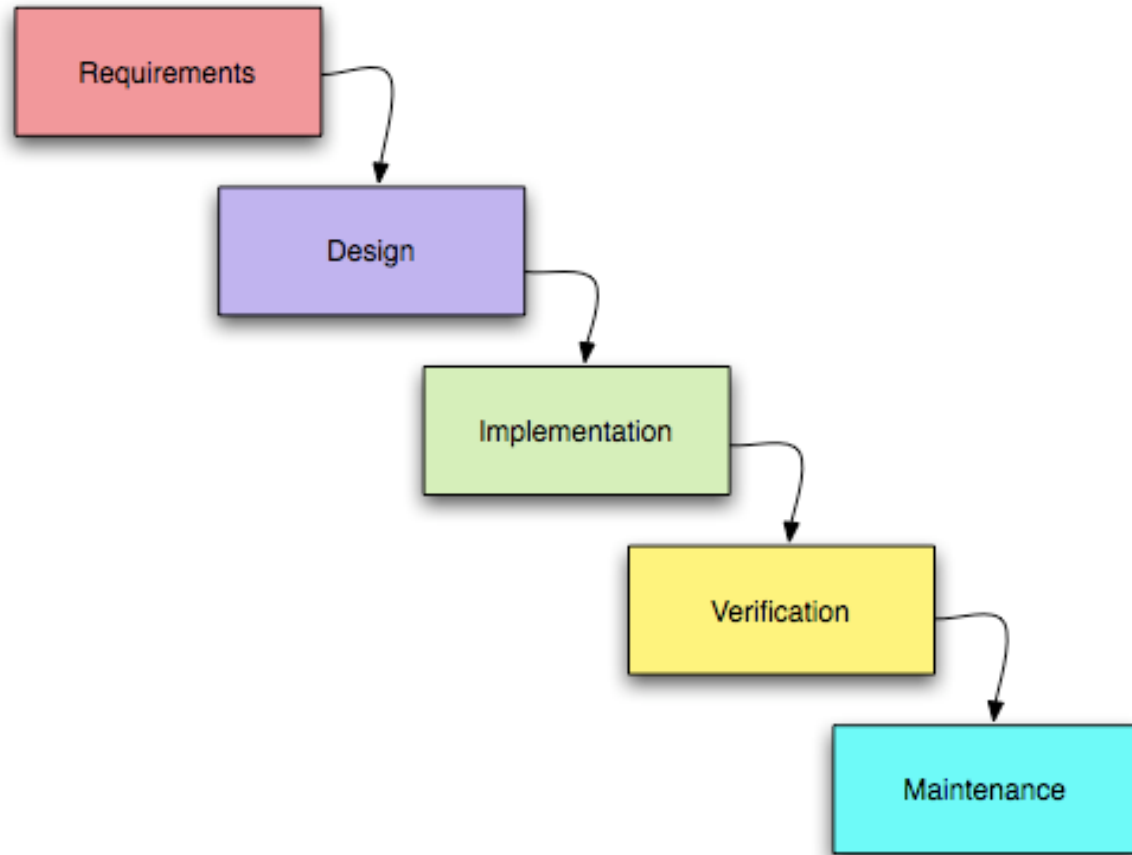
Выполните эту программу, убедитесь, что нельзя ввести недопустимый возраст

# Проектирование

В ООП очень важно предварительное проектирование. В общей сложности можно выделить следующие этапы разработки объектно-ориентированной программы:

1. Формулирование задачи.
2. Определение объектов, участвующих в ее решении.
3. Проектирование классов, на основе которых будут создаваться объекты.  
В случае необходимости установление между классами наследственных связей.
4. Определение ключевых для данной задачи свойств и методов объектов.
5. Создание классов, определение их полей и методов.
6. Создание объектов.
7. Решение задачи путем организации взаимодействия объектов.

# Каскадная модель – waterfall - водопад



- 1.Определение требований
- 2.Проектирование
- 3.Конструирование (также «реализация» либо «кодирование»)
- 4.Тестирование и отладка (также «**верификация**»)
- 5.Инсталляция, внедрение
- 6.Поддержка

Какие достоинства и какие недостатки?  
Где до сих пор можно встретить такую модель?  
Где до сих пор стоит придерживаться этой модели?  
Как можно пытаться преодолеть недостатки?

# Гибкая (agile) методология разработки - Scrum, Kanban и др.

- основополагающие принципы [Agile Manifesto](#)<sup>[5][6]</sup>:
- наивысшим приоритетом признается **удовлетворение заказчика** за счёт ранней и бесперебойной поставки ценного программного обеспечения;
- **изменение требований** приветствуется даже в конце разработки (это может повысить конкурентоспособность полученного продукта);
- **частая поставка работающего программного обеспечения** (каждые пару недель или пару месяцев с предпочтением меньшего периода);
- общение представителей бизнеса с разработчиками должно быть **ежедневным** на протяжении всего проекта;
- проекты следует строить вокруг заинтересованных людей, которых следует обеспечить нужными условиями работы, поддержкой и доверием;
- самый эффективный метод обмена информацией в команде — **личная встреча**;
- работающее программное обеспечение — лучший измеритель прогресса;
- спонсоры, разработчики и пользователи должны иметь возможность поддерживать постоянный темп на **неопределённый срок**;
- постоянное внимание к техническому совершенству и хорошему проектированию увеличивают гибкость;
- простота, как искусство не делать лишней работы, очень важна;
- лучшие требования, архитектура и проектные решения получаются у самоорганизующихся команд;
- команда регулярно обдумывает **способы повышения своей эффективности** и соответственно корректирует рабочий процесс.

# Задание

- Разработайте систему «Учебный процесс»
- Есть учитель Марьванна, есть ученики Петя и Вася, учитель учит учеников нескольким темам по ООП.
- Какие классы и методы будут полезны в этом?

# Учебный процесс

```
class Data:
    def __init__(self, *info):
        self.info = list(info)
    def __getitem__(self, i):
        return self.info[i]

class Teacher:
    def __init__(self):
        self.work = 0
    def teach(self, info, *pupil):
        for i in pupil:
            i.take(info)
            self.work += 1

class Pupil:
    def __init__(self):
        self.knowledge = []
    def take(self, info):
        self.knowledge.append(info)
```

- # продолжение программы

```
lesson = Data('class', 'object', 'inheritance', 'polymorphism',
'encapsulation')
```

```
marlvanna = Teacher()
```

```
vasy = Pupil()
```

```
pety = Pupil()
```

```
marlvanna.teach(lesson[2], vasy, pety)
```

```
marlvanna.teach(lesson[0], pety)
```

```
print(vasy.knowledge)
```

```
print(pety.knowledge)
```

# Напишите свой вариант учебного процесса

- Можно подглядывать и списывать, но только то, что понятно )))



# Задача 18

Разработать систему учета решения задач учениками курса «Разработчик на Питоне».

Проблема. Преподаватель каждый урок задает некоторое количество задач в качестве домашнего задания, для упрощения можно считать, что одну.

Каждый ученик решает каждую задачу. Переводит ее статус в решенную.

Преподаватель проверяет каждую задачу каждого ученика и либо подтверждает ее статус как решенную или меняет ее статус как не решенную.

Вопрос. Как спроектировать систему классов на Питоне для решения задачи учета.

Разработайте систему классов (Teacher, Pupil, Lesson, Task. Нужен ли класс Группа)

Разработайте систему атрибутов для каждого состояния каждого объекта.

Разработайте систему методов для каждого объекта.

Отчетность? Запросы? Начните с формулировки решаемой задачи – спецификации или технического задания. Затем спроектируйте классы, атрибуты, методы.

Протестируйте систему.