

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

Напишите программу программу, которая устраняет повторение повторение слов, т.е. результат результат должен быть следующим.

Напишите программу, которая устраняет повторение слов, т.е. результат должен быть следующим.

Задание 17-2

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

Текстовые аргументы – это строки в `args` и строковые значения в `kwargs`.

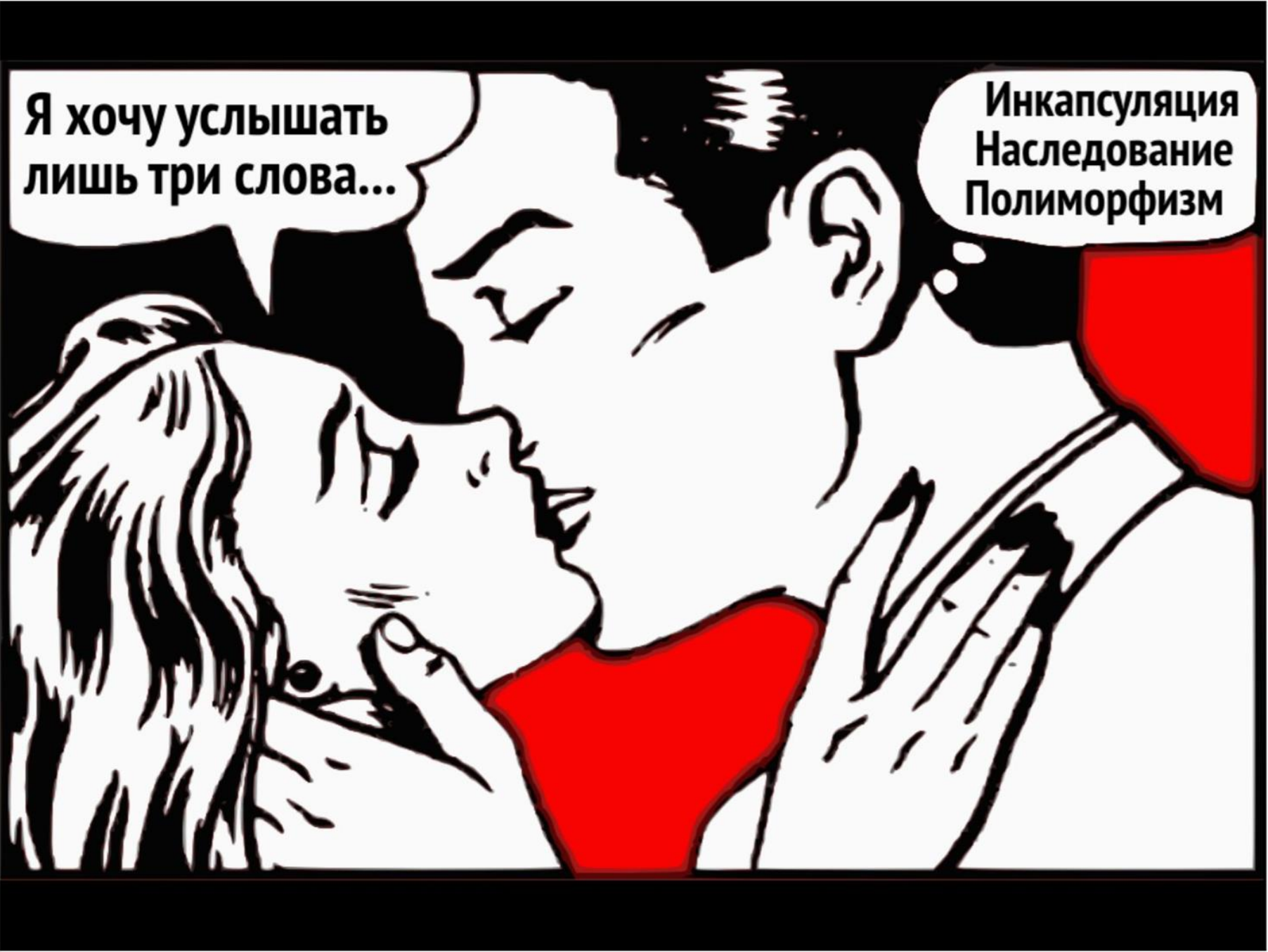
Задание 17-3

- 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, Абстракция

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

Абстракция: создавать интерфейсы и классы, которые определяют только те свойства и методы, которые необходимы для выполнения определенной задачи.

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 по умолчанию для создания экземпляра

 def ИМЯМЕТОДА(self, ...):

 self.переменная = значение

 # атрибуты и методы можно добавлять по мере необходимости.

Задание

Создадим класс Figure

Зададим его атрибуты – периметр и цвет

И метод `get_perimeter`, который печатает и возвращает периметр.

В дальнейшем мы унаследуем от этого класса - два класса: `Triangle` и `Rectangle`, для которых будем уже вычислять периметр.

Деструктор __del__

```
class Student:
```

```
def __init__(self, name):    # конструктор
    print('Inside Constructor')
    self.name = name
    print('Object initialized')
def show(self):
    print('Hello, my name is', self.name)
def __del__(self):    # деструктор
    print('Inside destructor')
    print('Object destroyed')
```

```
s1 = Student('Emma')    # создать объект
s1.show()
del s1                    # удалить объект
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).

```
print(isinstance(animal, Animal)) # Определить класс объекта
```

```
print(animal.__class__)
```

```
print(dir(animal)) # Поля объекта
```

```
# 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 заимствуемой 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 (f"Collected 20kg of {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, используя родительский класс Figure.

Добавьте необходимые атрибуты, стороны фигур.

Переопределите методы расчеты периметра каждой фигуры – для каждой фигуры (треугольник и прямоугольник) определите свой метод.

Rectangle / Square

```
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):
        super().__init__(length, length)
```

```
sqr = Square(4)
print(sqr.area())
rect = Rectangle(2, 4)
print(rect.area()) # Давайте проверим, что периметр считается правильно
```

Множественное наследование

При множественном наследовании дочерний класс наследует все свойства всех родительских классов.

Синтаксис множественного наследования очень похож на синтаксис обычного наследования.

```
class Horse:
```

```
    isHorse = True # атрибут класса
```

```
class Donkey:
```

```
    isDonkey = True # атрибут класса
```

```
class Mule(Horse, Donkey):
```

```
    pass
```

```
mule = Mule()
```

```
print(mule.isHorse) # True
```

```
print(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()
```

```
print(mule.isHorse) # True
```

```
print(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.

Введем атрибут `is_valid`, которая должна стать `True`, если треугольник может быть составлен, и `False`, если данные некорректны.

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, неформальный

 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 (последовательность): 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 объектом.

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
```



Выполните эту программу. Попробуйте ввести различные значения.

```
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} Возраст: {self.__age}")
```

```
tom = Person("Tom")
```

```
tom.display_info() # Имя: Tom Возраст: 1
```

```
tom.set_age(25)
```

```
tom.display_info() # Имя: Tom Возраст: 25
```

```
tom.__name
```

```
tom.__age
```

```
#tom._Person__name
```

```
#tom._Person__age
```

# Задание

Разработайте систему «Учебный процесс».

Есть учитель Марьванна, есть ученики Петя и Вася, учитель учит учеников нескольким темам по ООП.

Перед тем как писать код, ответьте на вопросы:

- Какие классы будут полезны?

- Какие атрибуты у них должны быть?

- Какие методы нужно запланировать?

- Как их реализовать?

# 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)
```

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

```
marlvanna = Teacher()
vasya = Pupil()
petya = Pupil()
```

```
marlvanna.teach(lesson[2], vasya, petya)
marlvanna.teach(lesson[0], petya)
```

```
print(vasya.knowledge)
print(petya.knowledge)
```

```
Какие плюсы и какие минусы этого
подхода? Что можно и нужно улучшить?
```

# Напишите свой вариант учебного процесса

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

# Задача 18

Разработать систему решения задач учениками курса «Разработчик на Питоне» и проверке их преподавателем.

1. Преподаватель каждый урок задает некоторое количество задач в качестве домашнего задания, для упрощения можно считать, что одну. Посылает их каждому ученику.
  2. Каждый ученик решает каждую задачу, переводит ее статус в решенную и посылает решение преподавателю.
  3. Преподаватель проверяет каждую задачу каждого ученика и подтверждает ее статус как решенную или меняет ее статус на нерешенную, и посылает результаты ученику.
- Разработайте систему классов (Teachers, Pupils). Можно создать другие классы, например, Tasks.
  - Разработайте систему атрибутов для каждого класса для хранения и использования описанных процессов.
  - Разработайте систему методов для каждого класса для реализации описанных процессов.
  - Проверьте ее работу на одном учителе и на 2-3 учениках.